Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica
Settore Scientifico Disciplinare: INF/01

Ph.D. Thesis

Adaptivity: Linguistic Mechanisms and Static Analysis Techniques

Letterio Galletta

Supervisor	Supervisor
Pierpaolo Degano	Gian-Luigi Ferrari

May 13, 2014

Abstract

Adaptive systems modify their behaviour in order to run *always* and *everywhere*. Their structure is therefore subject to continuous changes, which however could compromise the correct behaviour of applications and break the guarantees on their non-functional requirements. Effective mechanisms are thus required to adapt software to the new added functionalities and to changes of the operational environment, namely the *context* in which applications are plugged in. These mechanisms must also maintain the applications properties after adaptation occurs.

Consequently, a shift in programming technologies and methodologies is needed to manage adaptivity successfully. Since every system, be it adaptive or not, has to be programmed, programming languages need to natively support adaptivity. Furthermore, having adaptivity as a linguistic construct enables us to design and to develop more adequate verification tools that can help to prevent system failures due to erroneous or unexpected changes.

This thesis addresses adaptivity, adopting an approach firmly based on programming languages and formal methods. In particular, we have two main concerns. The first one consists of introducing appropriate linguistic primitives to describe the context and to express adaptation. The second one is about the design of verification tools, based on static analysis techniques, in order to ensure that the software maintains its consistency after adaptation.

To my family

Acknowledgments

I would like to my express my special appreciation and thanks to my advisers Professor Pierpaolo Degano and Professor Gian-Luigi Ferrari, who have been tremendous mentors for me. They taught me what scientific research is and how to carry it out. Without their guidance and persistent help this dissertation would not have been possible.

I would also like to give special thanks to my reviewers, Professor Dave Clarke, Professor Carlo Ghezzi and Professor David Van Horn, for reading a preliminary draft of this dissertation. Their advice helped me to improve the quality of this work.

I am indebted to Professor Giorgio Levi for teaching me the theory of programming languages and for introducing me to static analysis.

Finally, I would like to thank my parents, friends and colleagues who encouraged and advised me during my PhD. Without their continuous support the success of my PhD would not have been possible.

A programming language is low level when
its programs require attention to the irrelevant.
Alan J. Perlis

Contents

List of Definitions, Lemmata and Theorems

List of Figures

Introduction

Information Technology is so pervasive in our life and in our society that most of our everyday activities depend on it.

On the one hand computing devices of different shapes and sizes are now our “digital friends”. Portable media players, smartphones, tablets, laptops are some examples of devices we interact with every day. We exploit them to communicate with our contacts; to access services provided through the Internet; to send or receive emails and text messages; to chat with our buddies; to take pictures and share them with our friends through social media; to book a flight. All these activities have become so natural for us that we take them for granted.

On the other hand computing systems lie at the core of society infrastructures, such as financial, business, communication, transportation, defence and healthcare systems, to cite a few.

We are aware that Information Technology has simplified our lives, by increasing our communication capabilities and by automatizing a large number of repetitive everyday tasks. But it is important to notice that our dependency on computing systems exposes us to many unknown threats, which include system failures as well as attacks by hackers. These threats can have catastrophic consequences. For example, software mistakes or attacks may shut down large parts of financial or business infrastructures causing enormous economic damage. Now this scenario has become increasingly harder and it poses severe challenges to computer scientists. The main reason is that we require systems to run in environments only partially known, in which the anticipation of all possible environments conditions and the perfect understanding of software requirements are unachievable at design and development time. Furthermore, the complexity of systems gains every day so that it is very hard to design and to develop correct and safe software systems with millions of lines of code and with large numbers of interacting components. Additionally, once built and run such systems, their maintenance is highly a difficult task, because every change is required to occur at runtime while the system is operating.

Moreover, the ubiquity of mobile devices and mobile applications has been adding a further level of complexity. Indeed now applications can move and run in different execution environments which are unknown, open and possibly hostile and where the computational resources change dynamically.

It would be highly desirable to have tools and software engineering methodologies that support us in tacking these new challenges. By exploiting them, we would be able to design and develop systems that are verified correct and that are capable of managing themselves while they are running (almost) without any human involvement. Under this hypothesis, systems might continually monitor themselves and perform the operations which are the most adequate for facing the current situation. The vision in which computing systems are capable of managing themselves is called *autonomic computing*. One of the most notable feature in this vision is *self-adaptability*, i.e. the capability of systems to modify their own behaviour depending on the execution environment, namely the *context*. Self-adaptability is a notable requirement to support autonomic computing, and it represents itself an interesting research topic and a hard challenge.

We think that current mainstream development tools do not consider adaptivity a main concern, thus, they are not fully adequate to build effective and robust self-adaptive systems. Consequently, we need a change in the programming technologies and methodologies. Since every system, be it adaptive or not, has to be programmed, we believe that some first mechanisms to tackle adaptation must be provided at programming language level. We regard that addressing adaptation at this level pushes it down to elementary software components and allows describing fine-grain adaptability mechanisms. Furthermore, making adaptivity a main concern in programming languages enables us to design and develop verification tools that can help to prevent some kinds of system failures at earlier stages of the development process.

For these reasons, this thesis addresses some foundational issues of self-adaptability, by adopting an approach firmly based on programming languages and formal methods. In particular, we have two main concerns. The first one consists of introducing appropriate linguist primitives and abstractions to describe the environment hosting the application and to effectively express adaptation. The second concern is about the design of verification tools, based on static analysis techniques, in order to ensure that the "software behaves well" also after the adaptation steps. This means that software

- adequately reacts to changes in its execution environment;
- ensures some security guarantees.

To do that, we followed the research line of Context-oriented Programming (COP), a programming paradigm which has been recently proposed for developing adaptive applications. The two most noteworthy features of COP languages are the notions of context and of behavioural variation. The *context* is a description of the environment where the application is running, and it usually contains information about the device capabilities, users (e.g. preferences, profiles, etc.) and the psychical environment (e.g. location, level of noise, etc.). *Behavioural variations* are constructs specifying chunks of behaviour that can be *dynamically* activated depending on information picked up from the context. Thus, at runtime these chunks of behaviour alter the basic one of the application. Furthermore, behavioural variations can be composed, and the result of their *combination* determines the actual behaviour.

Although this new class of languages has proved quite adequate to program adaptation, there are still some open issues, which we detail later on. Among these we consider most impelling the following:

1. semantics foundations have not been sufficiently studied so far;

2. security and formal verification have not been main concerns in the design of COP languages;

3. primitives to describe the context are too-low level and not powerful enough to model complex working environments, which may include heterogeneous information;

4. the current implementation of behavioural variations, e.g. done through partial definition of methods or classes, is not sufficiently expressive.

The contribution of this thesis is to formally address the above issues. For (1) and (2), we introduce a development methodology which extends and integrates together technique from COP, type theory and model-checking. The kernel of our proposal is ContextML, a core functional language belonging to the ML family. Our choice is supported by the elegance of ML, which enables us to reason at a sufficiently abstract level, and by the following pragmatic reasons. Nowadays functional languages, e.g. F# and Scala, are actually used for development of real systems. Also many main stream languages as Python, Javascript, Ruby, to cite just a few, have borrowed ideas from the functional paradigm. Finally, as already noted, the semantics of functional languages is clear and well understood, and it enabled us to exploit a large number of known formal techniques from the literature.

The most notable features of ContextML include layers as first class values and behavioural variation at expression level; simple constructs for resource manipulation; mechanisms to declare and enforce security policies; and abstract primitives for communicating with other parties by message exchanging. Furthermore, we introduce a static technique that not only ensures software "behaves well" as pointed out above, but also some security properties. Software will then:

- adequately adjust its behaviour to context changes;
- safely use the available resources;
- correctly interact with the other parties.

Although ContextML and its companion static technique effectively tackle (1) and (2), issues (3) and (4) above remain still open. To attack them, we introduce $\mathrm{ML}_{\mathrm{CoDa}}$, an evolution and redesign of ContextML. Its main novelty consists of having two components: a declarative constituent for programming the context and a functional one for computing. This bipartite nature is motivated by the following observation: the context requires customised abstractions for its description, which are different from those used for programming applications. Think about web applications where there are specialized languages for the content visualization (e.g. HTML and CSS) and for the application logic. Our context is a knowledge base implemented as a Datalog program. Adaptive programs can therefore query the context by simply verifying whether a given property holds in it. In spite of the fact that this may involve possibly complex deductions. In addition, $\mathrm{ML}_{\mathrm{CoDa}}$ introduces a notion of open context, i.e. the holding properties not only depend on the software code, but also on the system, where it will run. This affects our verification machinery (as we will detail).

As for programming adaptation, $\mathrm{ML}_{\mathrm{CoDa}}$ includes *context-dependent binding* and *first-class behavioural variations*. The first feature allows program variables to assume different values depending on the context. The second one introduces *behavioural variations* as values, i.e. they can be bound to identifiers, and passed as arguments to, and returned by functions. This allows us to easily express dynamic and compositional adaptation patterns as well as reusable and modular code.

Furthermore, $\mathrm{ML}_{\mathrm{CoDa}}$ is equipped with a static technique to ensure that software will not fail due to an unexpected context arising at runtime. To effectively manage the open context of $\mathrm{ML}_{\mathrm{CoDa}}$, we perform the static analysis in two phases: at compile time (a type and effect system) and at loading time (a control flow analysis). During type checking we compute an abstraction over-approximating the capabilities required at runtime as an effect. When the software is about to run, this abstraction is exploited to verify that the actual hosting environment satisfies all the capabilities required by the application, so that no failure will arise during the execution.

We point out that here we are only interested in studying the introduction of new mechanisms for adaptivity inside programming languages from a foundational point of view, hence, we completely omit issues about the concrete implementation of the compilers and the verification mechanisms. Moreover, we do not consider event-based adaptivity. Often, context-aware systems are also event-based, where the occurrence of an event may trigger context changes and vice versa. Events can be both internal and external to programs and they denote the occurrence of a given action or condition, for example the activation of an hardware device or an input by the user. This situation could be naively encoded in our calculi, by introducing two threads sharing the context: the first listens for incoming events. Once an event is received, it modifies the context accordingly. The second thread runs the code of the program. In this thesis we only are interested in mechanisms for programming the second thread. Consequently, we will not discuss any linguistic primitive for events and we consider no mechanism for their management. Therefore, we only deal with single threaded programs, so allowing a manageable formal development of our calculi.

Thesis structure

The thesis is organized as follows:

Chapter 1 presents Context-oriented Programming and intuitively introduces the static analysis techniques which we use later in the text.

Chapter 2 introduces ContextML. The main contribution of this chapter is the semantics of ContextML and its annotated type system, assuring that well-typed program are able to react to context changes at runtime.

Chapter 3 extends the material of the previous chapter, by adding constructs for communication, for resource manipulation and for security policy enforcement. The main contribution of this chapter is the introduction of a static analysis technique, which includes a type and effect system and a model-checking procedure. We prove the soundness of our analysis and that programs passing checks comply with the security policies and correctly communicate with the other parties.

Chapter 4 is about the design and the formal description of $\mathrm{ML}_{\mathrm{CoDa}}$. The main contributions of the chapter are the semantics of $\mathrm{ML}_{\mathrm{CoDa}}$ and the definition of its two phases static analysis; we also prove its soundness. Furthermore, we define an analysis algorithm for the loading time part, prove its correctness, and compute its worst case complexity.

Chapter 5 concludes the thesis by presenting ongoing research activities and future work. Then, we extend our proposal to enforce security policies over the context.

Some portions of thesis were published in the following papers:

- A short version of Chapter 2 in

  Pierpaolo Degano, Gian Luigi Ferrari, Letterio Galletta, and Gianluca Mezzetti. Typing context-dependent behavioural variations. In Simon Gay and Paul Kelly, editors, *Proceedings Fifth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2012*, volume 109 of *EPTCS*, pages 28–33, 2012

- The material of Chapter 3 in

  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

  Chiara Bodei, Pierpaolo Degano, Gian Luigi Ferrari, Letterio Galletta, and Gianluca Mezzetti. Formalising security in ubiquitous and cloud scenarios. In Agostino Cortesi, Nabendu Chaki, Khalid Saeed, and Slawomir T. Wierzchon, editors, *Computer Information Systems and Industrial Management, CISIM 2012*, volume 7564 of *Lecture Notes in Computer Science*, pages 1–29. Springer, 2012

Previous work on static analysis, but not included in this thesis are:

Letterio Galletta and Giorgio Levi. An abstract semantics for inference of types and effects in a multi-tier web language. In Laura Kovács, Rosario Pugliese, and Francesco Tiezzi, editors, *Proceedings 7th International Workshop on Automated Specification and Verification of Web Systems, WWV 2011*, volume 61 of *EPTCS*, pages 81–95, 2011

Letterio Galletta. A reconstruction of a types-and-effects analysis by abstract interpretation. In *Proceedings Italian Conference of Theoretical Computer Science, ICTCS 2012*, 2012

Letterio Galletta. An abstract interpretation framework for type and effect systems, 2013. To appear in Fundamenta Informaticae

Chapter 1

Preliminaries

1.1 Adaptive Software

In this section we introduce the notion of adaptive software (see [ST09] for a complete survey) and we briefly review some language-based approaches which have been proposed to program this kind of system. There are many definitions of self-adaptability. For example, in a DARPA Broad Agency Announcement (BAA) [Lad97]:

> Self-adaptive software evaluates its own behaviour and changes behaviour when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible.

A similar definition is given in [OGT+99]:

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

Following [ST09] in this thesis we consider that adaptive software falls under the umbrella of autonomic computing, because self-adaptability is a fundamental constituent of this kind of system. The problem of self-adaptability been tackled by a variety of points of view such as control theory [KBE99], artificial intelligence [KW04], software engineering [ABZ12, PCZ13] and at the programming languages level. We follow this latter line of research. We strongly believe that language-level approaches are the most promising because they push the adaptation down to elementary software components. We will argue that at this level we can describe extremely fine-grain adaptability mechanisms and that it is easy to introduce fine-grain mechanized verification mechanisms, as those we will define in next chapters. The mainstream languages-level approaches to self-adaptability are *aspect-oriented programming* [KLM+97], *dynamic aspect-oriented programming* [PAG03] and *context-oriented programming* [HCN08]. We adopt this latter approach because it has an explicit notion of context, which better models the working environment hosting the software.

1.1.1 Autonomic Computing

The complexity of modern computing systems is such that even the most skilled system administrators can hardly manage and maintain them. Seeking for a solution, IBM released a manifesto [IBMon] in 2001, which advocated an overall rethinking of computing systems. Future systems are required decreasing the human involvement, by freeing administrators from the details of low-level operations and maintenance. The vision in which machines are able to self-manage is called autonomic computing [KC03, HM08]. The term "autonomic" comes from biology, in particular from autonomic nervous system. In the human body, the autonomic nervous system takes care of bodily functions that do not require our attention, for example heart rate and body temperature, so an autonomic system will maintain and adjust its operations in face of changing components, workloads, external conditions and failures without a conscious involvement. The essence of an autonomic system is therefore *self-management*. Under this hypothesis a system continuously monitors its own execution and performs the operations considered most adequate. For example, if the advertised features of an upgrade seem worth installing, a system will upgrade and reconfigure itself as necessary. If needed, the system reverses to an older version, based, for example on a regression test that checks whether the upgrade completed successfully or not.

According to [IBMon] self-management systems are characterized by the following properties:

- *self-configuration*. Autonomic systems configure themselves according to high-level policies specifying what is desired, not how to accomplish it. This means that when components are introduced they will incorporate themselves seamlessly and the rest of the system will adapt to their presence;

- *self-optimization*. Autonomic systems pro-actively seek ways to improve their own performance, efficiency and resource usage;

- *self-protection*. Autonomic systems automatically defend themselves against malicious attacks but also from end-users who inadvertently make unsafe software changes. Systems autonomously tune themselves to achieve and ensure an adequate security, privacy and protection level, and exhibit a proactive behaviour by anticipating security breaches and prevent them from occurring in the first place;

- *self-healing*. Autonomic systems are fully fault-tolerant and are able to detect, diagnose and repair localized software and hardware problems by installing the appropriate patches or isolating faulty components. It is important that as result of the healing process the systems are not further harmed, for example by the introduction of new bugs.

From an architectural point of view autonomic systems are massive collections of interactive elements, called *autonomic elements*. Each autonomic element is responsible for managing its own behaviour but it usually cooperates with others to perform its task. The interactions with others elements and with the environment are performed by exchanging signals and messages over predefined communication channels and in

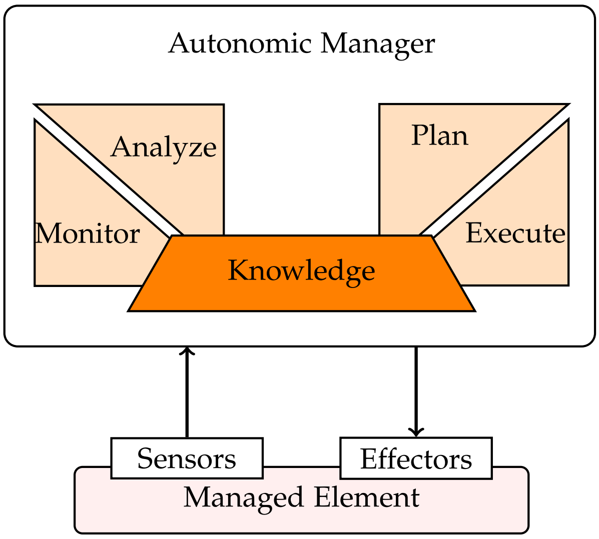


Figure 1.1: The structure of a typical autonomic element

accordance to fixed communication protocols. The internal behaviour of an autonomic element, as well as its relationships with others elements, are driven by policies that its administrator has embedded in it.

Structurally, autonomic elements are characterized by one or more *managed elements* coupled with a single (or multiple) *autonomic manager* that controls and manages them (see Figure 1.1). The managed element is essentially a standard software component, like those found in non-autonomic systems, except that it enables autonomic manager to monitor and control it. The autonomic manager follows a MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) control loop.

One of the most notable features of autonomic elements is self-adaptability, i.e. the ability of modifying behaviour depending on decisions taken by the autonomic manager. The problem of self-adaptability is one of hardest research challenges in designing and building autonomic systems.

1.1.2 Aspect-oriented Programming and Dynamic Aspect-oriented Programming

By analysing the source code of a computer program we discover that we can group it according to its responsibilities. For example, some parts of the code are responsible for implementing application logic, other ones for logging or for security checks. We call the application logic the core concern. The other ones instead implement program behaviour, functional or non-functional, that interacts with the core concern, but that are orthogonal to it. They are called cross-cutting concerns. Usually the code of cross-cutting concerns is intermixed with that of the application logic, causing *code scattering*. A program exhibits code scattering when the same or similar fragments of code are repeated throughout. It gives rise to the following issues:

- the code of the application logic is hard to maintain;

- the probability of error in coding increases and identifying errors becomes hard;
- any change of cross-cutting concerns can require changes in many source files.

Aspect-oriented programming (AOP) is a paradigm that decreases the code scattering by allowing us to separate and organize in stand-alone modules, called *aspects*, the orthogonal code, by decoupling it from the application logic. Aspect-oriented programming is based on a *joint point model*, specifying the following three notion:

join point position in a program where the additional features provided by an aspect can be inserted;

advice tool for specifying code to run at join point;

pointcut a pattern determining the set of join points where a given advice has to be inserted.

As a typical example of aspect-oriented programming, consider the *AspectJ* application shown in Figures 1.2 and 1.3. AspectJ [KHH^{+}01] is an extension of Java [GJSB05]. The application is a toy example taken from [Li05] showing code that can be found in a typical accounting system. The core application logic is in the `transferFunds()` method that transfers funds between two accounts by debiting from one account and then crediting to the other. `LogTransfer` defines an aspect implementing a logging mechanism. We define a pointcut called `transfer()` that matches any invocation of the `transferFunds()` method. The rest of code in `LogTransfer` defines two pieces of advice that are executed before and after the invocation of `transferFunds()`, respectively, realizing the logging mechanism. The link between join points and advice is performed during the compilation of the program.

Dynamic aspect-oriented programming allows adding (also aspects previously unknown to the system) and removing aspects to and from programs at runtime, hence, enabling us to perform adaptation of the system behaviour. This capability to dynamically activate or to deactivate advice has made dynamic aspect-oriented programming attractive for adaptive systems, especially for the development of autonomic ones.

In the literature there are also papers about the semantics foundation of AOP. Since AOP is not the topic of this thesis, we refer the interested reader to e.g. [WZL03, WKD04, LWZ06].

Although AOP adequately supports dynamic adjustments in a program, it has no mechanisms to explicitly represent the working environment hosting the application. Consequently, programmers need to implement it by themselves, providing mechanisms to intercept and react to the changes. For this reason, we have preferred adopting a paradigm where the notion of the context is within the language.

1.1.3 Context-oriented Programming

Context-oriented programming [CH05, HCN08] is a programming paradigm recently proposed as a viable approach to development of systems that are context-aware, i.e.

```
import com.ibm.dw.tutorial.aop.accts.CustomerAccount;

public class AccountManagerAOP {

   protected static final int ACCOUNT1 = 0;
   protected static final int ACCOUNT2 = 1;

   public static void main(String[] args) {
      AccountManagerAOP mgr = new AccountManagerAOP();
      mgr.transferFunds(ACCOUNT1, ACCOUNT2, 5);
   }
   public void transferFunds(int fromAcct, int toAcct, int amount) {
        CustomerAccount from = findAccount(fromAcct);
        CustomerAccount to = findAccount(toAcct);
        from.debit(amount);
        to.credit(amount);
   }

   protected CustomerAccount findAccount(int acctNum) {
      // simple stub code for tutorial
      return new CustomerAccount(acctNum, 10);
   }
}
```

Figure 1.2: A toy accounting system that shows the ideas underlying aspect-oriented programming

```
import com.ibm.dw.tutorial.aop.logging.SystemLogger;
import com.ibm.dw.tutorial.aop.accts.CustomerAccount;

public aspect LogTransfer {
  SystemLogger sysLog = SystemLogger.getInstance();

  pointcut transfer():
      call(* AccountManagerAOP.transferFunds(..));

  before(): transfer() {
     Object[] args = thisJoinPoint.getArgs();
     sysLog.logDetails(
           "transferFunds START - from Acct#" +
         args[0] + " to Acct#" + args[1] + " for $" +
         args[2]
     );
  }
  after(): transfer() {
     Object[] args = thisJoinPoint.getArgs();
     sysLog.logDetails(
         "transferFunds END - from Acct#" +
         args[0] + " to Acct#" + args[1] + " for $" +
         args[2]
    );
  }
}
```

Figure 1.3: A toy logging system for the `AccountManagerAOP` class defined by using aspect-oriented programming mechanisms

able to dynamically adapt their behaviour depending on changes occurring in their execution environment. Managing context-awareness is critical not only for autonomic computing but also for modern mobile applications and internet services. For example, consider applications that are location-based, situation-dependent or deeply personalized. Note that adaptation to the current context is an aspect that crosscuts the application logics and often it is orthogonal to the standard modularization mechanism provided by languages.

Currently, the only way to make the behaviour of a program context-dependent consists of modelling the context using a special data structure, which can answer to a fixed number of queries provided as methods or functions over it. Usually, to test the results of queries conditional statements, e.g. `if`, are exploited. This naive approach has at least two drawbacks:

1. the programmer has to implement its notion of context and the corresponding operations by himself;
2. the extensive use of conditional constructs to carry out adaptation fails to achieve a good separation of cross-cutting concerns.

Other approaches, which exploit traditional programming languages, are special design patterns that encapsulate the context-dependent behaviour into separate objects whose instantiation depends on the context [PCZ13]. Even if these approaches allow achieving a good separation of concerns, they make the use of automatic verification tools difficult.

Instead, context-oriented languages are designed with suitable constructs for adaptation and grant expressing context-dependent behaviour in a modular and isolating manner reducing the problem of code scattering. Furthermore, suitable constructs enable the compiler to optimise the code and the virtual machine to carry out automatised verification in order to assure that programs keep their correctness after the adaptation.

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

A layer is an elementary property of the context. Indeed, the context is a set of layers that can be activated and deactivated at runtime. Usually, the concept of *layer* is a language construct to identify an group related behavioural variations: activating/deactivating a layer corresponds to activating/deactivating the corresponding behavioural variation.

In [HCN08] Hirschfeld et al. have listed the essential features which a language must supply in order to support context-oriented programming; these include means:

- to specify behavioural variations
- to group variations in well defined and isolated layers;
- to dynamically activate/deactivate behavioural variations based on context;

- to explicitly and dynamically control the scope of behavioural variations.

They also have noted that layers[1] have to be first class objects in the language, i.e. they can be bound to variables, passed as argument and retuned by functions. This feature is especially required to allow different parts of a system to communicate and perform runtime adaptability.

Example 1.1.1. To clarify the main ideas of context-oriented programming we discuss a toy example taken from [HCN08, SGP11] and written in ContextJ [AHHM11], a context-oriented version of Java. The example is shown in Figures 1.4 and 1.5. We define a `Person` class and a `Employer` class and we use the context to capture people roles and their changes. Both classes declare a method `toString()` that is redefined inside the layers `Address` and `Employment`. When the method `toString()` is invoked on objects of `Person` class the implementation of the method to execute is chosen accordingly to the currently active layers. ContextJ provides the construct `with` to perform the dynamic activation of layers. When `toString()` is invoked on `somePerson` object the layers `Adress` `Employment` are active. These activations affect the method dispatching mechanism that searches inside layers in reverse activation order. In the example, the activation of `Employment` layer causes the printing of additional information about the person's employment, while the `Address` layer causes the printing of information about the job. In the code the invocation to `proceed()` is similar to the one of `super` in Java and forces the execution of the method in the next active layer. If there are no further active layers the original method is called. It is worth noting that layer activation performed by the `with` construct is dynamically scoped and affects the control-flow of the program: indeed, it modifies the behaviour not only for method calls systematically inside the code block, but also for the calls triggered in turn.

Example 1.1.2. As a further example, consider the toy adaptable storage server in Figure 1.6 (left), taken from [SGP12]. The server is implemented by a class `Storage` providing the method `getItem`, which allows one to retrieve data stored on the disk trough a key. The basic implementation of this method, the last one in the listing, simply looks up the object from the disk. However, this basic behaviour can change depending on the context hosting the server: the layer `logLayer` indicates that every access to the storage must be recorded in a log file; and the layer `cacheLayer` exploits a cache on the main memory to speed up the information retrieval. In Figure 1.6 (right) we create a new instance of the `Storage` class and invoke the method `getItem`, first in a empty context, and then in a context where the layers `logLayer` and `cacheLayer` are active. The output of this execution is shown in the comment under the code: we can understand by the prints how the activation of layers alter the control-flow in the body of the method `getItem`.

Inspired by the pioneering work of Costanza, a large number of COP proposals emerged, each of which addresses the problem of behavioural variation modularization and dynamic layer activation in different ways. To give an idea of the features offered

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

```
class Person {
  private String name, address;
  private Employer employer;

  Person(String newName, String newAddress, Employer newEmployer) {
    this.name = newName;
    this.address = newAddress;
    this.employer = newEmployer;
  }

  String toString(){ return "Name: " + name; }

  layer Address {

    String toString() {
      return proceed() + "; Address: " + address;
    }
  }

  layer Employment {

    String toString() {
      return proceed() + "; [Employer] " + employer;
    }
  }
}

class Employer {
  private String name, address;

  Employer(String newName, String newAddress){
    this.name = newName;
    this.address = newAddress;
  }

  String toString() { return "Name: " + name; }

  layer Address {

    String toString(){
      return proceed() + "; Address: " + address;
    }
  }
}
```

Figure 1.4: The `Person` and `Employer` class in ContextJ language

```
Employer vub = new Employer("VUB", "1050 Brussel");

Person somePerson =
    new Person("Pascal Costanza", "1000 Brussel", vub);

with(Address){
  with(Employment){
    System.out.println(somePerson);
  }
}
/*
[Output]
Name: Pascal Costanza; Address: 1000 Brussel;[Employer] Name: VUB;
Address: 1050 Brussel
*/
```

Figure 1.5: An execution of the program in Figure 1.4

```
class Storage{
  Cache cache = ...

  // Other methods

  Object getItem(int key){
    println("Disk lookup");
    Object item;
    // retrieve item from the disk
    return item;
  }

  layer logLayer {
    Object getItem(int key){
      println("Logging");
      // Send info to the log manager
      return proceed(key);
    }
  }

  layer cacheLayer {
    Object getItem(int key){
      println("Cache lookup");
      result = cache.get(key);
      if(result == null){
        result = proceed(key);
        cache.put(key, result);
      }
      return result;
    }
  }
}
```

```
Storage s = new Storage();

// No active layers
s.getItem(10);

// cacheLayer active
with(cacheLayer){
  s.getItem(10);
}

// logLayer and cacheLayer active
with(logLayer,cacheLayer){
  s.getItem(10);
}

/*
-- Execution of the above code--

> Disk Lookup

> Cache Lookup
> Disk Lookup

> Logging
> Cache Lookup
> Disk Lookup
*/
```

Figure 1.6: A toy adaptable storage server implemented in ContextJ

by the paradigm, we present a short overview of the most significant design choices concerning *activation mechanism* and *behavioural variation modularization*. See [SGP12] for more details about the different language proposals, [AHH+09] for an analysis of some implementations and their performance and [GPS11] for an comparison about the implementation of adaptive software by exploiting traditional languages and COP languages.

In general, we distinguish two *layer activation strategies* namely *global* and *local* activation. In the first case, there exists a global and shared context and the activation of a behavioural variation affects the control-flow of all threads. In the second case, there exists different local contexts, e.g. one for each thread or for each object of the applications, and the activation affects only the behaviour of entities depending on the modified context.

Regarding the *extent of layers activation*, we can identify two approaches: *dynamically scoped activation* and *indefinite activation*. The first is the approach proposed by the most COP languages. It is performed by the `with` statement that makes a sequence of layers actives in its code block. The activation influences also the nested calls and obeys to a stack discipline: whenever a new layer is activated, it is composed with the others by pushing it on the stack. When the scope expires the most recent layers are popped by retrieving the previous configuration. This behaviour allows programmers to properly delimit which parts of the program have to change their behaviour.

The *behavioural variation modularization* depends on the *layer declaration strategy*. We distinguish two layer declaration strategies, namely *class-in-layer* and *layer-in-class*. The first denotes layer declarations where the layer is defined outside the lexical scope of the modules for which it provides behavioural variations. This strategy allows layer encapsulation in dedicated and isolated modules. The second declaration strategy, instead, supports the declaration of a layer within the lexical scope of the module it augments, allowing modules definition to be completely specified. For example in Figure 1.4 the layers `Employment` and `Address` are defined inside the class `Person`.

1.1.4 Context-oriented programming for autonomic systems

Salvaneschi et al. [SGP11] proposed Context-oriented Programming as framework upon build autonomic systems. The underlying idea is to use context-oriented mechanisms to accomplish the adaptability requirements of the managed element by implementing it with a context-oriented language. When the control-flow enters in a `with` statement, the autonomic manager is queried for the active layers and the code in the scoped block is automatically adapted:

```
with(AutonomicManager.getActiveLayers()){
  // Dynamic adapted code
}
```

In this way the autonomic manager directly manages the actual adaptation that should be performed. This model requires that the designers precisely identify which parts of the code have to be included inside a `with` statement.

Furthermore, it is based on the strong hypothesis that all possible adaptations are known in advance and that all possible relevant changes in the execution environment can be anticipated at design time. This hypothesis is very strong and could bring down the model applicability to a reduced class of systems. They also highlighted that current context-oriented programming languages are not fully adequate for implementing autonomic systems, but that further improvements and developments are required. For example, in certain autonomic applications it could be necessary to express constraints on layers: two layers can conflict with each others or a dependency between them can exist, so that the (de)activation of a layer require the (de)activation of other one. Some prototype implementations including layer dependency enforcement have been proposed, e.g. in [CD08], but they are still premature. Another open issue is that layer activation in accordance to the program control-flow is often insufficient to express the adaptation needed by an autonomic systems since context changes are often triggered from external events. These situations require language mechanisms able to express the asynchronous event-driven adaptability. Recently, to cope this issue EventCJ was proposed in [KAM11], but it is still a prototype. For further details about the model and its corresponding open questions see [SGP11].

1.1.5 Calculi for context-oriented languages

So far most of the research efforts in the field of context-oriented programming have been directed toward the design and the implementation of concrete languages. To the best of our knowledge only a limited number of papers in the literature provides a precise semantic description of these languages. Here, we briefly review the most significant contributions. Later on we compare our proposals, i.e. ContextML and $\mathrm{ML}_{\mathrm{CoDa}}$ with them.

In [CS09] Clarke and Sergey have defined *ContextFJ*, an extension of Featherweight Java [IPW01] that includes layers, scoped layers activation, and deactivation. They have not considered constructs for expressing inheritance and have adopted a class-in-layer strategy to express behavioural variations. Since layers may introduce new methods not appearing in classes, they have also defined a static type system ensuring that there exists a binding for each dispatched method call.

In [HIM11] Hirschfeld et al. have defined a different Featherweight Java extension including inheritance and still exploiting a class-in-layer strategy. Also in this case, a static type system has been specified to statically prevent erroneous invocations at runtime. Their type system is much more restrictive than that of [CS09], 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. In [IHM12] Igarashi et al. have addressed the restriction of [HIM11] and extended the type system to handle dynamic layers composition.

Featherweight EventCJ [AKM11] is another calculus inspired by Featherweight Java introduced to formalize event-based layer activation. This mechanism allows activating layers in reaction to events triggered by the environment.

Contextλ [CCT09] extends the λ-calculus with layer definition, activation/deactivation and a dispatching mechanism. However, the calculus does not include high-order behavioural variations. Contextλ is exploited to study the issues deriving from the combination of closures and the special `proceed` construct, a sort of `super` invocation in object oriented languages [HCN08]. The problem arises when a `proceed` appears within a closure that escapes the context in which 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 [CCT09] several approaches to deal with this semantically relevant problem have been proposed, however no completely satisfactory solution has been put forward yet and this question is still open. Note that the `proceed` construct is strictly related to the idea of representing the context as a stack of layers.

1.1.6 Context-oriented languages open issues

Besides the limitations described in [SGP11] concerning the application of COP language as programming tool for autonomic computing, we identify the following restrictions as the most limiting for the development of complex adaptive software:

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

2. Primitives used to describe the context are too low-level and not enough powerful to model complex working environment. Indeed, layers are binary predicates, therefore, they cannot accurately identify the amount and the measure of different and structured pieces of information. Consider the storage server in Figure 1.6. There, the layer `cacheLayer` in the context represents only the fact that we are using a cache, but it does not give any information about the size or other features of the cache. So we believe that layers in this form are sometimes inadequate to program complex adaptive applications, where the context may contain structure information. Although in many implementations layers are implemented as objects so they can have a own state, this information do not affect the dispatching mechanism. Thus, there is no easy way to express adaptation based on the data carried by a layer.

3. Although behavioural variations are a basic notion in COP, they are not first class citizens in the languages. Usually, they are expressed as partial definition of modules in the programming language that underlies the actual COP language, such as procedures, classes or methods as we saw in the previous examples. However, behavioural variations are not values and one cannot easily manipulate them, e.g. passing them to functions as arguments, or binding them to variables. Instead, we believe that a language especially designed for adaptivity should have constructs to manipulate behavioural variations, for this reason one of our calculi provides such a feature.

4. Finally, to the best of our knowledge security issues have never been considered within COP languages. In particular, there is no mechanism to declare and enforce security polices over the context. We believe that this is an important open

issue because security could affect the adaptive capabilities of an application. For example, a security policy could prevent a behavioural variation to be activated.

In the following chapters, we address the above weakness, by adopting a language and formal method-based approach. In particular, ContextML attempts to tackle both (1) and (4) (Chapters 2 and 3). Whereas ML$_{\text{CoDa}}$ (Chapter 4) mainly deals with (1), (2), (3) and (4) by introducing a completely new treatment of COP primitives. Both our proposals support automatized verification.

1.2 Static analysis techniques

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

The purpose of static analysis is to acquire information about the runtime behaviour of a program without actually executing it but only by examining its source code. The results of an analysis can be used to optimize the execution, to discover errors in the code or to mathematically prove properties about programs.

Typical examples are

- data-flow analysis [NNH05, KSS09] which determines for each point in the code properties about the possible computed values;
- Flow Logic [NN02, BDNN01] that predicts safe and computable approximations to values that a program may compute during its execution;
- type systems and their extensions (type and effect systems [NN99], dependent types [XP99], refinement types [BBF$^+$11]) which compute for each program phrase its type augmented with a semantic annotation;
- (software) model-checking [CGP00, JM09] which allows algorithmically verifying whether the execution of a program[2] satisfies a property expressed by a temporal logic formula;
- abstract interpretation [CC77, CC79] through which we can systematically derive from the semantics of a program a sound analysis by an abstraction process.

These methods have been widely used to statically verify that programs satisfy a given specification both to detect unsafe and malicious behaviour. In the rest of this section we will explain in more detail the ideas underlying type systems, model-checking and Flow Logic to make the formal development in later chapters more clear.

1.2.1 Type Systems

Among formal methods used to ensure that a system behaves correctly, type systems are among the most popular and the most largely used. In [Pie02] a type system is defined as follows

[2] A special automaton (Kripke structure) needs to be derived from source code of the program

> *A type system is a tractable syntactic method for proving the absence of certain program behaviour (type errors) by classifying phrases according to the kinds of values that they compute.*

The definition of type error depends on the specific language and type system. For example, type systems for functional languages often prevent the application of a function to arguments for which it is not defined and the application of non-functional values.

Type systems belong to the class of deductive systems where the proved theorems are about the types of programs. The formal definition of a type system consists of the following elements: type syntax, type environment, typing judgement, typing rules, soundness theorem and type-checking algorithm.

Type syntax and type environment Each programming language makes available different kinds of values. A type can be thought as a description of a collection of homogeneous values. The type syntax describes the basic types and the type constructors which can be used to obtain new types from existent ones. Usually the type syntax is specified by a context-free grammar whose language is the set of all possible types. For example,

$$\mathtt{t} ::= \mathtt{integer} \mid \mathtt{boolean} \mid \mathtt{t}_1 \to \mathtt{t}_2 \mid \mathtt{t}_1 \times \mathtt{t}_2$$

defines type `integer` and `boolean` and the ones obtained by recursive applications of constructors → and ×.

A type environment stores the associations between program variables and the types of the values which these variables denote. Formally, a type environment is a list of bindings, i.e. pairs (variable, type), recursively defined as

$$\Gamma ::= \emptyset \mid \Gamma, \mathtt{x} : \mathtt{t}$$

where $\emptyset$ is the empty environment and $\Gamma, \mathtt{x} : \mathtt{t}$ is the environment Γ extended with the binding between $\mathtt{x}$ and $\mathtt{t}$.

Judgements Judgements are formal assertions about the typing of program phrases and correspond to the well-formed formulas of the deduction systems. A typical judgement has the form

$$\Gamma \vdash \mathcal{A}.$$

It means that Γ entails $\mathcal{A}$, where $\mathcal{A}$ is an assertion whose free variables are bound in Γ. Usually, $\mathcal{A}$ asserts that there exists a relationship between a program phrase (*has-type judgement*) or between two types (*subtype-of judgement*).

Typing rules Typing rules establish relationships among judgements and assert the validity of a certain judgement on the basis of others judgements that are already known to be valid, by inductively defining a logical theory about program types. Each typing rule has the form:

$$\textsc{(rule name)}\quad \frac{\Gamma_1 \vdash \mathcal{A}_1 \qquad \ldots \qquad \Gamma_n \vdash \mathcal{A}_n}{\Gamma \vdash \mathcal{A}}$$

where the judgements $\mathtt{\Gamma_i} \vdash \mathcal{A}_i$ above horizontal line are called premises, and the single judgement $\Gamma \vdash \mathcal{A}$ below the line is called the conclusion. When all of the premises are valid then it is possible to assert that the conclusion holds. Rules without premises are allowed and are used to assert judgements that are always valid (axioms).

A derivation is a tree of judgements having the form

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{\Gamma_{11} \vdash \mathcal{A}_{11} \ldots \Gamma_{1j} \vdash \mathcal{A}_{1j}}{\Gamma_{12} \vdash \mathcal{A}_{12}} \\ \vdots \end{array}}{\Gamma_{1m} \vdash \mathcal{A}_{1m}} \quad \cdots \quad \dfrac{\begin{array}{c}\dfrac{\Gamma_{n1} \vdash \mathcal{A}_{n1} \ldots \Gamma_{1k} \vdash \mathcal{A}_{1k}}{\Gamma_{n2} \vdash \mathcal{A}_{n2}} \\ \vdots \end{array}}{\Gamma_{nm} \vdash \mathcal{A}_{nm}}}{\Gamma \vdash \mathcal{A}}$$

where leaves (at the top) are axioms and where each internal node is a judgement obtained from the ones immediately above it by applying some rules. A judgement is valid if and only if it can be obtained as the root of a derivation. Hence, establishing that a program phrase $\mathtt{M}$ has type $\mathtt{t}$ is the same as finding a derivation with root $\Gamma \vdash \mathtt{M} : \mathtt{t}$.

Soundness theorem As noted above, the main goal of a type system is to prevent the occurrence of type errors during the execution. Traditionally, two classes of type errors are identified [Car04]:

trapped errors that can be caught by the abstract machine of language and that cause the computation to stop immediately (e.g. division by 0)

untrapped errors that go unnoticed for a while and that later cause arbitrary behaviour.

An effective type system should ensure that no untrapped error and no trapped error designated as forbidden occur.

Unfortunately, the definition of type system is not enough to guarantee that type errors do not happen at runtime. A formal proof of a soundness theorem is required to establish a connection between the semantics of the language and types, ensuring that the type system achieves its objectives. Any proof of the soundness theorem is intimately tied to the formulation of the semantics of the language. If the semantics is specified with an operational style, usually the soundness theorem is proved in two steps, commonly known as the *progress and preservation lemmata*:

progress a well-typed phrase is a value or it can take a step according to the evaluation rules

preservation if a well-type phrase takes a step of evaluation the resulting phrase is also well-typed (types are preserved under reduction).

Type checking and type inference algorithm Type checking is the problem of verifying that a given term $\mathtt{M}$ has type $\mathtt{t}$. Instead, type inference is the problem of finding for a given term $\mathtt{M}$ a type $\mathtt{t}$, i.e. finding a derivation for the judgement $\Gamma \vdash \mathtt{M} : \mathtt{t}$. In static typed languages these algorithms are part of the compiler front-end. Although their implementation can use different techniques, we need to ensure that their results are correct. This is accomplished by demonstrating that the algorithm is sound and complete with respect to the type system. The soundness property guarantees that if the algorithm gives as result the type $\mathtt{t}$ for the term $\mathtt{M}$, then the judgement $\Gamma \vdash \mathtt{M} : \mathtt{t}$ is valid. The completeness property, instead, guarantees that if $\Gamma \vdash \mathtt{M} : \mathtt{t}$ is valid, then the algorithm is able to determine that $\mathtt{t}$ is the type of $\mathtt{M}$.

1.2.2 Type and Effect Systems

Type and effect systems are a powerful extension of type systems which allows one to express general semantic properties and to statically reason about program execution. The underlying idea is to refine the type information so as to further express intentional or extensional properties of the semantics of the program. In practice, they compute the type of each program sentence and an approximate (but sound) description of its runtime behaviour.

The elements of a type system are extended as follows:

- in type syntax we annotate types with effects or tags describing some semantic properties. For example,

$$\begin{aligned} &\mathtt{t} ::= \mathtt{integer} \mid \mathtt{boolean} \mid \mathtt{t}_1 \xrightarrow{\phi} \mathtt{t}_2 \\ &\phi ::= \ldots \end{aligned}$$

  means that the functional types are annotated with the effect that will hold after the function application (latent effect);

- judgements not only have to assert correspondences between program sentences and types but also express properties about runtime behaviour. For example,

$$\Gamma \vdash \mathtt{M} : \mathtt{t} \rhd \phi$$

  means that in the environment Γ, $\mathtt{M}$ has type $\mathtt{t}$ and that the effect described by ϕ holds. In addition, we need to define further judgements and inference rules describing the relationship between effects;

- the correctness of a type and effect system is proved in two steps:
  1. prove that the type and effects system is a conservative extension of the underlying type system;
  2. prove a soundness theorem that consider the effects.

1.2.3 Flow Logic

Flow Logic [NN02] is a declarative approach to static analysis based on logical systems. It borrows and integrates many ideas from classical techniques, in particular from data flow analysis, constraint-based analysis, abstract interpretation and type systems. The distinctive feature of Flow Logic is to separate the *specification* of the analysis from its actual *computation*. Intuitively, the specification describes when the results, namely analysis *estimates*, are *acceptable*, i.e. describing the property which we are concerned with. Furthermore, Flow Logic provides us with a methodology to define a correct analysis algorithm which operates in polynomial time, by reducing the specification to a constraint satisfaction problem.

Usually, an analysis specification requires the definition of the following elements:

1. an abstract domain, i.e. the universe of discourse for the analysis estimate;
2. the format of the judgements;
3. the clauses.

The abstract domain is the space of properties of interest and it usually is a complete lattice. The clauses specify what the analysis computes. Formally, these clauses define a relationship that "assigns" to each program phrase its analysis. Flow Logic allows us to specify the analysis at different abstraction levels (different styles in the terminology of Flow Logic). In this thesis we will adopt a syntax-directed style (see [NN02] for a description of all possible styles and their relationships), so the analysis will be inductive defined by a set of inference rules. After the definition of the clauses, the Flow Logic methodology requires to prove that the analysis enjoys the properties below [3]:

1. the analysis is correct with respect to the dynamic semantics;
2. every expression has a best or most informative analysis estimate.

The statement of the correctness theorem and its proof depend on the style used to specify the semantics. In the case of small step operational semantics, which is the style of semantics we will use, the correctness result is a subject reduction theorem, expressing that the analysis estimate remains acceptable under reduction.

For (2) is sufficient to prove that the set of the acceptable analysis estimates enjoys a Moore family property: a Moore family is a subset V of a complete lattice such that whenever $Y \subseteq V$ then $\bigsqcap Y \in V$, for details see [DP02].

For the definition of an analysis algorithm we conveniently reformulate the analysis as a constraint satisfaction problem. To do that, it is necessary to define:

- the syntax and the (denotational) semantics of the constraints;
- a function that generates constraints for each program phrase;
- a relation saying when an analysis estimate satisfies a set of constraints.

Finally, we need to prove a syntactic soundness and completeness result expressing that any solution for the constraints is also an acceptable analysis estimate and vice versa.

[3] Actually, there are some cases (the abstract definition style) where it is also required to show that the analysis judgements are well-defined. See [NN02].

1.2.4 Model-checking

Model-checking is an automatic verification technique used to discover the presence of bugs in software and hardware. In particular, it consists of proving that a system complies with a given specification. This verification problem is equivalent to the problem in formal logic of checking that a given formula ϕ is satisfied in a finite domain M. Since the model M is finite it is possible to carry out an exhaustive search through all the possible models of the formula ϕ to decide its validity in M, i.e. $M \models \phi$. This exhaustive search is decidable in the finite case, and it corresponds to the model-checking procedure. Usually, M is a specification describing our system (the model), whereas the formula ϕ is obviously the property of interest. In classical model-checking[4] the specification M has the form of a Kripke structure and the formula ϕ is expressed in a modal temporal logic. This kind of logic allows us to express formulas about an universe where there are several possible states, and where a truth value is given to each proposition in each state. Traditionally, each state corresponds to time instants, and we say that it is possible to reach the state B from the state A, if B temporally follows A. In addition to classical truth operators, temporal formulas can include temporal operators saying in which states formulas should be true, e.g. in one future state (eventually) or in all future states (always).

In this thesis we will use a language-based approach to model-checking [VW86, VW08]. This approach exploits the fact that a model-checking problem can be reduced to a membership or inclusion problem in automata theory, since the set of the models of a temporal logic formula is a language (see [VW86, VW08] for details). In particular, we will compute an abstraction H for our programs and we use it as the model. The semantics of this abstraction is a context-free language, whereas our properties ϕ are regular languages. So in this case the model-checking reduces to verifying whether the intersection between the language of H and the complement of the language of ϕ (equivalent to $\neg\phi$) is empty, i.e. that the model H satisfies ϕ. Since from the classical results of automata theory [Hop79] we know that context-free languages are closed respect to the intersection with regular languages and the problem of emptiness is decidable for them, we have an effective model-checking procedure.

1.2.5 Language-based Security

Traditionally, *security* has been considered an external property of programs and has usually been tackled through mechanisms of operating systems, e.g. monitor, firewalls, etc. With the ubiquity of the Internet and of mobile computing devices it turns out that security is a fundamental concern, and that has to be taken into account from the first steps of the development process. *Language-based security* [Koz99, SMH01] has been proposed to address security concerns at programming language-level. Nowadays it is a wide research area, which can be summarized by the following two points:

1. the usage of techniques from compilers, from static analysis and from program transformations to enforce and verify security properties;

[4] Actually, there are different approaches as, e.g., statistical model-checking [LDB10] and probabilistic one [FKNP11].

2. the introduction into languages of constructs aimed at dealing with security issues.

Typically, the compiler creates a *certificate* containing extra information about the program, and packs it with the object code. The system, that wants to run this code (the consumer), uses the certificate to verify the compliance with some security properties by a verifier. If the certificate passes the test, the code is considered safe and it is run. This approach is the basis of the Java Bytecode verifier [LYBB13, Ros04] and Proof-Carrying Code [NL98]. In Chapter 4 we adopt the same schema to verify that our programs will adapt to every context arising at runtime.

An other interesting approach is the one proposed by Bartoletti et al. [BDFZ09, BDF09] and by Skalka et al. [SSVH08]. They introduced the notion of program history and the programming construct called security framing. A history is the sequence of actions that the program carries out at runtime. The notion of action is quite general, but they usually consider the ones that are relevant for security, e.g. the creation of a file, the access to a database record, etc.. The security framing is a construct that allows programmers to enforce a security policy ϕ on a program fragment e (in symbols $\phi[e]$). Intuitively, it works as a monitor: after each execution step the framing requires that the current history η satisfies ϕ (written $\eta \models \phi$). the security policy to be enforced is a regular property of the history. In [BDFZ09, BDF09, SSVH08] the authors consider regular properties of histories as security policies and show that the enforcement can be performed statically. Their idea consists of using a type and effect system to assign a type and an effect to each valid program. The effect is an expression in a suitable process algebra (*history expressions*) that soundly approximates all the histories that can be generated at runtime. Since history expressions are equivalent to context-free languages and since security policies are regular languages, they exploit the language-based model-checking to statically verify that a program is safe.

Note that this schema is an instance of the one described above, where the history expression works as the certificate and the model-checking procedure as the verifier. We adopt this approach in Chapter 3 to verify security and the compliance with the communication protocol of software components written in ContextML.

Chapter 2

A Basic Calculus for Context-Oriented Programming

In this chapter we give our first contribution to foundations of COP programming languages, in particular we investigate how the introduction of COP features affects the semantics of a "traditional" programming language. Here we introduce ContextML, a core functional language belonging the ML family extended by COP features. We considered a functional language because this class of languages has a clear and well understood semantics and because we can exploit the large number of formal techniques in the literature to statically prove properties about programs. At the same time, we equip ContextML with an annotated type system to ensure that the dispatch mechanism always succeeds at runtime for each well-typed expression.

The main contributions of this chapter are:

- the introduction of the notion of behavioural variation in a functional language as expressions (*layered expressions*)
- layers are values
- an annotated type system ensuring that the dispatching mechanism always succeeds for well-typed programs (Theorem 2.2.3).

First we present the features of ContextML through a running example and then we define its syntax, its dynamic semantics and its type system.

2.1 An example of ContextML features

To illustrate the novel features of ContextML and its design we exploit a running example. Consider a program implementing a game for mobile devices, like smartphones. The game can activate different effects depending on the battery charge level. For example, if the battery is fully charged, vibration is triggered when an enemy hits the player and sounds are produced. We assume that the level of charge is represented as discrete values, called profiles. For the sake of simplicity, we consider only two profiles, the *power saving mode* and the *performance one*. As usual in COP languages this kind of information concerns the context and can be represented in the code through two different layers: `PowerSavingMode` and `PerformanceMode`. Layers are expressible values, hence they can be returned by functions as results. The function `getBatteryProfile`, described below in a sugared syntax, queries the battery sensor to retrieve the charge level through the function `batSensor`. The returned layer describes the current active profile depending on a `threshold` value:

```
fun getBatteryProfile () =
  if batSensor () > threshold then
    PerformanceMode
  else
    PowerSavingMode
```

We represent the context as a stack of layers and we change the current context by pushing (activating) a layer. As an activation mechanism we adopt the *dynamic scoped activation* solution exploited by the most COP languages (see Section 1.1.3). This mechanism is implemented by the **with**(e_1) **in** e_2 construct. It activates the layer obtained evaluating e_1 and delimits the activation in the scope of the inner expression e_2. When the scope expires the activated layer is popped from the context, restoring the previous configuration. For instance, in the code below, the layer obtained as result of the call `getBatteryProfile`() is active throughout the execution of the inner expression.

```
with(getBatteryProfile()) in
  PowerSavingMode. basicEffects ()
  PerformanceMode. fullEffects ()
```

The above inner expression is a *layered expression* specifying a behavioural variation. Roughly, a layered expression is similar to pattern-matching but with layers in place of patterns and with the context as implicit parameter. Intuitively, a layered expression alters the control-flow of the application depending on the context. Its execution triggers a *dispatching mechanism* that inspects the context top-down to find the first layer matching one of the layered expression. The execution resumes from the expression of the matched layer. If the application `getBatteryProfile` () returns the layer `PerformanceMode`, we activate it through the **with** and the evaluation continues with the expression `fullEffects` () (the dispatching mechanism selects the second case since the top of the context is `PerformanceMode`).

Note that the dispatching mechanism fails when it cannot find a match. If the function `getBatteryProfile` returns an unexpected layer, e.g. `OnDemandMode`, then the program throws a runtime error being unable to adapt to the context. Later on, we show how to tackle these undesired behaviour by adopting static analysis techniques. The ContextML type system guarantees that well-typed programs are always capable to react to their changing environment, i.e. the dispatching always succeeds at runtime.

2.2 ContextML: a COP core of ML

ContextML is a purely functional fragment of ML extended with COP primitives. In ContextML the context is explicit and is part of the runtime environment. The language is endowed with primitives to manipulate the context and to specify behavioural variations in the form of layered expressions. The abstract syntax, the structural operational semantics and the type system of ContextML follow.

2.2.1 Abstract Syntax and Dynamic Semantics

Let *Const* be a set of constants (e.g. $()$, *true*, *false*), Ide a set of identifiers, LayerNames a set of layer names, then the syntax of ContextML is defined by the following grammar:

$$c \in Const \qquad x, f \in \mathsf{Ide} \qquad L \in \mathsf{LayerNames}$$

$$\begin{array}{lll l}
v ::= & & \textit{values} & \\
 & \mid c & \textit{constants} & \text{e.g. } 1, 2, \ldots, \texttt{true}, \texttt{false}, () \\
 & \mid \underline{L} & \underline{\textit{layers}} & \\
 & \mid \textbf{fun } f\ x\ \Rightarrow e & \textit{functions} & \\
 & & & \\
e ::= & & \textit{expressions} & \\
 & \mid v & \textit{values} & \\
 & \mid x & \textit{identifiers} & \\
 & \mid e_1\ e_2 & \textit{function application} & \\
 & \mid \textbf{let } x\ =\ e_1 \textbf{ in }\ e_2 & \textit{declaration} & \\
 & \mid e_1 \textbf{ op } e_2 & \textit{operators} & \text{e.g. } +, -, \times, \ldots \\
 & \mid \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else }\ e_2 & \textit{conditional} & \\
 & \mid \underline{\textbf{with}(e_1)\ \textit{in}\ e_2} & \underline{\textit{layers activation}} & \\
 & \mid \underline{\textbf{without}(e_1)\ \textit{in}\ e_2} & \underline{\textit{layers deactivation}} & \\
 & \mid \underline{lexp} & \underline{\textit{layered expressions}} & \\
 & & & \\
\underline{lexp} & ::= \underline{L.e} \mid \underline{L.e, lexp} & &
\end{array}$$

The novelties with respect to ML (underlined and in blue) are layers as expressible values; the **with** construct for activating layers; the **without** construct for deactivating layers; and the layered expressions ($lexp$). Recall that a context C is a stack of active layers. We denote with $L :: C$ the pushing of layer L on C and with $[L_1, \ldots, L_n]$ a context with n elements whose top is L_1. Formally,

Definition 2.2.1 (Context Extension)**.** The empty context is denoted by $[\,]$ and a context with n elements with L_1 at the top, by $[L_1, \ldots, L_n]$. Let $C = [L_1, \ldots, L_{i-1}, L_i, L_{i+1}, \ldots, L_n]$, $1 \leq i \leq n$ then

$$L :: C = \begin{cases} [L_i, L_1, \ldots, L_{i-1}, L_{i+1}, \ldots L_n] & \text{if } L = L_i \\ [L, L_1, \ldots, L_{i-1}, L_i, L_{i+1}, \ldots, L_n] & \text{otherwise} \end{cases}$$

Furthermore, we introduce the operation $C - L$ that removes a layer L from the context C if present, formally:

Definition 2.2.2 (Removing from Context)**.** Let $C = [L_1, \ldots, L_{i-1}, L_i, L_{i+1}, \ldots, L_n]$ be a context with n elements, then

$$C - L = \begin{cases} [L_1, \ldots, L_{i-1}, L_{i+1}, \ldots L_n] & \text{if } L = L_i \\ C & \text{otherwise} \end{cases}$$

The semantics is only defined for closed expressions and is characterised by judgements having the form $C \vdash e \rightarrow e'$ meaning that in context C the closed expression e reduces to e' in one evaluation step. In Figure 2.1 we show the semantic rules. Since most of constructs are inherited from ML, their semantics is standard. Hence, we comment only on rules $\textsc{with}_1$, $\textsc{with}_2$, $\textsc{with}_3$, $\textsc{without}_1$, $\textsc{without}_2$, $\textsc{without}_3$ $\textsc{lexp}$ that deal with the new constructs.

The rules for $\textbf{with}(e_1)$ **in** e_2 first evaluate e_1 in order to obtain a layer L; then, the expression e_2 is evaluated in a context extended (through the function ::) with L. Symmetrically, the rules for $\textbf{without}(e_1)$ **in** e_2 evaluate e_2 in a context where the layer L obtained evaluating e_1 is deactivated.

Example 2.2.1. Back to our running example, we consider to evaluate the second snippet of code above in a context containing the layer `AtHome` only and under the assumption that the function `batSensor` returns a value greater than the `threshold`. The following reduction shows how the layer activation mechanism works:

$$\begin{array}{lll}
[\texttt{AtHome}] \vdash & \textbf{with}(\texttt{getBatteryProfile ()})\ \textbf{in} & \rightarrow \\
& \quad \texttt{PowerSavingMode. basicEffects ()} & \\
& \quad \texttt{PerformanceMode. fullEffects ()} & \\
& & \\
& \textbf{with} \left(\begin{array}{l} \textbf{if}\ \texttt{batSensor ()} > \texttt{threshold}\ \textbf{then} \\ \quad \texttt{PerformanceMode} \\ \textbf{else} \\ \quad \texttt{PowerSavingMode} \end{array} \right) \textbf{in} & \rightarrow^{\star} \\
& \quad \texttt{PowerSavingMode. basicEffects ()} & \\
& \quad \texttt{PerformanceMode. fullEffects ()} & \\
& & \\
& \textbf{with}(\texttt{PerformanceMode})\ \textbf{in} & \rightarrow \quad \text{see Example 2.2.2} \\
& \quad \texttt{PowerSavingMode. basicEffects ()} & \\
& \quad \texttt{PerformanceMode. fullEffects ()} &
\end{array}$$

The expression inside the **with** is executed in the context [`PerformanceMode, AtHome`] and we will show its reduction in the next example.

When a layered expression $L_1.e_1, \ldots, L_n.e_n$ has to be evaluated (rule $\textsc{lexp}$), the *dispatch mechanism* is triggered. It inspects top-down the current context to select the expression e_i that corresponds to the first layer L_i which matches the context. Formally,

$$\textsc{if}_1\quad \frac{C \vdash e_0 \to e_0'}{C \vdash \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \to \textbf{if } e_0' \textbf{ then } e_1 \textbf{ else } e_2} \qquad \textsc{if}2\quad \frac{}{C \vdash \textbf{if } \texttt{true} \textbf{ then } e_1 \textbf{ else } e_2 \to e_1}$$

$$\textsc{if}3\quad \frac{}{C \vdash \textbf{if } \texttt{false} \textbf{ then } e_1 \textbf{ else } e_2 \to e_2} \qquad \textsc{let}_1\quad \frac{C \vdash e_1 \to e_1'}{C \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \to \textbf{let } x = e_1' \textbf{ in } e_2}$$

$$\textsc{let}_2\quad \frac{}{C \vdash \textbf{let } x = v \textbf{ in } e_2 \to e_2\,\{v/x\}} \qquad \textsc{op}_1\quad \frac{C \vdash e_1 \to e_1'}{C \vdash e_1 \textbf{ op } e_2 \to e_1' \textbf{ op } e_2} \qquad \textsc{op}_2\quad \frac{C \vdash e_2 \to e_2'}{C \vdash v \textbf{ op } e_2 \to v \textbf{ op } e_2'}$$

$$\textsc{op}3\quad \frac{v = v_1 \texttt{ op } v_2}{C \vdash v_1 \textbf{ op } v_2 \to v} \qquad \textsc{app}_1\quad \frac{C \vdash e_2 \to e_2'}{C \vdash e_1\ e_2 \to e_1\ e_2'} \qquad \textsc{app}_2\quad \frac{C \vdash e_1 \to e_1'}{C \vdash e_1\ v \to e_1'\ v}$$

$$\textsc{app}3\quad \frac{}{C \vdash (\textbf{fun } f\ x \Rightarrow e)v \to e\,\{\textbf{fun } f\ x \Rightarrow e/f, v/x\}} \qquad \textsc{with}_1\quad \frac{C \vdash e_1 \to e_1'}{C \vdash \textbf{with}(e_1) \textbf{ in } e_2 \to \textbf{with}(e_1') \textbf{ in } e_1}$$

$$\textsc{with}_2\quad \frac{L :: C \vdash e_2 \to e_2'}{C \vdash \textbf{with}(L) \textbf{ in } e_2 \to \textbf{with}(L) \textbf{ in } e_2'} \qquad \textsc{with}3\quad \frac{}{C \vdash \textbf{with}(L) \textbf{ in } v \to v}$$

$$\textsc{without}_1\quad \frac{C \vdash e_1 \to e_1'}{C \vdash \textbf{without}(e_1) \textbf{ in } e_2 \to \textbf{without}(e_1') \textbf{ in } e_2} \qquad \textsc{without}_2\quad \frac{C - L \vdash e \to e'}{C \vdash \textbf{without}(L) \textbf{ in } e \to \textbf{without}(L) \textbf{ in } e'}$$

$$\textsc{without}_4\quad \frac{}{C \vdash \eta, \textbf{without}(L) \textbf{ in } v \to v} \qquad \textsc{lexp}\quad \frac{dsp(C, \{L_1, \ldots, L_n\}) = L_i}{C \vdash L_1.e_1, \ldots, L_n.e_n \to e_i}$$

Figure 2.1: ContextML semantics.

Definition 2.2.3 (Dispatching mechanism)**.** The dispatch mechanism is implemented by the following partial function *dsp*, defined as

$$dsp([L_0, L_1, \ldots, L_n], A) = \begin{cases} L_0 & \text{if } L_0 \in A \\ dsp([L_1, \ldots, L_n], A) & \text{otherwise} \end{cases}$$

that returns the first layer in the context $[L_0, L_1, \ldots, L_m]$ which matches one of the layers in the set A. If no layer matches then the computation gets stuck.

Example 2.2.2. We continue the previous examples by showing the reduction of the layered expression inside the **with**. We assume that the functions `basicEffects` and `fullEffects` return the numbers of enabled effects.

$$\begin{aligned} C = [\texttt{PerformanceMode, AtHome}] \vdash \begin{matrix}\texttt{PowerSavingMode. basicEffects ()} \\ \texttt{PerformanceMode. fullEffects ()}\end{matrix} \quad &\rightarrow \\ \texttt{fullEffects ()} \rightarrow^{\star}& \\ 10& \end{aligned}$$

In the computation from the first to second step we trigger the dispatching mechanism that matches the top of the context with the layer `PerformanceMode`, i.e.

$$dsp(C, \{\texttt{PerformanceMode, PowerSavingMode}\}) = \texttt{PerformanceMode}.$$

As a further example, consider to evaluate the above layered expression in context $C' = [\texttt{OnDemandMode, AtHome}]$. In this case, the computation gets stuck

$$C' \vdash \begin{matrix}\texttt{PowerSavingMode. basicEffects ()} \\ \texttt{PerformanceMode. fullEffects ()}\end{matrix} \quad \not\rightarrow$$

since the invocation $dsp(C', \{\texttt{PerformanceMode, PowerSavingMode}\})$ fails. Our type system detects and avoids exactly this kind of error.

2.2.2 Type system

We introduce an annotated monomorphic type system for ContextML which ensures that the dispatch mechanism always succeeds at runtime for well-typed expressions. To do that, our type system over-approximates the possible contexts which may arise at runtime from the initial one by tracking all layers which may be activated. Furthermore, it verifies that all contexts, in which a layered expression (call it *lexp*) can be evaluated, contain at least one layer occurring in *lexp*.

Our type system is characterised by typing judgements of the form $\langle \Gamma; C \rangle \vdash e : \tau$. This means that "in the type environment Γ and in the context C, the expression e has type τ".

The type environment Γ binds the variables of an expression to their types. As usual, it is inductively defined as

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

where $\emptyset$ denotes the empty environment and $\Gamma, x : \tau$ denotes the environment Γ extended by the binding for the variable x (x does not occur in Γ). We denote with Γ_x the environment Γ where the binding for x is removed and with $dom(\Gamma)$ the set of variables for which Γ contains a binding. Moreover, we use the application notation $\Gamma(x)$ for retrieving the type τ bound to the variable x.

Types are constants, layers and functions.

$$\tau, \tau_1, \tau' ::= \tau_c \mid ly_\phi \mid \tau_1 \xrightarrow{\psi} \tau_2 \qquad \tau_c \in \{int,\ bool,\ unit,\ \dots\} \quad \phi, \psi \in \wp(\mathsf{LayerNames})$$

We annotate types with sets of layer names ϕ, ψ for analysis reasons. In ly_ϕ, ϕ over-approximates the layers an expression at runtime can be reduced to. In $\tau_1 \xrightarrow{\psi} \tau_2$, ψ over-approximates the layers that must be active in the context to apply the function (precondition of the function).

Example 2.2.3. Back to our example, the type of the function `getBatteryProfile` will be the following:

$$\texttt{unit} \xrightarrow{\emptyset} ly_{\{\texttt{PowerSavingMode},\,\texttt{PerformanceMode}\}}.$$

The intuition is that the function application yields a layer in the set

$$\{\texttt{PowerSavingMode}, \texttt{PerformanceMode}\}.$$

The function has no preconditions, i.e. it can be applied in any context.

Our typing rules are in Figure 2.2. Since types are annotated, the type system contains rules for dealing with the subtyping and the subeffecting. The rules SCONST, SLY, SFUN have judgements of the form $\tau_1 \leq \tau_2$, meaning that τ_1 is a subtype of τ_2. Furthermore, we assume that annotations are ordered by set-inclusion and that $|C|$ denotes the set of active layers in a context C, i.e. $|[L_1, \dots, L_n]| = \{L_1, \dots, L_n\}$.

The rule SCONST states that a constant type is a subtype of itself. By the rule SLY a layer type ly_ϕ is a subtype $ly_{\phi'}$ if and only if the annotation ϕ is a subset of ϕ'. The rule SFUN is a subtyping rule for functional types. As usual $\tau_1 \xrightarrow{\psi} \tau_2$ is *contravariant* in τ_1 but covariant in ϕ and τ_2.

By the rule TCONST we assign to a constant c the corresponding type τ_c. The rule TLY asserts that the type of a layer L is ly annotated with the singleton set $\{L\}$. By the rule TVAR we look up the type of an identifier x from the environment Γ. The rule TSUB allows us to enlarge the annotation of a type, by applying subtyping rules. In rule TFUN we guess a type for the bound variable, for the function f and we determine the type of the body under these additional assumptions and in a guessed context C'. Implicitly, we require that the guess of a type for f matches the one of the resulting function. Additionally, we require that the resulting type is annotated with a precondition that includes the layers in C'. The rule TWITH establishes that an expression **with** has type τ, provided that the type of e_1 is ly_ϕ (recall that ϕ is a set of layers) and that e_2 has type τ in all context obtained by extending the context C with the layers in ϕ. Symmetrically, the rule TWITHOUT establishes that an expression **without** has type τ, as long as the type of e_1 is ly_ϕ and that e_2 has type τ in all context obtained by removing from C the layers

$$\textsc{Sconst}\quad \frac{}{\tau_c \leq \tau_c} \qquad \textsc{Sly}\quad \frac{\phi \subseteq \phi'}{ly_{\phi} \leq ly_{\phi'}} \qquad \textsc{Sfun}\quad \frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2' \qquad \psi \subseteq \psi'}{\tau_1 \xrightarrow{\psi} \tau_2 \leq \tau_1' \xrightarrow{\psi'} \tau_2'} \qquad \textsc{Tconst}\quad \frac{}{\langle \Gamma; C \rangle \vdash c : \tau_c}$$

$$\textsc{Tly}\quad \frac{}{\langle \Gamma; C \rangle \vdash L : ly_{\{L\}}} \qquad \textsc{Tsub}\quad \frac{\langle \Gamma; C \rangle \vdash e : \tau' \qquad \tau' \leq \tau}{\langle \Gamma; C \rangle \vdash e : \tau} \qquad \textsc{TVar}\quad \frac{\Gamma(x) = \tau \qquad \text{if } x \in dom(\Gamma)}{\langle \Gamma; C \rangle \vdash x : \tau}$$

$$\textsc{Tfun}\quad \frac{\langle \Gamma_{x,f}, x : \tau_1, f : \tau_1 \xrightarrow{|C'|} \tau_2; C' \rangle \vdash e : \tau_2}{\langle \Gamma; C \rangle \vdash \textbf{fun}\ f\ x \Rightarrow e : \tau_1 \xrightarrow{|C'|} \tau_2} \qquad \textsc{Twith}\quad \frac{\langle \Gamma; C \rangle \vdash e_1 : ly_{\phi} \qquad \forall L' \in \phi. \langle \Gamma; L' :: C \rangle \vdash e_2 : \tau}{\langle \Gamma; C \rangle \vdash \textbf{with}(e_1)\ \textbf{in}\ e_2 : \tau}$$

$$\textsc{Twithout}\quad \frac{\langle \Gamma; C \rangle \vdash e_1 : ly_{\phi} \qquad \forall L' \in \phi. \langle \Gamma; L' - C \rangle \vdash e_2 : \tau}{\langle \Gamma; C \rangle \vdash \textbf{without}(e_1)\ \textbf{in}\ e_2 : \tau}$$

$$\textsc{Tlexp}\quad \frac{\forall i. \langle \Gamma; C \rangle \vdash e_i : \tau \qquad L_1 \in |C| \vee \cdots \vee L_n \in |C|}{\langle \Gamma; C \rangle \vdash L_1.e_1, \ldots, L_n.e_n : \tau}$$

$$\textsc{Tapp}\quad \frac{\langle \Gamma; C \rangle \vdash e_1 : \tau_1 \xrightarrow{\phi} \tau_2 \qquad \langle \Gamma; C \rangle \vdash e_2 : \tau_1 \qquad \phi \subseteq |C|}{\langle \Gamma; C \rangle \vdash e_1\ e_2 : \tau_2}$$

$$\textsc{Top}\quad \frac{\langle \Gamma; C \rangle \vdash e_1 : \tau_c \qquad \langle \Gamma; C \rangle \vdash e_2 : \tau_c}{\langle \Gamma; C \rangle \vdash e_1\ \textbf{op}\ e_2 : \tau_c} \qquad \textsc{Tlet}\quad \frac{\langle \Gamma; C \rangle \vdash e_1 : \tau_1 \qquad \langle \Gamma_x, x : \tau_1, C \rangle \vdash e_2 : \tau_2}{\langle \Gamma; C \rangle \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}$$

$$\textsc{Tif}\quad \frac{\langle \Gamma; C \rangle \vdash e_0 : bool \qquad \langle \Gamma; C \rangle \vdash e_1 : \tau \qquad \langle \Gamma; C \rangle \vdash e_2 : \tau}{\langle \Gamma; C \rangle \vdash \textbf{if}\ \ e_0\ \ \textbf{then}\ e_1\ \textbf{else}\ e_2 : \tau}$$

Figure 2.2: ContextML type system

$$\textsc{Twith}\ \dfrac{\textsc{Tlexp}\ \dfrac{\langle\Gamma;C'\rangle\vdash\texttt{fullEffects}\,():\tau \qquad \begin{array}{c}\langle\Gamma;C'\rangle\vdash\texttt{basicEffects}\,():\tau\\ \texttt{PowerSavingMode}\in|C'|\vee\texttt{PerformarceMode}\in|C'|\end{array}}{\langle\Gamma;C'\rangle\vdash \begin{array}{l}\texttt{PowerSavingMode.}\,\texttt{basicEffects}\,()\\ \texttt{PerformarceMode.}\,\texttt{fullEffects}\,()\end{array}:\tau} \qquad \begin{array}{c}\\ \langle\Gamma;C\rangle\vdash\texttt{getBatteryProfile}\,():ly_\phi\end{array} \qquad \dfrac{\cdots}{\langle\Gamma;C''\rangle\vdash\ldots}\ \textsc{Tlexp}}{\langle\Gamma;C\rangle\vdash\textbf{with}(\texttt{getBatteryProfile}\,())\ \textbf{in}\ \begin{array}{l}\texttt{PowerSavingMode.}\,\texttt{basicEffects}\,()\\ \texttt{PerformarceMode.}\,\texttt{fullEffects}\,()\end{array}:\tau}$$

Figure 2.3: The typing derivation of the running example.

in ϕ. By the Tlexp rule the type of a layered expression is τ, provided that each sub-expression e_i has type τ and that at least one among the layers $L_1, \ldots L_n$ is active in the context C. This requirement ensures that the dispatch mechanism always succeeds at runtime. Notably, when evaluating a layered expression one of the mentioned layers will be active in the current context.

Example 2.2.4. Back to our example, the considered expression is well-typed as witnessed in Figure 2.3. We assume `basicEffects,fullEffects` : `unit` $\xrightarrow{\emptyset}$ τ. We denote $\phi = \{\texttt{PowerSavingMode},\texttt{PerformanceMode}\}$; $C' = \texttt{PowerSavingMode} :: C$; $C'' = \texttt{PerformanceMode} :: C$. The type of `getBatteryProfile` ensures that one between the two layers `PowerSavingMode` and `PerformanceMode` is returned. One of them is required to be active in order to evaluate the layered expression. Hence, the Twith rule (the last used in the figure) guarantees that the whole expression is never stuck at runtime.

The rule Tapp is almost standard and reveals the role of function preconditions. The application gets a type if only if the layers in the precondition ϕ are active in the current context C.

Example 2.2.5. To better explain how preconditions work, consider the Figure 2.4. There the function **fun** $f\ x \Rightarrow L_1.0$ is shown having type $int \xrightarrow{\{L_1\}} int$. This means that L_1 must be active in the context where we apply the function.

The rule Top is standard: it states that if the sub-expressions e_1 and e_2 are constants, i.e. they have type τ_c, then the result of the operator **op** has type τ_c too. The rule Tlet requires that the expression e_2 gets a type τ_2 in an environment Γ, which was extended by the binding between the variable x and the type τ_1 computed for the expression e_1. The type of the overall **let** expression is the one computed for e_2. By the Tif rule a conditional expression gets the type of the branches if the guard (the expression e_1) is a boolean and the two branches have the same type τ.

Our type system guarantees not only that functional types are correctly used, but also that the evaluation of a layered expression never gets stuck. The following lemmata prove that our type system is sound with respect to the operational semantics. The first one ensures that types are preserved during reduction:

Lemma 2.2.1 (Preservation). *Let e_s be a closed expression, if $\langle\Gamma;C\rangle \vdash e_s : \tau$ and $C \vdash e_s \to e'_s$ then $\langle\Gamma;C\rangle \vdash e'_s : \tau$.*

$$\textsc{Tlet}\;\dfrac{\textsc{Tfun}\;\dfrac{\textsc{Tlexp}\;\dfrac{\langle \Gamma, x:\tau, f:\tau \xrightarrow{|C'|} \tau; C'\rangle \vdash 0:\tau \qquad L_1 \in C'}{\langle \Gamma, x:\tau, f:\tau \xrightarrow{|C'|} \tau; C'\rangle \vdash L_1.0:\tau}}{\langle \Gamma; C\rangle \vdash \textbf{fun}\, f\; x \Rightarrow L_1.0 : \tau \xrightarrow{|C'|} \tau} \qquad \dfrac{\begin{array}{c}\langle \Gamma, g:\tau \xrightarrow{|C'|} \tau; C\rangle \vdash g:\tau \xrightarrow{|C'|} \tau \\ \langle \Gamma, g:\tau \xrightarrow{|C'|} \tau; C\rangle \vdash 3:\tau \qquad |C'| \subseteq |C|\end{array}}{\langle \Gamma, g:\tau \xrightarrow{|C'|} \tau; C\rangle \vdash g\; 3:\tau}\;\textsc{Tapp}}{\langle \Gamma; C\rangle \vdash \textbf{let}\; g \;=\; \textbf{fun}\, f\; x \Rightarrow L_1.0 \;\textbf{in}\; g\; 3:\tau}$$

Figure 2.4: Derivation of a function with precondition. We assume that $C' = [L_1]$, L_1 is active in C and, for typesetting convenience, we also denote $\tau = int$.

The second one states that a well-typed expression is never stuck unless it is a value:

Lemma 2.2.2 (Progress). *Let e_s be a closed expression such that $\langle \Gamma; C\rangle \vdash e_s : \tau$ for some Γ and C. If $C \vdash e_s \not\rightarrow$, i.e. e_s is stuck, then e_s is a value.*

The following theorem ensures us that the dispatching mechanism always succeeds at runtime for a well-typed expression:

Theorem 2.2.3. *Let e_s be a closed expression if $\langle \emptyset; C\rangle \vdash e_s : \tau$ for some C, either the computation terminates by yielding a value ($C \vdash e_s \rightarrow^* v$) or it diverges, but it* ***never*** *gets stuck.*

2.3 Soundness of the Type System

This section contains the technical development to prove Lemma 2.2.1, Lemma 2.2.2 and Theorem 2.2.3. The proofs require an auxiliary definition and some lemmata which are stated below.

The following Definition formalizes the notion of substitution used in the dynamic semantic of ContextML.

Definition 2.3.1 (Capture avoiding substitutions)**.** Given the expressions e, e' and the variable x we define $e\{e'/x\}$ as following

$$\begin{array}{l}
c\{e'/x\} = c \\
L\{e'/x\} = L \\
(\textbf{fun}\; f\; x' \Rightarrow e)\{e'/x\} = \textbf{fun}\; f\; x' \Rightarrow e\{e'/x\} \\
\qquad\qquad\qquad\qquad \text{if } f \neq x \wedge x' \neq x \wedge f, x' \notin FV(e') \\
x\{e'/x\} = e' \\
x'\{e'/x\} = x' \qquad\qquad \text{if } x \neq x' \\
(e_1\; e_2)\{e'/x\} = e_1\{e'/x\}\; e_2\{e'/x\} \\
(e_1\; \textbf{op}\; e_2)\{e'/x\} = e_1\{e'/x\}\; \textbf{op}\; e_2\{e'/x\} \\
(\textbf{if}\; e_1\; \textbf{ then}\; e_2\; \textbf{else}\; e_3)\{e'/x\} = \\
\qquad\qquad \textbf{if}\; e_1\{e'/x\}\; \textbf{then}\; e_2\{e'/x\}\; \textbf{else}\; e_3\{e'/x\} \\
(\textbf{let}\; x' \;=\; e_1\; \textbf{in}\; e_2)\{e'/x\} = \textbf{let}\; x' \;=\; e_1\{e'/x\}\; \textbf{in}\; e_2\{e'/x\} \\
\qquad\qquad\qquad\qquad \text{if } x \neq x' \wedge x' \in FV(e')
\end{array}$$

$$(\mathbf{with}(e_1)\ \mathbf{in}\ e_2)\{e'/x\} = \mathbf{with}(e_1\{e'/x\})\ \mathbf{in}\ e_2\{e'/x\}$$
$$(\mathbf{without}(e_1)\ \mathbf{in}\ e_2)\{e'/x\} = \mathbf{without}(e_1\{e'/x\})\ \mathbf{in}\ e_2\{e'/x\}$$
$$(L_1.e_1, \dots, L_n.e_n)\{e'/x\} = L_1.e_1\{e'/x\}, \dots, L_n.e_n\{e'/x\}$$

The lemma below ensures that the order of bindings in a type environment does not affect the typing of an expression (useful to prove Lemma 2.3.3).

Lemma 2.3.1 (Permutation). *If* $\langle\Gamma; C\rangle \vdash e : \tau$ *and* Γ' *is a permutation of* Γ *then* $\langle\Gamma'; C\rangle \vdash e : \tau$.

Proof (Sketch). The proof is a straightforward induction on typing derivation and by cases on the last rule applied. We consider only few cases, the other ones follow the same schema.

- case Tvar
  By the premise of the rule we have $\Gamma(x) = \tau$. Since Γ' is a permutation of Γ, they are the same bindings, thus it holds also $\Gamma'(x) = \tau$ and the thesis follows from applying the rule Tvar.

- case Tlexp
  By the premise of the rule we know $\forall\ i \in \{1, \dots, n\}\ \langle\Gamma; C\rangle \vdash e_i : \tau$ and $L_i \in |C|$. By the induction hypothesis we have $\forall\ i \in \{1, \dots, n\}\ \langle\Gamma'; C\rangle \vdash e_i : \tau$. Hence, the thesis follows from applying the Tlexp.

- case Tlet
  By the premise of the rule we have $\langle\Gamma; C\rangle \vdash e_1 : \tau_1$ and $\langle\Gamma_x, x : \tau_1; C\rangle \vdash e_2 : \tau_2$. Now consider $\Gamma'' = \Gamma_x, x : \tau_1$, there are two cases:

  1. $x \notin dom(\Gamma)$ then $\Gamma_x = \Gamma$ and $\Gamma'' = \Gamma, x : \tau_1$. Since Γ' contains the same bindings of Γ it holds that $\Gamma'_x = \Gamma'$ and that $\Gamma', x : \tau_1$ is a permutation of Γ''.
  2. $x \in dom(\Gamma)$ then $\Gamma_x \neq \Gamma$ and $\Gamma'' = \Gamma_x, x : \tau_1$. Since Γ' is a permutation of Γ we have $x \notin dom(\Gamma')$ too and $\Gamma' \neq \Gamma'_x$; we removed the same binding from the two environment then Γ'_x is a permutation of Γ_x, thus $\Gamma'_x, x : \tau_1$ is a permutation of $\Gamma_x, x : \tau_1$.

  By using the induction hypothesis we have that $\langle\Gamma'; C\rangle \vdash e_1 : \tau_1$ and $\langle\Gamma'_x, x : \tau_1; C\rangle \vdash e_2 : \tau_2$. The thesis follows from applying the rule Tlet.

□

The lemma below allows us to insert a new binding in the type environment without changing the typing of an expression (exploited in the proof of Lemma 2.3.3).

Lemma 2.3.2 (Weakening). *If* $\langle\Gamma; C\rangle \vdash e : \tau$ *and* $y \notin dom(\Gamma)$ *then* $\langle\Gamma, y : \tau'; C\rangle \vdash e : \tau$.

Proof (Sketch). By induction on typing derivation and then by cases on the last rule applied. We consider only few cases, the other ones follow the same schema.

- case Tvar
  By the premise of the rule we know $\Gamma(x) = \tau$. Since it holds $y \notin dom(\Gamma)$, the adding of the new binding does not affect the other ones, hence, we have $\Gamma, y : \tau(x) = \tau$. Then, the thesis follows from using the rule Tvar.

- case Twith
  By the premise of rule it holds $\langle \Gamma; C \rangle \vdash e_1 : ly_\phi$ and $\forall L \in \phi$ $\langle \Gamma; L :: C \rangle \vdash e_2 : \tau$. By using the induction hypothesis we have $\langle \Gamma, y : \tau'; C \rangle \vdash e_1 : ly_\phi$ and $\forall L \in \phi$ $\langle \Gamma, y : \tau'; L :: C \rangle \vdash e_2 : \tau$. Then, the thesis follows from applying the rule Twith.

- case Tlet
  By the premise of the rule the judgements $\langle \Gamma; C \rangle \vdash e_1 : \tau_1$ and $\langle \Gamma, x : \tau_1; C \rangle \vdash e_2 : \tau_2$ hold. Since $y \notin (\Gamma)$ and $y \neq x$, then $y \notin (\Gamma, x : \tau_1)$. The thesis follows from the induction hypothesis and the conclusion of the rule Tlet.

□

The following lemma ensures that we can always enlarge the typing environment and the context where an expression type-checks (useful in the proof of the Lemma 2.3.4).

Lemma 2.3.3 (Inclusion)**.**

1. *If* $\langle \Gamma; C \rangle \vdash e : \tau$ *and* $\Gamma \subseteq \Gamma'$ *for some* Γ'*, then* $\langle \Gamma'; C \rangle \vdash e : \tau$*.*

2. *If* $\langle \Gamma; C \rangle \vdash e : \tau$ *and* $|C| \subseteq |C'|$ *for some* C'*, then* $\langle \Gamma; C' \rangle \vdash e : \tau$*.*

Proof.

1. Since $\Gamma \subseteq \Gamma'$ there exists some bindings $\{x_1 : \tau_1, \dots, x_n : \tau_n\}$ that are in Γ' but not in Γ. By adding these bindings to Γ we obtain $\Gamma'' = \Gamma, x_1 : \tau_1, \dots, x_n : \tau_n$. Then by using n times Lemma 2.3.2 we have $\langle \Gamma''; C \rangle \vdash e : \tau$. Since Γ'' is a permutation of Γ', the thesis follows from Lemma 2.3.1.

2. (Sketch) By induction on typing derivation and then by cases on the last rule applied. We consider only few cases, the other ones follow the same schema.

   - case Tlexp
     By the premise of the rule it holds $\forall i \in \{1, \dots, n\}$ $\langle \Gamma; C \rangle \vdash e_i : \tau$ and $\bigvee_i L_i \in |C|$. By induction hypothesis we have $\forall i \in \{1, \dots, n\}$ $\langle \Gamma; C' \rangle \vdash e_i : \tau$ and since $|C| \subseteq |C'|$ then $\bigvee_i L_i \in |C'|$ holds. The thesis follows from the rule Tlexp.

   - case Tapp
     By the premise of the rule we have $\langle \Gamma; C \rangle \vdash e_1 : \tau_1 \xrightarrow{\phi} \tau_2$, $\langle \Gamma; C \rangle \vdash e_2 : \tau_1$ and $\phi \subseteq |C|$. By induction hypothesis we know $\langle \Gamma; C' \rangle \vdash e_1 : \tau_1 \xrightarrow{\phi} \tau_2$, $\langle \Gamma; C' \rangle \vdash e_2 : \tau_1$. Since $\phi \subseteq |C| \subseteq |C'|$ we can apply the rule Tapp to prove the thesis.

   - case Twith
     By the premise of rule it holds $\langle \Gamma; C \rangle \vdash e_1 : ly_\phi$ and $\forall L \in \phi$ $\langle \Gamma; L :: C \rangle \vdash e_2 : \tau$. If $L \in |C|$ then $L \in |C'|$ and it holds $L :: C \subseteq L :: C'$; if $L \notin |C|$ and $L \in |C'|$ it

holds $L :: C \subseteq C'$; if $L \notin |C|$ and $L \notin |C'|$ it holds $L :: C \subseteq L :: C'$. Then, by the induction hypothesis we have $\langle \Gamma; C' \rangle \vdash e_1 : ly_\phi$ and $\forall L \in \phi$ $\langle \Gamma; L :: C' \rangle \vdash e_2 : \tau$. The thesis follows from the rule Twith.

□

The lemma below (useful to prove Lemma 2.2.1) ensures us that if a function type-checks in a context where its preconditions are satisfied, then its body will type-check too, of course, adding the needed bindings to the type environment.

Lemma 2.3.4 (Decomposition). *If* $\langle \Gamma; C \rangle \vdash \textbf{\textit{fun}}\, f\, x \Rightarrow e : \tau_1 \xrightarrow{\phi} \tau_2$ *and* $\phi \subseteq |C|$ *then* $\langle \Gamma_{x,f}, x : \tau_1, f : \tau_1 \xrightarrow{\phi} \tau_2; C \rangle \vdash e : \tau_2$.

Proof. By the premise of the rule Tfun it holds $\langle \Gamma_{x,f}, x : \tau_1, f : \tau_1 \xrightarrow{\phi} \tau_2; \phi \rangle \vdash e : \tau_2$. The thesis follows from using Lemma 2.3.3 (2). □

The following lemma guarantees that the types are preserved under substitution (it is used to prove Lemma 2.2.1).

Lemma 2.3.5 (Substitution). *If* $\langle \Gamma_y, y : \tau'; C \rangle \vdash e : \tau$ *and* $\langle \Gamma; C \rangle \vdash v : \tau'$ *then* $\langle \Gamma; C \rangle \vdash e\{v/y\} : \tau$.

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

- case Tconst
  By Definition 2.3.1 it holds $c\{v/y\} = c$. Since Tconst is an axiom we can trivially clams $\langle \Gamma; C \rangle \vdash c : \tau$, i.e. the thesis holds.

- case Tly
  It follows from a reasoning similar to the previous one.

- case Tvar
  By the premise of the rule we know $\Gamma(x) = \tau$. We have two cases:

  1. $x = y$ then $\langle \Gamma_x, x : \tau'; C \rangle \vdash x : \tau$ and $\tau = \tau'$. By Definition 2.3.1 we know $x\{v/x\} = v$, hence, the thesis trivially holds.
  2. $x \neq y$ then by Definition 2.3.1 we have $x\{v/y\} = x$, hence the thesis holds, since we removed from the environment the binding for y only, but not for x.

- case Tsub By the premise of the rule we know that it holds $\langle \Gamma_y, y : \tau'; C \rangle \vdash e : \tau''$ and $\tau'' \leq \tau$. By the induction hypothesis we know that $\langle \Gamma_y, y : \tau'; C \rangle \vdash e\{v/x\} : \tau''$ holds, hence, the thesis follows from applying the rule Tsub again.

- case Tfun
  Without loss of generality we assume that $x \neq y$. By the premise of the rule we have $\langle \Gamma_{x,f}, x : \tau_1, f : \tau_1 \xrightarrow{|C'|} \tau_2, y : \tau'; C' \rangle \vdash e : \tau_2$. By induction hypothesis it holds $\langle \Gamma_{x,f}, x : \tau_1, f : \tau_1 \xrightarrow{|C'|} \tau_2; C' \rangle \vdash e\{v/x\} : \tau_2$, and then $\langle \Gamma; C \rangle \vdash \textbf{fun}\ f\ x \Rightarrow e\{v/y\}$:

$\tau_1 \xrightarrow{|C'|} \tau_2$ by the rule Tfun. The thesis follows from the Definition 2.3.1 applied from right to left.

- case Twith
  By the premise of the rule we have $\langle\Gamma, y : \tau'; C\rangle \vdash e_1 : ly_\phi$ and $\forall L' \in \phi.\langle\Gamma, y : \tau'; L' :: C\rangle \vdash e_2 : \tau$. By using the induction hypothesis it holds $\langle\Gamma; C\rangle \vdash e_1\{v/y\} : ly_\phi$ and $\forall L' \in \phi.\langle\Gamma; L' :: C\rangle \vdash e_2\{v/y\} : \tau$. The thesis follows applying the rule Twith and then Definition 2.3.1 from right to left.

- case Twithout
  Similar to the case Twith.

- case Tlexp
  By the premise of the rule we know for all $i \in \{1, \ldots, n\}$ $\langle\Gamma, y : \tau'; C\rangle \vdash e_i : \tau$ and $\bigvee_i L_i \in |C|$. The induction hypothesis ensures us that for all $i \in \{1, \ldots, n\}$ $\langle\Gamma; C\rangle \vdash e_i\{v/x\} : \tau$, thus the thesis follows using the rule Tlexp and then Definition 2.3.1 from right to left.

- case Tapp
  By the premise of the rule it holds $\langle\Gamma, y : \tau'; C\rangle \vdash e_1 : \tau_1 \xrightarrow{\phi} \tau_2$, $\langle\Gamma, y : \tau'; C\rangle \vdash e_2 : \tau_1$ and $\phi \subseteq |C|$. By exploiting the induction hypothesis we have $\langle\Gamma; C\rangle \vdash e_1\{v/y\} : \tau_1 \xrightarrow{\phi} \tau_2$ and $\langle\Gamma; C\rangle \vdash e_2\{v/y\} : \tau_1$. The thesis follows using the rule Tapp and then Definition 2.3.1 from right to left.

- case Top
  It is similar to the case Tapp.

- case Tif
  It is similar to the case Tapp.

- case Tlet
  Without loss of generality assume that $x \neq y$. The proof is similar to the case Tapp.

□

The following lemma states that values type-check in every context (it is exploited in Lemma 2.2.1).

Lemma 2.3.6. *If* $\langle\Gamma; C\rangle \vdash v : \tau$ *then for all* C' $\langle\Gamma; C'\rangle \vdash v : \tau$.

Proof. We consider three cases depending on the form of v:

1. case $v = c$
   Straightforward because the typing rule Tconst is an axiom.

2. case $v = L$
   Straightforward because the typing rule Tly is an axiom.

3. case $v = \textbf{fun}\ f\ x \Rightarrow e$
   By the premise of the typing rule we know, so the context C'' does not depend on C and that the binding for x is added to the environment, so we can use the typing rule to conclude the thesis.

□

Lemma 2.2.1 (Preservation). *Let e_s be a closed expression, if $\langle \Gamma; C \rangle \vdash e_s : \tau$ and $C \vdash e_s \to e'_s$ then $\langle \Gamma; C \rangle \vdash e'_s : \tau$.*

Proof. By induction of the depth of the typing derivation and then, by cases on the last rule applied. Below for each case we report the premises of the corresponding rule.

- case Tconst or Tly or Tvar or Tfun
  In this case we know that e_s is a value (or a variable in the case of Tvar), then it cannot be the case $C \vdash e_s \to e'_s$ for any e'_s, so the theorem vacuously holds.

- case Tsub $\quad \langle \Gamma; C \rangle \vdash e_s : \tau' \quad \tau' \leq \tau$
  By using the induction hypothesis we also know $\langle \Gamma; C \rangle \vdash e'_s : \tau'$, hence, by applying the rule Tsub it holds $\langle \Gamma; C \rangle \vdash e'_s : \tau$, i.e. the thesis.

- case Twith $\quad e_s = \textbf{with}(e_1)\ \textbf{in}\ e_2 \quad \langle \Gamma; C \rangle \vdash e_1 : ly_\phi \quad \forall L' \in \phi\ \langle \Gamma; L' :: C \rangle \vdash e_2 : \tau$
  There are three semantic rules from which $C \vdash e_s \to e'_s$ can be derived.

  1. case with$_1$
     In this case we know $C \vdash e_1 \to e'_1$ by the premise of the semantic rule with$_1$, hence $e'_s = \textbf{with}(e'_1)\ \textbf{in}\ e_2$. By the induction hypothesis we have $\langle \Gamma; C \rangle \vdash e'_1 : ly_\phi$, thus using the rule Twith we conclude $\langle \Gamma; C \rangle \vdash e'_s : \tau$.
  2. case with$_2$
     In this case by the premise of the semantic rule with$_2$ we have that e_2 reduces to e'_2 ($C \vdash e_2 \to e'_2$), that $e_1 = L$ and that $e'_s = \textbf{with}(L)\ \textbf{in}\ e'_2$. By using the induction hypothesis we have $\forall L' \in \phi\ \langle \Gamma; L' :: C \rangle \vdash e'_2 : \tau$. Than the thesis follows from applying the typing rule Twith.
  3. case with$_3$
     In this case we have $e_s = \textbf{with}(L)\ \textbf{in}\ v$ and $e'_s = v$ and $\forall L' \in \phi\ \langle \Gamma; L' :: C \rangle \vdash v : \tau$ by our assumptions. The thesis follows from Lemma 2.3.6.

- case Twithout
  Similar to the case Twith.

- case Tlexp $\quad e_s = L_1.e_1, \ldots, L_n.e_n \quad \forall i \in \{1, \ldots, n\}\ \langle \Gamma; C \rangle \vdash e_i : \tau \quad \bigvee_i L_i \in |C|$
  By the semantic rule lexp e_s reduces ($C \vdash e_s \to e'_s$) to an subexpression of it, i.e. $e'_s = e_i$ for some $i \in \{1, \ldots, n\}$. The thesis vacuously follows from our assumptions.

- case Tapp $\quad e_s = e_1\ e_2 \quad \langle \Gamma; C \rangle \vdash e_1 : \tau_1 \xrightarrow{\phi} \tau_2 \quad \langle \Gamma; C \rangle \vdash e_2 : \tau_1 \quad \phi \subseteq |C|$
  There are three semantic rules from which $C \vdash e_s \to e'_s$ can be derived:

1. case APP1
   In this case we know $C \vdash e_2 \rightarrow e_2'$ from the premise of the semantic rule APP1, hence $e_s' = e_1\ e_2'$. By induction hypothesis it holds $\langle \Gamma; C \rangle \vdash e_2' : \tau_1$, thus the thesis follows from the typing rule TAPP.
2. case APP2
   In this case the premise of semantic rule APP2 says that the subexpression e_1 reduces ($C \vdash e_1 \rightarrow e_1'$) and that $e_s' = e_1'\ v$. By induction hypothesis we have $\langle \Gamma; C \rangle \vdash e_1 : \tau_1 \xrightarrow{\phi} \tau_2$, thus using the typing rule TAPP we prove the thesis.
3. case APP3
   In this case we have $e_s = (\mathbf{fun}\ f\ x \Rightarrow e)\ v$ and $e_s' = e\,\{\mathbf{fun}\ f\ x \Rightarrow e/f, v/x\}$. From our assumption we know that $\langle \Gamma; C \rangle \vdash \mathbf{fun}\ f\ x \Rightarrow e : \tau_1 \xrightarrow{\phi} \tau_2$ and $\phi \subseteq |C|$ and from Lemma 2.3.4 it holds $\langle \Gamma_{x,f}, x : \tau_1, f : \tau_1 \xrightarrow{\phi} \tau_2; C \rangle \vdash e : \tau_2$. Using two times the Lemma 2.3.5 for the variable x and f, we obtain the thesis.

- case TOP $\quad e_s = e_1\ \mathbf{op}\ e_2 \quad \langle \Gamma; C \rangle \vdash e_1 : \tau_c \quad \langle \Gamma; C \rangle \vdash e_2 : \tau_c$
  There are three semantic rules which drive the reduction $C \vdash e_s \rightarrow e_s'$:

  1. case OP1
     From the premise of the semantic rule OP1 we know that the subexpression e_1 reduces to e_1' ($C \vdash e_1 \rightarrow e_1'$), hence, $e_s' = e_1'\ \mathbf{op}\ e_2$. By induction hypothesis we have that $\langle \Gamma; C \rangle \vdash e_1' : \tau_c$, thus, the thesis follows from applying the rule TOP.
  2. case OP2
     In this case the subexpression e_2 reduces to e_2' ($C \vdash e_1 \rightarrow e_1'$ from the premise of the semantic rule), and $e_s' = c\ \mathbf{op}\ e_2'$. The thesis follows from applying the induction hypothesis ($\langle \Gamma; C \rangle \vdash e_2' : \tau_c$ holds) and then the typing rule TOP.
  3. case OP3
     In this case $e_s' = c$, than the thesis trivially holds by applying the rule TCONST.

- case TLET $\quad e_s = \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \quad \langle \Gamma; C \rangle \vdash e_1 : \tau_1 \quad \langle \Gamma_x, x : \tau_1, C \rangle \vdash e_2 : \tau_2$
  There are two rules from which $C \vdash e_s \rightarrow e_s'$ can be derived:

  1. case LET1
     In this case it holds $C \vdash e_1 \rightarrow e_1'$ (premise of the semantic rule), $e_s' = \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2$ and $\langle \Gamma; C \rangle \vdash e_1' : \tau_1$ by induction hypothesis. The thesis follows from applying the typing rule TLET.
  2. case LET2
     From the semantic rule we know $e_1 = v$ and $e_s' = e_2\{v/x\}$. By applying Lemma 2.3.5 the thesis holds.

- case TIF $\quad e_s = \mathbf{if}\ \ e_0\ \ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \quad \langle \Gamma; C \rangle \vdash e_0 : bool \quad \langle \Gamma; C \rangle \vdash e_1 : \tau$ $\langle \Gamma; C \rangle \vdash e_2 : \tau$
  There are three rules which drives the reduction of e_s:

1. case IF$_1$
From the premise of the semantic rule it holds $C \vdash e_0 \rightarrow e'_0$, hence $e'_s =$ **if** e_0 **then** e_1 **else** e_2. By using the induction hypothesis $\langle \Gamma; C \rangle \vdash e_0 : \mathit{bool}$, thus the thesis holds by applying the typing rule TIF.

2. case IF$_2$
From the semantic rule we know that $e'_s = e_1$, than the thesis trivially holds by our assumptions.

3. case IF$_3$
Similar to the previous case with e_2 in place of e_1.

□

The lemma below claims that we can easily deduce the syntactic form of a value from its type (it is exploited in Lemma 2.2.2).

Lemma 2.3.7 (Canonical form)**.** *Let v be a value then*

1. *If $\langle \Gamma; C \rangle \vdash v : \tau_c$ then $v = c$ for some c;*

2. *If $\langle \Gamma; C \rangle \vdash v : \tau_1 \xrightarrow{\phi} \tau_2$ then $v =$ **fun** $f\ x \Rightarrow e$ for some f,x and e;*

3. *If $\langle \Gamma; C \rangle \vdash e : ly_{\{L_1, \dots, L_n\}}$ then $v \in \{L_1, \dots, L_n\}$.*

Proof.

1. We have three kind of values only: constants, functions and layers. If v has type τ_c, our derivation consist of an application of the rule TCONST, followed by n (possibly $n = 0$) applications of the rule TSUB, hence, $v = c$ for some c.

2. It follows from a reasoning similar to the previous one.

3. The type ly with annotation $\{L_1, \dots, L_n\}$ can be only deduced by applying deduced the rule TSUB, starting from a type annotated with a singleton set $\{L\}$ for some $L \in \{L_1, \dots, L_n\}$. So this type can be obtained by the rule TLY only, hence, $v = L$.

□

Lemma 2.2.2 (Progress)**.** *Let e_s be a closed expression such that $\langle \Gamma; C \rangle \vdash e_s : \tau$ for some Γ and C. If $C \vdash e_s \not\rightarrow$, i.e. e_s is stuck, then e_s is a value.*

Proof. By induction of the depth of the typing derivation and then, by cases on the last rule applied. The cases TCONST, TLY, TFUN are immediate because e_s is a value. The case TVAR cannot occur because we assume that e_s is a closed. To prove the others cases, we assume that e_s is stuck and it is not a value and then we prove that this assumption leads to contraction.

- case TSUB
Straightforward by induction hypothesis.

- case TWITH $\qquad e_s \;=\; \textbf{with}(e_1)\ \textbf{in}\ e_2$
  There are only two cases in which e_s can be stuck:

  1. case e_1 is stuck
     By induction hypothesis e_1 is a value. By the premise of the typing rule TWITH it gets ly_ϕ as type and by the Lemma 2.3.7 (3) it holds $e_1 = L$ for some $L \in \phi$. If e_2 reduces, then we use the semantic rule WITH$_2$ to reduces e_s (contradiction), otherwise we are in the case (2).
  2. case e_1 is a value and e_2 is stuck
     By the induction hypothesis e_2 is a value, so we have a contradiction because we can reduce e_s by using the semantic rule WITH$_3$.

- case TWITHOUT
  Similar to the case TWITH.

- case TLEXP $\qquad e_s \;=\; L_1.e_1, \ldots, L_n.e_n$
  If e_s is stuck, then the dispatching mechanism failed to find a match between the layers of e_s and the context C. But by the premise of the typing rule TLEXP, we know that $\bigvee_i L_i \in |C|$ holds, i.e. there exists at least one of the layers of e_s (say L_i) which is the in context C. Then the invocation $dsp(C, \{L_1, \ldots, L_n\}) = L_i$ and e_s reduces to e_i by the semantic rule LEXP (contradiction).

- case TAPP $\qquad e_s \;=\; e_1\ e_2$
  If e_s is stuck then there are two only cases:

  1. e_2 is stuck
     By induction hypothesis e_2 is a value. If e_1 reduces, then the semantic rule APP$_2$ applies, so there is a contradiction because e_s reduces too. If e_1 is stuck, we are in case (2).
  2. e_2 is a value and e_1 is stuck
     By induction hypothesis e_1 is a value, by the premise of the typing rule TAPP it gets a function type and by Lemma 2.3.4 (2) $e_1 = \textbf{fun}\ f\ x \Rightarrow e$.

- case TOP $\qquad e_s \;=\; e_1\ \textbf{op}\ e_2$
  There are only two cases when e_s is stuck:

  1. e_1 is stuck.
     By induction hypothesis e_1 is a value. If e_2 reduces, then the semantic rule OP$_2$ applies, so we obtain a contradiction because e_s reduces. If e_2 is stuck, we are in the case (2).
  2. e_1 is a value and e_2 is stuck
     By induction hypothesis e_2 is a value. By the premise of the typing rule TOP e_1 and e_2 gets τ_c as type and then by Lemma 2.3.7 (1) $e_1 = c_1$ and $e_2 = c_2$. So the semantic rule OP$_3$ applies, and the e_s reduces (contradiction).

- case TLET $\qquad e'_s \;=\; \textbf{let}\ x = e_1\ \textbf{in}\ e_2$
  If e_s is stuck, then it is only the case that e_1 is stuck and by induction hypothesis

this can only occur when e_1 is a value. So we can apply the semantic rule $\textsc{let}_2$ to reduce e_s (contradiction).

- case $\textsc{Tif}$ $\qquad e'_s = \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2$
  If e_s is stuck, then it is only the case that e_1 is stuck. By induction hypothesis it can only occur when e_1 is a value v. Since $\langle \Gamma; C \rangle \vdash v : bool$ by our hypothesis and $v = true$ or $v = false$ by Lemma 2.3.7 (1). So either rule $\textsc{if}_2$ or $\textsc{if}_3$ applies and hence, e_s cannot be stuck (contraction).

□

Theorem 2.2.3. *Let e_s be a closed expression if $\langle \emptyset; C \rangle \vdash e_s : \tau$ for some C, either the computation terminates by yielding a value ($C \vdash e_s \rightarrow^* v$) or it diverges, but it **never** gets stuck.*

Proof (By contradiction). Assume that $C \vdash e_s \rightarrow^i e'_s \not\rightarrow$ for some $i \in \mathbb{N}$ where e'_s is a non-value stuck expression. By applying i times the Preservation Lemma we know that $\langle \Gamma; C \rangle \vdash e'_s : \tau$, then by using the Progress Lemma we know that e'_s is a value, hence, a contradiction. □

2.4 Remarks

Our type system is based on the assumption which the initial context is known at compile time and that all changes performed over it can be determined from the application code. Note that the approaches in [HIM11, CS09] adopt the same assumption. However, our proposal differs from the ones in [HIM11, CS09, CCT09] because in ContextML layers are values. This fact affects the type system and requires a special treatment, i.e. the annotations ϕ in the layer types. Furthermore, with respect to [CCT09], ContextML introduces behavioural variation as expressions, but it does not have any construct similar to `proceed` to avoid the problematic situation analysed in [CCT09]. Despite its simplicity, ContextML is the kernel of the material presented in the Chapter 3 and its design influenced ML_{CoDa} presented in Chapter 4.

Chapter 3

A Methodology for Programming Context-aware Components

The development of complex adaptive systems presents issues that cannot be tackled using COP primitives only. In fact, as described in Section 1.1.1, a typical characteristic of these systems is that they are made up of a massive number of interacting components. Each component is able to modify its behaviour, can access its private resources and has to satisfy a set of non-functional requirements (e.g. QOS, security, etc.). A system behaves correctly when each component satisfies its own non-functional requirements and correctly interacts with others components, i.e. it respects the communication protocol.

The aim of this chapter is to introduce a language-based methodology for programming complex adaptive software and for ensuring their correctness. The main contribution is the definition of a static analysis technique that guarantees the enforcement of security policies and the correctness of interactions among components. We assume a model (see Figure 3.1) where each adaptive component has

- mechanisms to manipulate the context;
- an abstract and declarative representation of the operational environment;
- security policies governing behaviour and resource usages;
- a protocol ruling the messages exchanged with other parties.

We adopt a *top-down* approach [C^+09] to describe the interactions with other components, because we do not want to wire a component to a specific communication infrastructure. For this reason, we consider a communication model based on a bus through which messages are exchanged. Moreover, we assume in the following formal development that the other components respect the communication protocol and never fail.

Under these assumptions we define a methodology which suitably extends and integrates together techniques from COP, type theory and model-checking. In particular, it consists of a static technique ensuring that a component

(i) adequately reacts to context changes;

(ii) accesses resources in accordance with security policies;

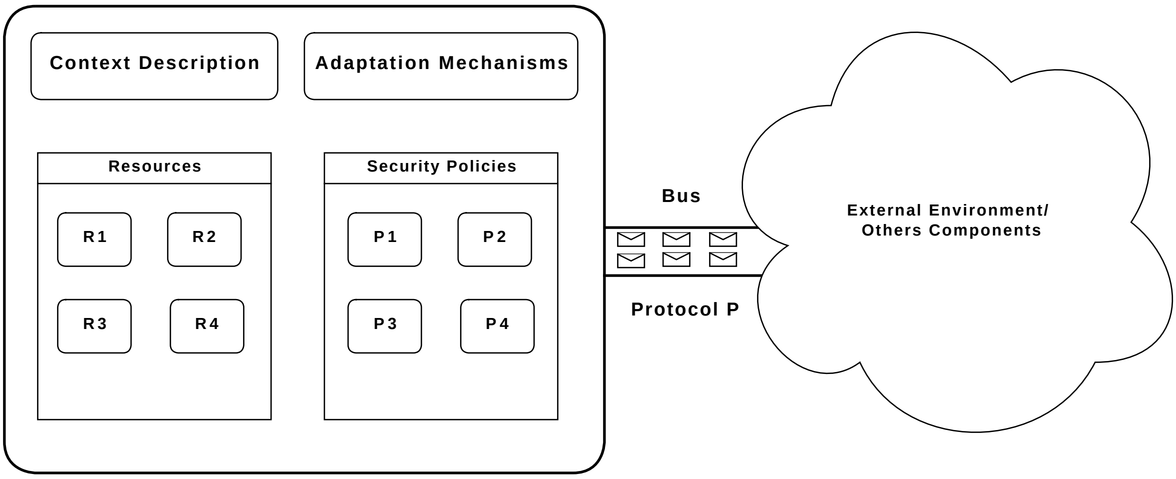


Figure 3.1: Our component model. We assume that each component has a private declarative description of the context and its sets of resources, security policies and adaptation mechanisms. Security policies rule both behaviour and resource usage. The bus is the unique point of interaction with others components and communication through it is governed by a protocol *P*.

(*iii*) exchanges messages on the bus, complying with a specific communication protocol provided by the operational environment.

Technically, the contribution of this chapter consists of three points:

1. we extend ContextML with constructs for resource manipulation (in the spirit of [BDFZ09]); mechanisms to declare and enforce security policies by adopting the local sandbox approach of [BDFZ09]; and the introduction of message passing constructs for the communication with external parties.

2. We design a type and effect system (an extension of the type system presented in Chapter 2) for ensuring that programs adequately react to context changes and for computing as effect an abstract representation of the overall behaviour. This representation, in the form of *History Expressions*, describes the sequences of resource manipulation and communication with external parties in an abstract form.

3. We model check effects to verify that the component behaviour is correct, i.e. that adaptation and actions over resources are in accordance with the security policies and that the communication protocol is respected. The model checking is performed in two phases:

   (a) adherence to security policies (*safety*)

   (b) compliance with the protocol (*compliance*).

The chapter is organized as follows. In the next section we introduce a motivating example to illustrate our methodology. In Section 3.2 we extend ContextML with the new constructs. Section 3.3 describes our type and effect system and Section 3.4 our model-checking procedure.

3.1 A mobile application for a library of e-books

Consider a simple scenario consisting of a smartphone app that uses some services supplied by a cloud infrastructure. The cloud offers a repository to store and synchronize a library of e-books and computational resources to execute customised applications among which there is a full-text search.

A user buys e-books online and reads them locally through the app. The purchased e-books are stored in the remote user library and some of them are kept locally in the smartphone. The two libraries may not be synchronized. The synchronization is triggered on demand and depends on several factors: the actual bandwidth available for the connection, the free space on the device, etc. We specify below the fragment of the app that implements the full-text search over the user's library.

As in Chapter 2 the context dependent behaviour emerges because of the different energy profiles of the smartphone. We assume that there are two profiles: one is active when the device is plugged in, the other one when the battery is used. These profiles are represented by two *layers*: `ACMode` and `BatMode`. The function `getEnergyProfile` returns the layer describing the current active profile. The actual energy profile depends on the value returned by the sensor which is queried through the function `isPlugged`:

```
fun getEnergyProfile () =
  if isPlugged () then
    ACMode
  else
    BatMode
```

As usual layers can be activated to modify the context through the **with** construct. In Figure 3.2a we display the code for the part of the application running on the smartphone. The code consists of a **with** construct (lines 13-14) that activates the layer representing the actual energy profile and calls the function `ftsearch` (defined at lines 1-11). This function is formed by nested layered expressions describing behavioural variations matching different configurations of the execution environment. The code exploits context dependency to take into account also the actual location of the execution engine (remote in the cloud at line 3 or local on the device at line 4), the synchronization state of the library at lines 5-6 and the active energy profile at lines 2 and 10. The smartphone communicates with the cloud system over the bus through message passing primitives at lines 7-9. Communication primitives are labelled with the types of the transmitted values. In our example τ denotes the type ly_ϕ where ϕ is a set of layers including `ACMode`; the type τ' is the functional type `unit` $\xrightarrow{\mathbb{P}|H}$ τ'' (see later for details about $\mathbb{P}$ and H); the type of search result is denoted by τ''.

The search is performed locally only if the library is fully synchronized and the smartphone is plugged in. If the device is plugged in but the library is not fully synchronized, then the code of function `ftsearch` is sent to the cloud and is executed remotely by a suitable server.

In Figure 3.2b we show a fragment of the environment provided by the cloud infrastructure. The considered service offers generic computational resources to the devices connected on the bus by continuously running the function `serve`. At the line 2, it listens

```
1  fun ftsearch x =
2    ACMode.
3      OnCloud. search ()
4      OnSmartphone.
5        LibrarySync. search ()
6        LibraryUnsyc.
7          sendτ (ACMode);
8          sendτ' (ftsearch);
9          receiveτ''
10   BatMode.
   (* do something else *)
11
12 ....
13 with(getEnergyProfile ())
14   ftsearch ()
```

(a) The code running on the smartphone

```
1  fun serve x =
2    let lyr = receiveτ in
3    let g = receiveτ' in
4    ϕ [
5      with(lyr) in
6        let res = g () in
7        sendτ''(res)
8    ];
9    serve ()
10 ...
11 serve ()
```

(b) The code of the cloud infrastructure

Figure 3.2: The code of the e-library application

to the bus for incoming code (assumed to be a function) and an incoming layer (line 3). Then, it executes the received function (line 6) in a context extended with the received layer (line 5). In the code of the cloud at the line 4, there is a security policy ϕ which is enforced while running the received code from the bus. The enforcement is expressed by a special construct called security framing $\phi[\ldots]$, that causes the enclosed expression to be executed under the strict monitoring of ϕ. In this chapter, we are not interested in showing either how the policies are defined or their concrete syntax. In Section 3.4 we will see that policies are equivalent to finite automata, so the reader can suppose them to be written in the code as regular expressions or as a (subset of) LTL formulae. Here, assume ϕ to be a policy which forbids the received code from writing on the library (represented by the action `write(library)`), but it always allows reading. The framing guarantees that the execution of foreign code does not alter the remote library. In this example, the policy ϕ only concerns actions on resources, e.g. the library, but our approach also allows us to enforce security policies governing behaviour adaptation and communication.

The cloud system constrains communications on the bus by declaring a protocol P, that prescribes the viable interactions. Additionally, the cloud infrastructure will make sure that the protocol P is indeed an abstraction of the behaviour of the various services involved in the interactions. Here we are not interested in the problem of how protocols are defined by the environment, but only in checking whether a client respects the given protocol.

The protocol to communicate with the cloud infrastructure is

$$P = (send_{\tau} send_{\tau'} receive_{\tau''})^*$$

and expresses the sequence of communication actions which a client has to perform. Specially, it requires that the client must send a value of type τ, then a value of type τ' and then must receive back a value of type τ''. These actions can be repeated a certain number of times as indicated by the symbols $*$.

A static analysis technique is exploited to verify that all security policies are enforced (*safety*) and that the communication protocol is respected (*compliance*). As already said, our technique consists of a type and effect system and a model-checking procedure. Below we explain how it applies to our example.

Function getEnergyProfile returns a value of type $ly_{\{\texttt{ACMode},\texttt{BatMode}\}}$, i.e. the returned layer is one of ACMode and BatMode.

The type of function ftsearch is $\tau' = \texttt{unit} \xrightarrow{\mathbb{P}|H} \tau''$, assuming that the value returned by the search function has type τ''. The type τ' is annotated by a set of preconditions $\mathbb{P}$ (see below) and a latent effect H (discussed later on).

$$\mathbb{P} = \{\{\texttt{ACMode},\texttt{IsLocal},\texttt{LibrarySynced}\},\{\texttt{ACMode},\texttt{IsCloud}\},\dots\}$$

Each precondition in $\mathbb{P}$ is a set of layers. To apply getEnergyProfile, the context of the application must contains all the layers in v, for a precondition $v \in \mathbb{P}$.

As we will see later on, our type system is an extension of the one of Chapter 2. Not only it guarantees that the dispatching mechanism always succeeds at runtime but also it computes an effect H (history expression). This effect represents an over-approximation of the sequences of events, i.e. resource manipulations or layer activations or communication actions. In our example, the **with** will be well-typed whenever the context in which it will be evaluated contains IsLocal or IsCloud and LibraryUnsync or LibrarySync. The requirements about ACMode and BatMode coming from the body of the function ftsearch are ensured at the line 13. This is due to the type of getEnergyProfile guarantees that one of them will be activated in the context by the **with**. The effect H in τ' is the latent effect of ftsearch, over-approximating the set of histories, i.e. the sequences of events, possibly generated by ftsearch.

Effects are then used to check whether a client complies with the policy and the interaction protocol provided by the environment. Verifying that the code of ftsearch obeys the policy ϕ is done by model-checking the effect of ftsearch (a context-free language) against the policy ϕ (a regular language). Obviously, the app never writes, so the policy ϕ is satisfied, assuming that the code for the BatMode case has empty effect.

To check compliance with the protocol, we only considering communications. Thus, the effect of the body of ftsearch becomes:

$$H_{sr} = send_{\tau} \cdot send_{\tau'} \cdot receive_{\tau''}$$

Verifying whether the program correctly interacts with the cloud system consists of checking that the histories generated by H_{sr} are a subset of those allowed by the protocol P. In our scenario this is indeed the case.

3.1.1 Expressing Role-based Access Control Policies

Here, we extend our running example to show that contexts can include information on principals using resources, and that we can implement a kind of role based access

```
1  fun ftsearch1 x =
2    ACMode.
3      OnCloud. search ()
4      OnSmartphone.
5        LibrarySync. search ()
6        LibraryUnsyc.
7          send$_{\tau_{id}}$(id_code);
8          send$_{\tau}$ (ACMode);
9          send$_{\tau'}$ (ftsearch);
10         receive$_{\tau''}$
11   BatMode.
   (* do something else *)
12
13 ....
14 with(getEnergyProfile ())
15   ftsearch1 ()
```

(a) The code running on the smartphone

```
1  fun serve1 x =
2  $\phi'$[ with(Root) in
3     let id = receive$_{\tau_{id}}$ in
4       cd(/billing);
5       write(bid_id);
6       cd(/lib_id);
7     with(Usr) in
8       let lyr = receive$_{\tau}$ in
9       let g = receive$_{\tau'}$ in
10      $\phi$ [
11        with(lyr) in
12          let res = g () in
13          send$_{\tau''}$(res)
14      ];
15      serve1 ()
16 ]
17 ...
18 serve1 ()
```

(b) The code of the cloud infrastructure

Figure 3.3: The code of the e-library application extended with the role based access control. The new code is underlined and in blue.

control by our policies. Assume that each user has an unique identifier needed to access the cloud services. The cloud uses the id to grant services and to update information about user's account, e.g. charging money for offered services. Figure 3.3a shows the new version of the function `ftsearch`: the app on the smartphone sends the user's id to the cloud at line 7. In the function `serve1` the layer `Root` (line 2) is activated to indicate that the code is running with administrator privileges, i.e. it can access all system resources. Once the user's id has been received, the information about the bill is updated (line 4-5) and the current working directory is changed into the one containing the user's library (line 6). Then, the layer `User` is activate and the same code of `serve` is executed with user privileges.

Regarding security, `serve1` has two policies ϕ and ϕ'. The first one is the same as before (it forbids writing on the library). The second one, instead, specifies infrastructural rules of the Cloud. Among the various controls, it inhibits the code running with user privileges, (i.e. with the layer *Usr* active in the context) to perform operations allowed only to the administrator, e.g. changing the working directory or modifying system files. This is possible because policies can also require that contexts satisfy certain properties.

3.2 Extending ContextML

We extend ContextML by introducing resource manipulation, enforcement of security properties and communication.

Resources available in the system are represented by identifiers and can be manipulated by a fixed set of actions. For simplicity, we omit the fact that resources can have a complex structure with their own private state, but we treat them as abstract data types. In addition, we do not provide ContextML with constructs for dynamically creating resources, but these can be added following [BDFZ09, BDF09].

We enforce security properties by protecting expressions with policies: $\phi[e]$. This mechanism is known in the literature as *policy framing* [BDF09]. Roughly, it means that during the evaluation of e the computation must respect ϕ. Our policies turn out to be regular properties of special computation traces, called histories; more details are in Section 3.4.

The communication model is based on a bus which allows programs to interact with the environment by message passing. The operations of writing and reading values over this bus can be seen as a simple form of asynchronous I/O. We will not specify this bus in detail, but we will consider it as an abstract entity representing the whole external environment and its interactions with programs. Therefore, ContextML programs operate in an open-ended environment. The syntax and the structural operational semantics of ContextML follow.

3.2.1 Syntax

Let *Const* be a set of constants, Ide a set of identifiers, $\mathsf{LayerNames}$ a finite set of layer names, $\mathsf{Policies}$ a set of security policies, Res a finite set of resources identifiers and Act a finite set of actions for manipulating resources. Then, the syntax of ContextML is:

$c \in Const$ $\quad x, f \in \mathsf{Ide}$ $\quad L \in \mathsf{LayerNames}$ $\quad \phi \in \mathsf{Policies}$ $\quad r \in \mathsf{Res}$ $\quad \alpha, \beta \in \mathsf{Act}$

$v ::=$	*values*	
$\mid c$	*constants*	e.g. $1, 2, \ldots,$ `true`, `false`, $()$
$\mid L$	*layers*	
$\mid \textbf{fun}\ f\ x \Rightarrow e$	*functions*	
$e ::=$	*expressions*	
$\mid v$	*values*	
$\mid x$	*identifiers*	
$\mid e_1\ e_2$	*function application*	
$\mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2$	*declaration*	
$\mid e_1\ \textbf{op}\ e_2$	*operators*	e.g. $+, -, \times, \ldots$
$\mid \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2$	*conditional*	
$\mid \textbf{with}(e_1)\ \textbf{in}\ e_2$	*layers activation*	

$$
\begin{array}{lll}
 & \mid \mathbf{without}(e_1)\ \ \mathbf{in}\ \ e_2 & \textit{layers deactivation} \\
 & \mid lexp & \textit{layered expressions} \\
 & \mid \underline{\alpha(r)} & \underline{\textit{resource manipulation}} \\
 & \mid \underline{\varphi\,[e]} & \underline{\textit{enforcement of security policy}} \\
 & \mid \underline{\mathbf{send}_\tau} & \underline{\textit{communication}} \\
 & \mid \underline{\mathbf{receive}_\tau} & \underline{\textit{communication}} \\
 & \mid \underline{auxe} & \underline{\textit{auxiliary expressions}} \\
lexp ::= L.e \mid L.e, lexp & & \\
auxe ::= & & \\
 & \mid \mathbf{with}(\bar{L})\ \textit{in}\ e_2 & \\
 & \mid \mathbf{without}(\bar{L})\textit{in}\ e_2 & \\
 & \mid \bar{\varphi}[e] &
\end{array}
$$

Additionally, we assume the syntactic sugar $e_1; e_2$ for $(\mathbf{fun}\ f x \Rightarrow e_2)\, e_1$ where x and f are not free in e_2.

The novelties of ContextML with respect to the previous version of the Chapter 2 (underlined and in blue) are primitives for handling resources, policy framing and communication. The expression $\alpha(r)$ indicates that we access the resource r through the action α, possibly causing side effects. The security properties are enforced by policy framing $\varphi[e]$ guaranteeing that the computation satisfies the policy φ. Of course, policy framings can be nested. The communication is performed by $\mathbf{send}_\tau$ and $\mathbf{receive}_\tau$. They allow us to interact with the external environment by writing and reading values of type τ to and from the bus. The auxiliary expressions *auxe* are not used by the programmer but they are needed in the dynamic semantics for intermediate configurations (see below).

3.2.2 Dynamic Semantics

We provide a new small-step operational semantics for ContextML programs, defined as usual only for closed expressions. Note that, since now ContextML programs can read values from the bus, a closed expression can be open with respect to the external environment. For example, consider the expression $\mathbf{let}\ x = \mathbf{receive}_\tau\ \mathbf{in}\ x + 1$. It is closed but it reads an unknown value v from the bus. To give meaning to such programs, we use an approach similar to the early input of the π-calculus [SW01].

Furthermore, the semantics is history dependent. Program *histories* are sequences of interesting events that may arise at runtime. In our case, events *ev* record the activation and the deactivation of layers, the dispatching of behavioural variations and program actions, such as resource accesses, entering and exiting policy framing and communication. The syntax of events *ev* and programs histories η is the following:

$$ev ::= (\!|_L \mid |\!)_L \mid \{_L \mid \}_L \mid \mathsf{Disp}(L) \mid \alpha(r) \mid send_\tau \mid receive_\tau \mid [_\varphi \mid]_\varphi$$

$$\eta ::= \epsilon \mid ev \mid \eta\,\eta$$

The event $(\!|_L$ marks that we begin the evaluation of a **with** body in a context where the layer L is activated; the event $|\!)_L$ marks the end of the activation; symmetrically, the event $\{_L$ signals that we begin the evaluation of a **without** body in a context where the layer L is masked; instead the $\}_L$ signals the end of the masking; the event $\mathsf{Disp}(L)$ signals that layer L has been selected by the dispatch mechanism; the event $\alpha(r)$ marks that the action α has been performed over the resource r; the event $send_\tau$ indicates that we send a value of type τ over the bus; symmetrically, the event $receive_\tau$ records that we read a value from the bus; the event $[_\phi$ marks the beginning of the enforcement of the policy ϕ; instead, the event $]_\phi$ marks the end. As before a context C is a stack of active layers and in the following we use the operation :: and $-$ of Definition 2.2.1 and Definition 2.2.2.

The transitions have the form $C \vdash \eta, e \to \eta', e'$, meaning that in the context C, starting from a program history η, the expression e may evolve to e' and the history η to η' in one evaluation step.

The semantic rules are shown in Figures 3.4 and 3.5. Most of them are inherited from the Chapter 2 but we adapted them to deal with the history. We briefly comment on those about new constructs.

The rules for $\textbf{with}(e_1)$ **in** e_2 are similar to those presented previously, but additionally we store in the history the events $(\!|_L$ and $|\!)_L$ marking the beginning and the end of the evaluation of e_2. Note that the fact of being within the scope of the activation of the layer L is recorded by using the auxiliary expression $\textbf{with}(\overline{L})$ **in** e_2. Symmetrically, the rules for $\textbf{without}(e_1)$ **in** e_2 evaluate e_2 in a context where the layer obtained evaluating e_1 is deactivated. Moreover, they record the events $\{_L$ and $\}_L$ in the history to mark the beginning and the end of the evaluation of e_2. Also in this case we denote with $\textbf{without}(\overline{L})$ **in** e_2 the fact of being within the scope of the deactivation of the layer L.

The new version of the rule LEXP stores in the history which case of the layered expression is selected by the dispatching mechanism (Definition 2.2.3). Also in this case the computation gets stuck, if no layer matches.

The rule ACTION establishes that performing an action α over a resource r yields the unit value () and extends η with $\alpha(r)$.

The rules governing communication reflect our notion of protocol, which abstractly represents the behaviour of the environment, showing the sequence of direction/type of messages. Accordingly, our primitives carry types as tags, rather than dynamically checking the exchanged values. In particular, there is no check that the type of the received value matches the annotation of the receive primitive. Our static analysis will guarantee the correctness of this operation.

In detail, $\textbf{send}_\tau(e)$ evaluates e and sends the obtained value over the bus. Additionally, the history is extended with the event $send_\tau$. A $\textbf{receive}_\tau$ reduces to the value v read from the bus and appends the corresponding event to the current history. This rule is similar to that used in the early semantics of the π-calculus, where a name is guessed and transmitted over the channel [SW01].

The rules for framing say that an expression $\phi[e]$ can reduce to $\phi[e']$, provided that the resulting history η', purged of all security framing events ($\eta'^{-[]}$), obeys the policy ϕ, in symbols $\eta'^{-[]} \models \phi$ (see next Sections for a precise definition). Also here, placing a bar

$$\textsc{with}_1\quad \frac{C \vdash \eta, e_1 \to \eta', e_1'}{C \vdash \eta, \mathbf{with}(e_1)\ \mathbf{in}\ e_2 \to \eta', \mathbf{with}(e_1')\ \mathbf{in}\ e_2}$$

$$\textsc{with}_2\quad \frac{}{C \vdash \eta, \mathbf{with}(L)\ \mathbf{in}\ e \to \eta \llparenthesis_L, \mathbf{with}(\bar{L})\ \mathbf{in}\ e}$$

$$\textsc{with}_3\quad \frac{L :: C \vdash \eta, e \to \eta', e'}{C \vdash \eta, \mathbf{with}(\bar{L})\ \mathbf{in}\ e \to \eta', \mathbf{with}(\bar{L})\ \mathbf{in}\ e'}$$

$$\textsc{with}_4\quad \frac{}{C \vdash \eta, \mathbf{with}(\bar{L})\ \mathbf{in}\ v \to \eta \rrparenthesis_L, v}$$

$$\textsc{without}_1\quad \frac{C \vdash \eta, e_1 \to \eta', e_1'}{C \vdash \eta, \mathbf{without}(e_1)\ \mathbf{in}\ e_2 \to \eta', \mathbf{without}(e_1')\ \mathbf{in}\ e_2}$$

$$\textsc{without}_2\quad \frac{}{C \vdash \eta, \mathbf{without}(L)\ \mathbf{in}\ e \to \eta \{_L, \mathbf{without}(\bar{L})\ \mathbf{in}\ e}$$

$$\textsc{without}_3\quad \frac{C - L \vdash \eta, e \to \eta', e'}{C \vdash \eta, \mathbf{without}(\bar{L})\ \mathbf{in}\ e \to \eta', \mathbf{without}(\bar{L})\ \mathbf{in}\ e'}$$

$$\textsc{without}_4\quad \frac{}{C \vdash \eta, \mathbf{without}(\bar{L})\ \mathbf{in}\ v \to \eta \}_L, v}$$

$$\textsc{lexp}\quad \frac{dsp(C, \{L_1, \ldots, L_n\}) = L_i}{C \vdash \eta, L_1.e_1, \ldots, L_n.e_n \to \eta\ \mathsf{Disp}(L_i), e_i}$$

$$\textsc{action}\quad \frac{}{C \vdash \eta, \alpha(r) \to \eta\ \alpha(r), ()}$$

$$\textsc{send}_1\quad \frac{C \vdash \eta, e \to \eta', e'}{C \vdash \eta, \mathbf{send}_\tau(e) \to \eta', \mathbf{send}_\tau(e')}$$

$$\textsc{send}_2\quad \frac{}{C \vdash \eta, \mathbf{send}_\tau(v) \to \eta\ send_\tau, ()}$$

$$\textsc{receive}\quad \frac{}{C \vdash \eta, \mathbf{receive}_\tau \to \eta\ receive_\tau, v}$$

$$\textsc{framing}_1\quad \frac{\eta^{-[]} \models \phi}{C \vdash \eta, \phi[e] \to \eta [_\phi, \overline{\phi}[e]}$$

$$\textsc{framing}_2\quad \frac{C \vdash \eta, e \to \eta', e' \qquad \eta'^{-[]} \models \phi}{C \vdash \eta, \overline{\phi}[e] \to \eta', \overline{\phi}[e']}$$

$$\textsc{framing}_3\quad \frac{\eta^{-[]} \models \phi}{C \vdash \eta, \overline{\phi}[v] \to \eta]_\phi, v}$$

Figure 3.4: The semantics rules of extended ContextML: part I

$$\textsc{app}_1\quad \frac{C \vdash \eta, e_2 \to \eta', e_2'}{C \vdash \eta, e_1\, e_2 \to \eta', e_1\, e_2'} \qquad \textsc{app}_2\quad \frac{C \vdash \eta, e_1 \to \eta', e_1'}{C \vdash \eta, e_1\, v \to \eta', e_1'\, v}$$

$$\textsc{app}_3\quad \frac{}{C \vdash \eta, (\textbf{fun}\ f\ x \Rightarrow e)v \to \eta, e\{\textbf{fun}\ f\ x \Rightarrow e/f, v/x\}}$$

$$\textsc{if}_1\quad \frac{C \vdash \eta, e_0 \to \eta', e_0'}{C \vdash \eta, \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \to \eta', \textbf{if}\ e_0'\ \textbf{then}\ e_1\ \textbf{else}\ e_2}$$

$$\textsc{if}_2\quad \frac{}{C \vdash \eta, \textbf{if}\ \texttt{true}\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \to \eta, e_1} \qquad \textsc{if}_3\quad \frac{}{C \vdash \eta, \textbf{if}\ \texttt{false}\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \to \eta, e_2}$$

$$\textsc{let}_1\quad \frac{C \vdash \eta, e_1 \to \eta', e_1'}{C \vdash \eta, \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \to \eta', \textbf{let}\ x = e_1'\ \textbf{in}\ e_2} \qquad \textsc{let}_2\quad \frac{}{C \vdash \eta, \textbf{let}\ x = v\ \textbf{in}\ e_2 \to \eta, e_2\{v/x\}}$$

$$\textsc{op}_1\quad \frac{C \vdash \eta, e_1 \to \eta', e_1'}{C \vdash \eta, e_1\ \textbf{op}\ e_2 \to \eta', e_1'\ \textbf{op}\ e_2} \qquad \textsc{op}_2\quad \frac{C \vdash \eta, e_2 \to \eta', e_2'}{C \vdash \eta, v\ \textbf{op}\ e_2 \to \eta', v\ \textbf{op}\ e_2'} \qquad \textsc{op}_3\quad \frac{v = v_1\ \texttt{op}\ v_2}{C \vdash \eta, v_1\ \textbf{op}\ v_2 \to \eta, v}$$

Figure 3.5: The semantics rules of extended ContextML: part II

over ϕ records that the policy is active. If η' does not obey ϕ, then the computation gets stuck. Of course, we store in the history through $[_\phi/]_\phi$ the point where the enforcement of ϕ starts/ends.

Example 3.2.1. Back to our motivating example in Section 3.1, consider the part of the application running on the smartphone and assume to invoke the function `ftsearch` in the context $C = [\texttt{ACMode}, \texttt{OnSmartphone}, \texttt{LibraryUnSync}]$ and with a history η. The following evaluation shows how the mechanism of histories and the communication work:

$$
\begin{array}{l}
[\texttt{ACMode}, \texttt{OnSmartphone,LibrarySync}] \vdash \\
\qquad\qquad \eta, \texttt{ftsearch}\ () \rightarrow^* \\
\qquad\qquad \eta_1, \left(\begin{array}{l} \texttt{OnCloud.search()} \\ \texttt{OnSmartphone.} \\ \quad \texttt{LibrarySync.search()} \\ \quad \texttt{LibraryUnsync.} \\ \qquad \textbf{send}_{\tau}(\texttt{ACMode}); \\ \qquad \textbf{send}_{\tau'}(\texttt{ftseach}); \\ \qquad \textbf{receive}_{\tau''}; \end{array} \right) \rightarrow^* \\
\qquad\qquad \eta_2, \left(\begin{array}{l} \textbf{send}_{\tau}(\texttt{ACMode}); \\ \textbf{send}_{\tau'}(\texttt{ftseach}); \\ \textbf{receive}_{\tau''}; \end{array} \right) \rightarrow \\
\qquad\qquad \eta_3, \left(\begin{array}{l} \textbf{send}_{\tau'}(\texttt{ftseach}); \\ \textbf{receive}_{\tau''}; \end{array} \right) \rightarrow \\
\qquad\qquad \eta_4, \textbf{receive}_{\tau''} \rightarrow \\
\qquad\qquad \eta_5, v
\end{array}
$$

where the histories generated during the evaluation are:

$\eta_1 = \eta Disp(\texttt{ACMode})$ because the dispatching mechanism matched the layer `ACMode`;

$\eta_2 = \eta_1 Disp(\texttt{OnSmartphone}) Disp(\texttt{LibraryUnsync})$ because the layers `OnSmartphone` and `LibraryUnsync` was selected in order;

$\eta_3 = \eta_2\, send_\tau$ because we sent the layer `ACMode` over the bus;

$\eta_4 = \eta_3\, send_{\tau'}$ because the function `ftsearch` was written on the bus;

$\eta_5 = \eta_4\, receive_{\tau''}$ because we read the value v from the bus.

After the third transition the layer `ACMode` and the code of the function `ftsearch` are we sent on the bus. These value are read from the bus by the cloud server running the function `serve`. We assume that the function `serve` is run in context $C' = [\texttt{OnCloud}]$ and with a history η' (remember that the policy ϕ prevents the received code from modifying the library):

$$C' \vdash \eta', \texttt{serve}\ () \rightarrow$$

$$
\begin{array}{l}
\eta', \left(\begin{array}{l}
\mathbf{let}\ \mathtt{lyr} = \mathbf{receive}_{\tau}\ \mathbf{in}\\
\mathbf{let}\ \mathrm{g} = \mathbf{receive}_{\tau'}\ \mathbf{in}\\
\quad \phi[\mathbf{with}(\mathtt{lyr})\ \mathbf{in}\\
\qquad\quad \mathbf{let}\ \mathtt{res} = \mathrm{g}\ ()\ \mathbf{in}\\
\qquad\qquad \mathbf{send}_{\tau''}(\mathtt{res})]\\
\mathit{serve}\ ()
\end{array}\right) \to^{*}\\
\eta'_1, \left(\begin{array}{l}
\mathbf{let}\ \mathrm{g} = \mathbf{receive}_{\tau'}\ \mathbf{in}\\
\quad \phi[\mathbf{with}(\mathtt{ACMode})\ \mathbf{in}\\
\qquad\quad \mathbf{let}\ \mathtt{res} = \mathrm{g}\ ()\ \mathbf{in}\\
\qquad\qquad \mathbf{send}_{\tau''}(\mathtt{res})]\\
\mathit{serve}\ ()
\end{array}\right) \to^{*}\\
\eta'_2, \left(\begin{array}{l}
\phi[\mathbf{with}(\mathtt{ACMode})\ \mathbf{in}\\
\qquad \mathbf{let}\ \mathtt{res} = \mathtt{ftsearch}\ ()\ \mathbf{in}\\
\qquad\quad \mathbf{send}_{\tau''}(\mathtt{res})]\\
\mathit{serve}\ ()
\end{array}\right) \to^{*}\\
\eta'_3, \left(\begin{array}{l}
\overline{\phi}[\mathbf{with}(\overline{\mathtt{ACMode}})\ \mathbf{in}\\
\qquad \mathbf{let}\ \mathtt{res} = \mathtt{ftsearch}\ ()\ \mathbf{in}\\
\qquad\quad \mathbf{send}_{\tau''}(\mathtt{res})]\\
\mathit{serve}\ ()
\end{array}\right) \to^{*}\\
\eta'_4, \left(\begin{array}{l}
\overline{\phi}[\mathbf{with}(\overline{\mathtt{ACMode}})\ \mathbf{in}\\
\qquad \mathbf{send}_{\tau''}(v)]\\
\mathit{serve}\ ()
\end{array}\right) \to^{*}\\
\eta'_5, \left(\begin{array}{l}
\overline{\phi}[\mathbf{with}(\overline{\mathtt{ACMode}})\ \mathbf{in}\\
\qquad ()]\\
\mathit{serve}\ ()
\end{array}\right) \to^{*}\\
\eta'_6, \mathit{serve}\ ()
\end{array}
$$

The evaluation yields the following histories:

$\eta'_1 = \eta' \mathit{receive}_{\tau}$ — the layer ACMode is read from the bus;

$\eta'_2 = \eta'_1 \mathit{receive}_{\tau'}$ — the code of the function ftsearch is taken from the bus;

$\eta'_3 = \eta'_2 [_{\phi}\ ⦇_{\mathtt{ACMode}}$ — the enforcement of the policy ϕ is started and the layer ACMode is activated in the context;

$\eta'_4 = \eta'_3 \mathit{Disp}(\mathtt{ACMode}) \mathit{Disp}(\mathtt{OnCloud})$ — while evaluating the body of the function ftsearch the dispatching mechanism selected the layer ACMode and then OnCloud;

$\eta'_5 = \eta'_4 \mathit{send}_{\tau''}$ — the value v is sent over the bus;

$\eta'_6 = \eta'_5 ⦈_{\mathtt{ACMode}}\ ]_{\phi}$ — the layer ACMode is deactivated and then the enforcement of the policy ϕ is ended.

Of course, the value v read by the smartphone is the same one sent by the cloud over the bus.

3.2.3 Validity of a History

Here, we introduce the notion of validity of a history. The framing construct allows us to enforce a security policy on a expression. Security framings can be nested, so during the evaluation there might be many policies whose scope has been entered but not exited yet. These policies are called *active policies*. Intuitively, a history is valid if it satisfies all the active policies.

Given a history η, we denote with $\eta^{-[]}$ the history purged of all security framing events, formally

$$\begin{aligned}
&\epsilon^{-[]} = \epsilon\\
&(\eta\, ev)^{-[]} = \eta^{-[]}\, ev \quad \text{if } ev \neq [_\phi,]_\phi\\
&(\eta\, ev)^{-[]} = \eta^{-[]} \quad \text{otherwise}
\end{aligned}$$

Given a history η, the multiset $ap(\eta)$ collects all the policies ϕ still active, and it is defined as follows:

$$\begin{aligned}
&ap(\epsilon) = \{\,\} && ap(\eta\, [_\phi) = ap(\eta) \cup \{\phi\}\\
&ap(\eta\, ev) = ap(\eta) \quad ev \neq [_\phi,]_\phi && ap(\eta\,]_\phi) = ap(\eta) \setminus \{\phi\}
\end{aligned}$$

The validity of a history η is inductively defined as follows, assuming the notion of safety $\eta \vDash \phi$ (see Definition 3.4.1):

Definition 3.2.1 (Validity)**.** Given a history η we say that is valid, $\models \eta$ in symbols, if

$$\begin{aligned}
&\vDash \epsilon && \text{if } \eta = \epsilon \quad \text{and}\\
&\vDash \eta' ev && \text{if } \eta = \eta' ev \text{ and } \vDash \eta' \text{ and } (\eta' ev)^{-[]} \vDash \phi \text{ for all } \phi \in ap(\eta' ev)
\end{aligned}$$

Example 3.2.2. Returning to our example, consider the histories η'_1, η'_2, η'_3, η'_4, η'_5 and η'_6 generated by the reduction of the function `serve` in Example 3.2.1. As we said in Section 3.1, the policy ϕ forbids writing on the library, i.e. the action `write(library)`. Since this action is never carried out, all the histories are valid.

If a history is valid, also its prefixes are valid, i.e. validity is a prefix-closed property, as stated by the following property [BDFZ09]:

Property 3.2.1. *If a history η is valid, then each prefix of η is valid too.*

Proof. It is straightforward by applying Definition 3.2.1. □

Example 3.2.3. As a further example, consider a policy ϕ protecting a resource r and amounting to *no* $read(r)$ *after* $write(r)$. The history $\eta_0 = write(r) \cdot [_\phi \cdot read(r) \cdot]_\phi$ is not valid because $write(r) \cdot read(r) \nvDash \phi$. On the other hand, the history $\eta_1 = [_\phi \cdot read(r) \cdot]_\phi \cdot write(r)$ is valid because both $read(r) \vDash \phi$ and $read(r) \cdot write(r) \vDash \phi$.

The semantics of ContextML (in particular the rules for framing) ensures that the histories generated at runtime are all valid:

Property 3.2.2. *If $C \vdash \epsilon, e \to^* \eta', e'$, then η' is valid.*

Proof. If $\eta = \epsilon$ and $e' = e$ the proof is trivial. Otherwise we proceed by contradiction. Assume that η' is not valid and denote with Φ the set of policies occurring in policy framings of e. This mean that there exist e'' and η'' such that $C \vdash \epsilon, e \to^* \eta'', e'' \to \eta', e'$ and that $\eta''^{-[]} \not\models \phi$ for some $\phi \in \Phi$. Thus the computation gets stuck because no rule for framing applies for η'' and e'' (contradiction). □

3.3 Type and Effect System

We now associate a ContextML expression with a type and an abstraction, called *history expression*. The model checking procedure exploits history expressions to verify that programs respect security policies and the protocol. Firstly, we introduce history expressions and then our type and effect system.

3.3.1 History Expressions

History expressions [SSVH08, BDFZ09, BDF09] are a simple process algebra providing an abstraction over the set of histories that a program may generate. Here, history expressions approximate activating or deactivating layers, enforcing policies and sending or receiving messages. We adopt the same notion of [BDF09], though we consider histories with a different set of events, namely ev.

The syntax of history expressions is:

$$\begin{array}{lllcl}
H & ::= \epsilon & & \text{empty} & \\
 & \mid ev & & \text{events} & \text{e.g. } \alpha(r),\ send_\tau,\ \llparenthesis_L \\
 & \mid H_1 + H_2 & & \text{non-deterministic sum} & \\
 & \mid H_1 \cdot H_2 & & \text{sequence} & \\
 & \mid h & & \text{recursion variable} & \\
 & \mid \mu h.H & & \text{recursion} & \\
 & \mid \phi[H] & & \text{security framing} & \text{abbreviation for } [_\phi \cdot H \cdot]_\phi
\end{array}$$

The empty history expression ϵ abstracts programs that perform no relevant action; the "atomic" history expression ev indicates that an interesting event may occur during evaluation such as carrying out an action over a resource, the activation and deactivation of a layer, etc.; 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 application; $\mu h.H$ abstracts possibly recursive function with recursion variable h; the safety framing $\phi[H]$ represents the enforcement of the policy ϕ on H.

The behaviour of a history expression H is formalized by the labelled transition system inductively defined in Figure 3.6. Configurations have the form $H \xrightarrow{w} H'$ where $w \in (ev \cup \{\epsilon\})$ meaning that H reduces to H' carrying out the action w.

We briefly comment on the rules of the transition system. The "atomic" history expression ev reduces to ϵ yielding itself as label; the recursion $\mu h.H$ reduces to the body H substituting $\mu h.H$ for the recursion variable h (unfolding) and yielding ϵ; the

$$\frac{}{ev \xrightarrow{ev} \epsilon} \qquad \frac{}{\mu h.H \xrightarrow{\epsilon} H\{\mu h.H/h\}} \qquad \frac{H_1 \xrightarrow{ev} H_1'}{H_1 \cdot H_2 \xrightarrow{ev} H_1' \cdot H_2} \qquad \frac{}{\epsilon \cdot H_2 \xrightarrow{\epsilon} H_2}$$

$$\frac{H_1 \xrightarrow{ev} H_1'}{H_1 + H_2 \xrightarrow{ev} H_1'} \qquad \frac{H_2 \xrightarrow{ev} H_2'}{H_1 + H_2 \xrightarrow{ev} H_2'}$$

Figure 3.6: Transition system of History Expressions.

rules for sequence $H_1 \cdot H_2$ reduce H_1 up to obtaining ϵ and then the overall expression reduces to H_2; the sum $H_1 + H_2$ reduces to the history expression resulted from the reduction of one of its subexpressions (non-deterministically chosen). Note that there is no rule for the $\phi[H]$ because it is an abbreviation for $[_\phi \cdot H \cdot]_\phi$, then it is handled by the rules for events.

By exploiting the transition system in Figure 3.6 we define the semantics of a history expression H as a prefix closed set of histories. Intuitively, the semantics is the language of strings (prefix closed) obtained reducing H through the transition system in all possible way (the alphabet of the language is the set of the events ev). Formally,

Definition 3.3.1 (Semantics of History Expressions)**.** Let H be a closed history expression, we define its semantics $\llbracket H \rrbracket$ to be the set of histories $\eta = w_1 \ldots w_n$ $(w_i \in ev \cup \{\epsilon\}, 0 \leq i \leq n)$ such that $\exists H'.\ H \xrightarrow{w_1} \cdots \xrightarrow{w_n} H'$.

Note that the empty history expression ϵ plays also the role of the empty string, so in the following we assume that it is the identity element of the concatenation "$\cdot$", i.e. $\epsilon \cdot \eta = \eta$. We can easily extend the notion of history validity introduce in Definition 3.2.1 to history expression: a history expression H is valid, if $\models \eta$ for all $\eta \in \llbracket H \rrbracket$.

From [BDFZ09] we inherit the following equational theory on history expressions:

Definition 3.3.2 (Equational Theory of History Expression)**.**

$$\begin{array}{ll}
H + H \equiv H \equiv \epsilon \cdot H \equiv H \equiv H \cdot \epsilon & H_1 + H_2 \equiv H_2 + H_1 \\
H_1 \cdot (H_2 \cdot H_3) \equiv (H_1 \cdot H_2) \cdot H_3 & H_1 + (H_2 + H_3) \equiv (H_1 + H_2) + H_3 \\
H_1 \cdot (H_2 + H_3) \equiv (H_1 \cdot H_2) + (H_1 \cdot H_3) & (H_1 + H_2) \cdot H_3 \equiv (H_1 \cdot H_3) + (H_2 \cdot H_3)
\end{array}$$

Example 3.3.1. Back to our motivating example in Section 3.1, assume that H is the history expression over-approximating the behaviour of the function read from the bus at line 3 of Figure 3.2b. Then, the history expression for the function `serve` is

$$H_s = \mu h.receive_\tau \cdot receive_{\tau'} \cdot \phi\, [(\!|_{\texttt{ACMode}} \cdot H \cdot send_{\tau''} |\!)_{\texttt{ACMode}}] \cdot h.$$

Assuming that H reduces to ϵ yielding the labels $Disp(\texttt{ACMode})$ and $Disp(\texttt{OnCloud})$, a possible reduction of H_s is

$$\mu h.receive_\tau \cdot receive_{\tau'} \cdot \phi\, [(\!|_{\texttt{ACMode}} \cdot H \cdot send_{\tau''} \cdot |\!)_{\texttt{ACMode}}] \cdot h \xrightarrow{\epsilon}$$

$$\begin{aligned}
receive_{\tau} \cdot receive_{\tau'} \cdot \phi\,[\llparenthesis_{\texttt{ACMode}} \cdot H \cdot send_{\tau''} \cdot \rrparenthesis_{\texttt{ACMode}}] \cdot H_s &\xrightarrow{receive_{\tau}} \\
receive_{\tau'} \cdot \phi\,[\llparenthesis_{\texttt{ACMode}} \cdot H \cdot send_{\tau''} \cdot \rrparenthesis_{\texttt{ACMode}}] \cdot H_s &\xrightarrow{receive_{\tau'}} \\
\left[_{\phi} \cdot \llparenthesis_{\texttt{ACMode}} \cdot H \cdot send_{\tau''} \cdot \rrparenthesis_{\texttt{ACMode}} \cdot \right]_{\phi} \cdot H_s &\xrightarrow{[_{\phi}} \\
\llparenthesis_{\texttt{ACMode}} \cdot H \cdot send_{\tau''} \cdot \rrparenthesis_{\texttt{ACMode}} \cdot]_{\phi} \cdot H_s &\xrightarrow{\llparenthesis_{\texttt{ACMode}}} \\
H \cdot send_{\tau''} \cdot \rrparenthesis_{\texttt{ACMode}} \cdot]_{\phi} \cdot H_s &\xrightarrow{Disp(\texttt{ACMode}) \cdot Disp(\texttt{OnCloud})}{}^{*} \\
send_{\tau''} \cdot \rrparenthesis_{\texttt{ACMode}} \cdot]_{\phi} \cdot H_s &\xrightarrow{send_{\tau''}} \\
\rrparenthesis_{\texttt{ACMode}} \cdot]_{\phi} \cdot H_s &\xrightarrow{\rrparenthesis_{\texttt{ACMode}}} \\
]_{\phi} \cdot H_s &\xrightarrow{]_{\phi}} \\
H_s &
\end{aligned}$$

The sequence $receive_{\tau} receive_{\tau'} [_{\phi} \llparenthesis_{\texttt{ACMode}} Disp(\texttt{ACMode}) Disp(\texttt{OnCloud}) send_{\tau''} \rrparenthesis_{\texttt{ACMode}}]_{\phi}$ is a history (call it η_s). By Definition 3.3.1 the semantics of H_s is

$$[\![H_s]\!] = \{ (receive_{\tau} receive_{\tau'} [_{\phi} \llparenthesis_{\texttt{ACMode}} l_c send_{\tau''} \rrparenthesis_{\texttt{ACMode}}]_{\phi})^{\star} \mid l_c \in [\![H]\!] \}.$$

Obviously, by taking $l_c = Disp(\texttt{ACMode}) Disp(\texttt{OnCloud})$ we have that $\eta_s \in [\![H_s]\!]$. Note also that the evaluation of H_s above abstracts the function `serve` in Example 3.2.1: indeed, it is easy to observe that $\eta'_4 = \eta' \eta_s$, where η' is the history with which the evaluation of the function `serve` starts.

3.3.2 Typing rules

Here, we define the typing rules defining the type and effect system of ContextML.

Our typing judgements have the form $\langle \Gamma; C \rangle \vdash e : \tau \triangleright H$. This means that in "in the type environment Γ and in the context C the expression e has type τ and effect H".

The typing environment Γ is defined as usual to map identifiers to their types. Types are constants, layers and functions:

$$\tau_c \in \{\texttt{int}, \texttt{bool}, \texttt{unit}, \dots\} \qquad \sigma \in \wp(\mathsf{LayerNames}) \qquad \mathbb{P} \in \wp\left(\wp(\mathsf{LayerNames})\right)$$

$$\tau, \tau_1, \tau' ::= \tau_c \mid ly_{\sigma} \mid \tau_1 \xrightarrow{\mathbb{P} | H} \tau_2$$

As we did in Section 2.2.2, we annotate types with sets of layer names σ for analysis reason. In ly_{σ}, σ over-approximates the set of layers an expression can be reduced to. In $\tau_1 \xrightarrow{\mathbb{P}|H} \tau_2$, the set $\mathbb{P}$ contains the *preconditions* v. Each $v \in \mathbb{P}$ over-approximates the set of layers that must occur in the context to apply the function. The history expression H is the latent effect, i.e. the sequence of events generated while evaluating the function.

We now introduce the ordering $\sqsubseteq_{\mathbb{P}}$ and $\sqsubseteq_H$ on the set of preconditions and history expressions, respectively (often we omit the indices when unambiguous). The ordering on the set of preconditions and history expressions are defined as follows:

$$\mathbb{P} \sqsubseteq_{\mathbb{P}} \mathbb{P}' \text{ iff } \forall v \in \mathbb{P} \,.\, \exists v' \in \mathbb{P}' \,.\, v' \subseteq v$$

$$\textsc{Sref}\ \frac{}{\tau \leq \tau} \qquad \textsc{Sfun}\ \frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2' \quad \mathbb{P} \sqsubseteq \mathbb{P}' \quad H \sqsubseteq H'}{\tau_1 \xrightarrow{\mathbb{P}|H} \tau_2 \leq \tau_1' \xrightarrow{\mathbb{P}'|H'} \tau_2'} \qquad \textsc{Sly}\ \frac{\sigma \subseteq \sigma'}{ly_\sigma \leq ly_{\sigma'}}$$

$$\textsc{Tsub}\ \frac{\langle \Gamma; C\rangle \vdash e : \tau' \triangleright H' \quad \tau' \leq \tau \quad H' \sqsubseteq H}{\langle \Gamma; C\rangle \vdash e : \tau \triangleright H}$$

Figure 3.7: Subtyping rules

$$H_1 \sqsubseteq_H H_2 \text{ iff } H_2 \equiv H_1 + H_3 \text{ for some } H_3$$

Note that the partial ordering $H_1 \sqsubseteq H_2$ is defined over the quotient induced by the (semantic-preserving) equational theory introduced in Definition 3.3.2. Clearly, it holds that $H_1 \sqsubseteq H_2$ implies $[\![H_1]\!] \subseteq [\![H_2]\!]$, but the theory is not complete i.e. $[\![H_1]\!] \subseteq [\![H_2]\!]$ does not imply $H_1 \sqsubseteq H_2$ (see [BDFZ09]). Intuitively, the partial order $H_1 \sqsubseteq H_2$ means that the abstraction represented by H_2 is less precise than the one by H_1.

Since our types are annotated and we have effects, we need rules for subtyping ($\tau_1 \leq \tau_2$) and for subeffecting ($H \sqsubseteq H'$)displayed in Figure 3.7. The rule Sref states that the subtyping relation is reflexive. The rule Sly says that a layer type ly_σ is a subtype of $ly_{\sigma'}$ whenever the annotation σ is a subset of σ'. The rule Sfun defines subtyping for functional types. As usual, it is contravariant in τ_1 but covariant in $\mathbb{P}, \tau_2$ and H. By the Tsub rule, we can always enlarge types and effects.

Figure 3.8 shows the typing rules of our type and effect system. Most of them extends the one shown in Figure 2.2 to deal with the effects. For instance, the rules Tconst, Tvar and Tly are the same apart from the fact that they yield the empty effect ϵ. So we only comment in detail on the rules which changed and the ones for the new (and COP) constructs. In the rule Tfun we guess a set of preconditions $\mathbb{P}$, a type for the bound variable x and for the function f. For all preconditions $v \in \mathbb{P}$ we also guess a context C', under which we determine the type of the body e. We impose that the precondition v contains all the layers in C', in symbols $v \subseteq |C'|$, where $|C'|$ denotes the set of active layers in the context C' as done in Section 2.2.2. Implicitly, we require that the guessed type for f, as well as its latent effect H, match the ones of the resulting function. Additionally, we require that the resulting type is annotated with $\mathbb{P}$. The application gets a type (rule Tapp) if there exists a precondition $v \in \mathbb{P}$ such that it is satisfied in the current context C. A context satisfies the precondition v whenever it contains all the layers in v, in symbols $|C'| \subseteq v$. The effect is obtained by concatenating the ones of e_2 and e_1 and the latent effect H. The rule Twith is almost unchanged except for the effect. It is the union of the possible effects resulting from evaluating the body. This evaluation is carried on the different contexts obtained by extending C with one of the layers in σ. The special events $(\!|_L$ and $|\!)_L$ express the scope of this layer activation. The rule Twithout is similar to Twith, but instead removes the layers in σ and use $\{_L$ and $\}_L$ to delimit layer hiding. The rule Tlexp is extended in such a way that its effect is the

TVar

$$\frac{\Gamma(x) = \tau}{\langle \Gamma; C \rangle \vdash x : \tau \triangleright \epsilon}$$

Tconst

$$\frac{}{\langle \Gamma; C \rangle \vdash c : \tau_c \triangleright \epsilon}$$

Tly

$$\frac{}{\langle \Gamma; C \rangle \vdash L : ly_{\{L\}} \triangleright \epsilon}$$

Tfun

$$\frac{\forall v \in \mathbb{P}. \quad \langle \Gamma, x : \tau_1, f : \tau_1 \xrightarrow{\mathbb{P}|H} \tau_2; C' \rangle \vdash e : \tau_2 \triangleright H \qquad \left|C'\right| \subseteq v}{\langle \Gamma; C \rangle \vdash \textbf{fun}\ f\ x \Rightarrow e : \tau_1 \xrightarrow{\mathbb{P}|H} \tau_2 \triangleright \epsilon}$$

Tapp

$$\frac{\langle \Gamma; C \rangle \vdash e_1 : \tau_1 \xrightarrow{\mathbb{P}|H} \tau_2 \triangleright H_1 \qquad \langle \Gamma; C \rangle \vdash e_2 : \tau_1 \triangleright H_2 \qquad \exists v \in \mathbb{P}. v \subseteq |C|}{\langle \Gamma; C \rangle \vdash e_1 e_2 : \tau_2 \triangleright H_2 \cdot H_1 \cdot H}$$

Tif

$$\frac{\langle \Gamma; C \rangle \vdash e_0 : \texttt{bool} \triangleright H \qquad \langle \Gamma; C \rangle \vdash e_1 : \tau \triangleright H_1 \qquad \langle \Gamma; C \rangle \vdash e_2 : \tau \triangleright H_2}{\langle \Gamma; C \rangle \vdash \textbf{if}\ \ e_0\ \ \textbf{then}\ e_1\ \textbf{else}\ e_2 : \tau \triangleright H \cdot (H_1 + H_2)}$$

Tlet

$$\frac{\langle \Gamma; C \rangle \vdash e_1 : \tau_1 \triangleright H_1 \qquad \langle \Gamma, x : \tau_1, C \rangle \vdash e_2 : \tau_2 \triangleright H_2}{\langle \Gamma; C \rangle \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2 \triangleright H_1 \cdot H_2}$$

Top

$$\frac{\langle \Gamma; C \rangle \vdash e_1 : \tau_c \triangleright H_1 \qquad \langle \Gamma; C \rangle \vdash e_2 : \tau_c \triangleright H_2}{\langle \Gamma; C \rangle \vdash e_1\ \textbf{op}\ e_2 : \tau_c \triangleright H_1 \cdot H_2}$$

Twith

$$\frac{\langle \Gamma; C \rangle \vdash e_1 : ly_{\{L_1, \ldots, L_n\}} \triangleright H' \qquad \forall L_i \in \{L_1, \ldots, L_n\}. \langle \Gamma; L_i :: C \rangle \vdash e_2 : \tau \triangleright H_i}{\langle \Gamma; C \rangle \vdash \textbf{with}(e_1)\ \textbf{in}\ e_2 : \tau \triangleright H' \cdot \sum_{L_i} \llparenthesis_{L_i} \cdot H_i \cdot \rrparenthesis_{L_i}}$$

Twithout

$$\frac{\langle \Gamma; C \rangle \vdash e_1 : ly_{\{L_1, \ldots, L_n\}} \triangleright H' \qquad \forall L_i \in \{L_1, \ldots, L_n\}. \langle \Gamma; C - L_i \rangle \vdash e_2 : \tau \triangleright H_i}{\langle \Gamma; C \rangle \vdash \textbf{without}(e_1)\ \textbf{in}\ e_2 : \tau \triangleright H' \cdot \sum_{L_i} \{_{L_i} \cdot H_i \cdot \}_{L_i}}$$

Tlexp

$$\frac{\forall i. \langle \Gamma; C \rangle \vdash e_i : \tau \triangleright H_i \qquad L_1 \in |C| \vee \cdots \vee L_n \in |C|}{\langle \Gamma; C \rangle \vdash L_1.e_1, \ldots, L_n.e_n : \tau \triangleright \sum_{L_i \in \{L_1, \ldots, L_n\}} \mathsf{Disp}(L_i) \cdot H_i}$$

Talpha

$$\frac{}{\langle \Gamma; C \rangle \vdash \alpha(a) : \texttt{unit} \triangleright \alpha(a)}$$

Tphi

$$\frac{\langle \Gamma; C \rangle \vdash e : \tau \triangleright H}{\langle \Gamma; C \rangle \vdash \phi[e] : \tau \triangleright [_{\phi} \cdot H \cdot]_{\phi}}$$

Trec

$$\frac{}{\langle \Gamma; C \rangle \vdash \textbf{receive}_{\tau} : \tau \triangleright \mathit{receive}_{\tau}}$$

Tsend

$$\frac{\langle \Gamma; C \rangle \vdash e : \tau \triangleright H \qquad H' = H \cdot \mathit{send}_{\tau}}{\langle \Gamma; C \rangle \vdash \textbf{send}_{\tau}(e) : \texttt{unit} \triangleright H'}$$

Figure 3.8: Typing rules

$$\textsc{Tbwith}\quad \frac{\langle \Gamma; L :: C\rangle \vdash e_2 : \tau \rhd H}{\langle \Gamma; C\rangle \vdash \textbf{with}(\overline{L})\ \textbf{in}\ e_2 : \tau \rhd H \cdot \rrparenthesis_L} \qquad \textsc{Tbwithout}\quad \frac{\langle \Gamma; C - L\rangle \vdash e_2 : \tau \rhd H}{\langle \Gamma; C\rangle \vdash \textbf{without}(\overline{L})\ \textbf{in}\ e_2 : \tau \rhd H \cdot \}_L}$$

$$\textsc{Tbphi}\quad \frac{\langle \Gamma; C\rangle \vdash e : \tau \rhd H}{\langle \Gamma; C\rangle \vdash \bar{\phi}[e] : \tau \rhd H \cdot]_{\phi}}$$

Figure 3.9: Typing rules for auxiliary syntactic configurations

sum of the effects H_i of sub-expressions, each of which is preceded by $Disp(L_i)$ event. The rule Talpha gives an expression $\alpha(r)$ the type `unit` and effect $\alpha(r)$. By the rule Tphi the expression $\phi[e]$ has the same type of e and as effect the one of e enclosed between the events $[_{\phi}$ and $]_{\phi}$. The expression $\textbf{send}_{\tau}(e)$ has type `unit` and its effect is that of e extended with event $send_{\tau}$. The expression $\textbf{receive}_{\tau}$ has type τ and its effect is the event $receive_{\tau}$. Note that the rules establish the correspondence between the type declared in the syntax and the checked type of the value sent/received. An additional check is however needed and will be carried on also taking care of the interaction protocol (Section 3.4).

For technical reasons, we need the rules shown Figure 3.9 for dealing with the auxiliary syntactic configurations for **with**, **without** and $\phi[e]$. The rule Tbwith requires that the subexpression e_2 gets a type τ and an effect H in the context C enlarged by the layer L. If this happens, the overall expression has type τ and effect obtained by appending the event $\rrparenthesis_L$ to H. Note that when we reach this configuration during the evaluation we already know which the layer L is activated and its activation was recorded in the history. The rule Tbwithout is the similar to Tbwith, but it adds the event $\}_L$ to the effect computed for the subexpression e_2. By the rule Tbphi the $\bar{\phi}[e]$ has the same type of e and its effect is obtained by concatenating the one of e with the event $]_{\phi}$.

Example 3.3.2. In Figure 3.10 we show an example of type derivation tree for a part of the body of the function `serve`. We assume that $\tau = ly_{\{\texttt{ACMode}\}}$, $\tau' = \texttt{unit} \xrightarrow{H} \tau''$ and that the application $g\ ()$ gets the type τ'' and the effect H. Furthermore, in the type derivation tree we use the following type environments, contexts and history expressions:

$$\begin{array}{rlrl}
\Gamma_1 = & \Gamma, \texttt{lyr} : \tau & H_1 = & receive_{\tau} \cdot receive_{\tau'} \cdot \phi[\llparenthesis_{\texttt{ACMode}} H \cdot send_{\tau''} \rrparenthesis_{\texttt{ACMode}}] \\
\Gamma_2 = & \Gamma, \texttt{lyr} : \tau, \texttt{g} : \tau' & H_2 = & receive_{\tau'} \cdot \phi[\llparenthesis_{\texttt{ACMode}} H \cdot send_{\tau''} \rrparenthesis_{\texttt{ACMode}}] \\
\Gamma_3 = & \Gamma, \texttt{lyr} : \tau, \texttt{g} : \tau', \texttt{res} : \tau'' & H_3 = & \phi[\llparenthesis_{\texttt{ACMode}} H \cdot send_{\tau''} \rrparenthesis_{\texttt{ACMode}}] \\
C_1 = & \texttt{ACMode} :: C & H_4 = & \llparenthesis_{\texttt{ACMode}} H \cdot send_{\tau''} \rrparenthesis_{\texttt{ACMode}}
\end{array}$$

It is easy to see that the resulting effect H_1 is a subterm of the history expression H_s introduced in Example 3.3.1 as effect of the function `serve`, i.e. $H_s = \mu h. H_1 \cdot h$.

Our type and effect system ensures us that

- functions are correctly used;

$$
\dfrac{\langle\Gamma;C\rangle \vdash \mathbf{receive}_{\tau} : \tau \triangleright \mathit{receive}_{\tau} \qquad
\dfrac{\langle\Gamma_1;C\rangle \vdash \mathbf{receive}_{\tau'} : \tau' \triangleright \mathit{receive}_{\tau'} \qquad
\dfrac{\dfrac{\langle\Gamma_2;C\rangle \vdash \mathtt{lyr} : \tau \triangleright \epsilon \qquad
\dfrac{\langle\Gamma_2;C_1\rangle \vdash g\,() : \tau'' \triangleright H \qquad
\dfrac{\langle\Gamma_3;C_1\rangle \vdash \mathtt{res} : \tau'' \triangleright \epsilon}{\langle\Gamma_3;C_1\rangle \vdash \mathbf{send}_{\tau''}(\mathtt{res}) : \mathtt{unit} \triangleright \mathit{send}_{\tau''}}}
{\langle\Gamma_2;C_1\rangle \vdash \left(\begin{array}{c}\textbf{let}\ \mathtt{res} = \mathtt{g}\ ()\ \textbf{in}\\ \mathbf{send}_{\tau''}(\mathtt{res})\end{array}\right) : \mathtt{unit} \triangleright H \cdot \mathit{send}_{\tau''}}}
{\langle\Gamma_2;C\rangle \vdash \left(\begin{array}{l}\textbf{with}(\mathtt{lyr})\ \textbf{in}\\ \quad \textbf{let}\ \mathtt{res} = \mathtt{g}\ ()\ \textbf{in}\\ \quad\quad \mathbf{send}_{\tau''}(\mathtt{res})\end{array}\right) : \mathtt{unit} \triangleright H_4}}
{\langle\Gamma_2;C\rangle \vdash \left(\begin{array}{l}\phi[\textbf{with}(\mathtt{lyr})\ \textbf{in}\\ \quad \textbf{let}\ \mathtt{res} = \mathtt{g}\ ()\ \textbf{in}\\ \quad\quad \mathbf{send}_{\tau''}(\mathtt{res})]\end{array}\right) : \mathtt{unit} \triangleright H_3}}
{\langle\Gamma_1;C\rangle \vdash \left(\begin{array}{l}\textbf{let}\ \mathtt{g} = \mathbf{receive}_{\tau'}\ \textbf{in}\\ \quad \phi[\textbf{with}(\mathtt{lyr})\ \textbf{in}\\ \quad\quad \textbf{let}\ \mathtt{res} = \mathtt{g}\ ()\ \textbf{in}\\ \quad\quad\quad \mathbf{send}_{\tau''}(\mathtt{res})]\end{array}\right) : \mathtt{unit} \triangleright H_2}}
{\langle\Gamma;C\rangle \vdash \left(\begin{array}{l}\textbf{let}\ \mathtt{lyr} = \mathbf{receive}_{\tau}\ \textbf{in}\\ \textbf{let}\ \mathtt{g} = \mathbf{receive}_{\tau'}\ \textbf{in}\\ \quad \phi[\textbf{with}(\mathtt{lyr})\ \textbf{in}\\ \quad\quad \textbf{let}\ \mathtt{res} = \mathtt{g}\ ()\ \textbf{in}\\ \quad\quad\quad \mathbf{send}_{\tau''}(\mathtt{res})]\end{array}\right) : \mathtt{unit} \triangleright H_1}
$$

Figure 3.10: The type derivation of the body of function `serve`

- the evaluation of a layered expression never gets stuck;
- the computed effect is a safe over-approximation of the histories may arise at run-time.

The following two lemmata guarantee that our type and effect system is sound with respect to the operational semantics. The first assures that types are preserved during the evaluation. In the statement we denote with ηH the history expression obtained concatenating the events of η with H.

Lemma 3.3.1 (Preservation). *Let e be a closed expression, if* $\langle\Gamma;C\rangle \vdash e : \tau \triangleright H$ *and* $C \vdash \eta, e \to \eta', e'$, *then* $\langle\Gamma;C\rangle \vdash e' : \tau \triangleright H'$ *with* $\eta H \sqsupseteq \eta' H'$.

The second lemma claims that under the assumptions of the safety and compliance a well-typed expression never gets stuck unless it is a value. In the statement we denote with $C \vdash \eta, e \not\to$ the fact that e is stuck.

Lemma 3.3.2 (Progress). *Let e be a closed expression such that* $\langle\Gamma;C\rangle \vdash e : \tau \triangleright H$. *If* $C \vdash \eta, e \not\to$ *and* ηH *is valid and it is compliant with the communication protocol, then e is a value.*

From the two lemmata above we prove the following proposition. It states that the history expression obtained as effect of an expression e over-approximates the set of histories that may actually be generated during the execution of e.

Proposition 3.3.3 (Over-approximation). *If* $\langle\Gamma;C\rangle \vdash e : \tau \triangleright H$ *and* $C \vdash \epsilon, e \to^* \eta, e'$, *then* $\eta \in [\![H]\!]$.

The following theorem guarantees that a well-typed expression never gets stuck, provided that its H is valid and that it adheres to the communication protocol.

Theorem 3.3.4. *If $\langle \emptyset; C\rangle \vdash e : \tau \triangleright H$ and H is valid with balanced policy framings and it is compliance with the communication protocol, then either the computation terminates by yielding a value ($C \vdash \epsilon, e \rightarrow^* \eta', v$) or it diverges, but it never gets stuck.*

3.3.3 Soundness of the Type and Effect System

We collect below the technical development to prove Lemma 3.3.1, Lemma 3.3.2, Proposition 3.3.3 and Theorem 3.3.4. The proofs require some auxiliary results stated below. We refer to the definition of the partial order $\sqsubseteq$ we will use $=$ instead of $\equiv$.

The following definition extends the notion of substitution introduced in Definition 2.3.1.

Definition 3.3.3 (Substitution)**.** Given the expressions e, e' and the identifier x we define $e\{e'/x\}$ as following:

$$\begin{aligned}
n\{e'/x\} &= n\\
L\{e'/x\} &= L\\
()\{e'/x\} &= ()\\
(\mathbf{fun}\, f x' \Rightarrow e)\{e'/x\} &= \mathbf{fun}\, f x \Rightarrow e\{e'/x\} \quad \text{if } f \neq x \wedge x \neq x' \wedge f, x' \notin fv(e')\\
x\{e'/x\} &= e'\\
x'\{e'/x\} &= x' \quad \text{if } x' \neq x\\
(e_1\, e_2)\{e'/x\} &= e_1\{e'/x\}e_2\{e'/x\}\\
(e_1\, op\, e_2)\{e'/x\} &= e_1\{e'/x\}\, op\, e_2\{e'/x\}\\
(\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2)\{e'/x\} &= \mathbf{if}\ e_0\{e'/x\}\ \mathbf{then}\ e_1\{e'/x\}\ \mathbf{else}\ e_2\{e'/x\}\\
(\mathbf{with}(e_1)\ \mathbf{in}\ e_2)\{e'/x\} &= \mathbf{with}(e_1\{e'/x\})\ \mathbf{in}\ e_2\{e'/x\}\\
(\mathbf{without}(e_1)\ \mathbf{in}\ e_2)\{e'/x\} &= \mathbf{without}(e_1\{e'/x\})\ \mathbf{in}\ e_2\{e'/x\}\\
(\phi[e])\{e'/x\} &= \phi[e\{e'/x\}]\\
\alpha(r)\{e'/x\} &= \alpha(r)\\
(\mathbf{send}_\tau(e))\{e'/x\} &= \mathbf{send}_\tau(e\{e'/x\})\\
(\mathbf{receive}_\tau)\{e'/x\} &= \mathbf{receive}_\tau\\
(L_1.e_1, \ldots, L_n.e_n)\{e'/x\} &= L_1.e_1\{e'/x\}, \ldots, L_n.e_n\{e'/x\}
\end{aligned}$$

The following Lemma guarantees us that the arrangement of the bindings within the type environment does not influence the typing (used in the proof of Lemma 3.3.7).

Lemma 3.3.5 (Permutation)**.** *If $\langle \Gamma; C\rangle \vdash e : \tau \triangleright H$ and Γ' is a permutation of Γ then $\langle \Gamma'; C\rangle \vdash e : \tau \triangleright H$.*

Proof. Similar to the one of Lemma 2.3.1. □

The Weakening Lemma below states that inserting new bindings in the type environment for new variables does not affect the typing (useful in the proof of Lemma 3.3.7).

Lemma 3.3.6 (Weakening). *If* $\langle \Gamma; C\rangle \vdash e : \tau \triangleright H$ *and* $y \notin dom(\Gamma)$ *then* $\langle \Gamma, y : \tau'; C\rangle \vdash e : \tau \triangleright H$.

Proof. Similar to the one of Lemma 2.3.2. □

By the Inclusion Lemma below we can always enlarge the type environment and the context where an expression type-checks (used to prove Lemma 3.3.8).

Lemma 3.3.7 (Inclusion).

1. *If* $\langle \Gamma; C\rangle \vdash e : \tau \triangleright H$, *then* $\forall \Gamma'$ *s.t.* $\Gamma \subset \Gamma'$ *it holds* $\langle \Gamma'; C\rangle \vdash e : \tau \triangleright H$
2. *If* $\langle \Gamma; C\rangle \vdash e : \tau \triangleright H$, *then* $\forall C'$ *s.t.* $|C| \subset |C'|$ *it holds* $\langle \Gamma; C'\rangle \vdash e : \tau \triangleright H$

Proof. Similar to the one of Lemma 2.3.3. □

The following Lemma is required to prove Lemma 3.3.9.

Lemma 3.3.8 (Decomposition). $\langle \Gamma; C\rangle \vdash \boldsymbol{fun}\ f x \Rightarrow e : \tau \xrightarrow{\mathbb{P}|H} \tau' \triangleright H'$ *and* $\exists v \in \mathbb{P}.\, |C| \supseteq v$ *implies* $\langle \Gamma, x : \tau, f : \tau \xrightarrow{\mathbb{P}|H} \tau'; C\rangle \vdash e : \tau' \triangleright H$.

Proof. Sketch: By the rule Tfun we know that for all $v \in \mathbb{P}$ there exists a guessed context C' such that $|C'| \subseteq v$ and $\langle \Gamma, x : \tau, f : \tau \xrightarrow{\mathbb{P}|H} \tau'; C'\rangle \vdash e : \tau' \triangleright H$. Since $|C'| \subseteq |C|$, thesis follows by Lemma 3.3.7 (2). □

The Substitution Lemma states that types are preserved under substitution (we use it in the proof of Lemma 3.3.1).

Lemma 3.3.9 (Substitution Lemma). *Let* $\langle \Gamma, x : \tau'; C\rangle \vdash e : \tau \triangleright H$ *and* $\langle \Gamma; C\rangle \vdash v : \tau' \triangleright \epsilon$ *then* $\langle \Gamma; C\rangle \vdash e\{v/x\} : \tau \triangleright H$

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

- cases Tconst, Tly, Talpha and Trec
  Since by Definition 3.3.3 holds for these cases that $e\{v/x\} = e$ the property vacuously holds.

- case Tphi
  By the precondition of the rule Tphi $\langle \Gamma, x : t'; C\rangle \vdash e_1 : \tau \triangleright H'$ holds where $H = [_{\phi} H']_{\phi}$. By Definition 3.3.3 we know that $\phi[e_1]\{v/x\} = \phi[e_1\{v/x\}]$, hence by the applying inductive hypothesis $\langle \Gamma; C\rangle \vdash e_1\{v/x\} : \tau \triangleright H'$. So by the rule Tphi and by Definition 3.3.3 we can conclude that $\langle \Gamma; C\rangle \vdash \phi[e_1]\{v/x\} : \tau \triangleright H$ holds.

- case Twith
  By the precondition of the rule Twith we know that $\langle \Gamma, x : \tau'; C\rangle \vdash e_1 : ly_{\{L_1,\ldots,L_n\}} \triangleright H'$ and $\forall L_i \in \{L_1, \ldots, L_n\}.\langle \Gamma, x : \tau'; L_i :: C\rangle \vdash e_2 : \tau \triangleright H_i$ hold where $H = H' \cdot \sum_{L_i} (\!| _{L_i} \cdot H_i \cdot |\!)_{L_i}$. By Definition 3.3.3 $(\mathbf{with}(e_1)\ \mathbf{in}\ e_2)\{v/x\} = \mathbf{with}(e_1\{v/x\})\ \mathbf{in}\ e_2\{v/x\}$. By the inductive hypothesis $\langle \Gamma, x : \tau'; C\rangle \vdash e_1\{v/x\} : ly_{\{L_1,\ldots,L_n\}} \triangleright H'$ and $\forall L_i \in \{L_1, \ldots, L_n\}.\langle \Gamma, x : \tau'; L_i :: C\rangle \vdash e_2\{v/x\} : \tau \triangleright H_i$ hold. So by the rule Twith and by Definition 3.3.3 we can conclude that $\langle \Gamma; C\rangle \vdash (\mathbf{with}(e_1)\ \mathbf{in}\ e_2)\{v/x\} : \tau \triangleright H$.

- case TWITHOUT
  Similar to the case for the rule TWITH.

- case TLEXP
  By the rule TLEXP we know that $\forall i \in \{L_1, \ldots, L_n\} \quad \langle \Gamma, x : \tau'; C \rangle \vdash e_i : \tau \triangleright H_i$, $L_1 \in |C| \vee \cdots \vee L_n \in |C|$ and $H = \sum_{L_i \in \{L_1,\ldots,L_n\}} \mathsf{Disp}(L_i) \cdot H_i$. By Definition 3.3.3 we know that $(L_1.e_1, \ldots, L_n.e_n)\{v/x\}$. By the inductive hypothesis we can state that $\forall i \in \{L_1, \ldots, L_n\}$ $\langle \Gamma; C \rangle \vdash e_i\{v/x\} : \tau \triangleright H_i$. Since $L_1 \in |C| \vee \cdots \vee L_n \in |C|$ by the rule TLEXP and by Definition 3.3.3 we can conclude that $\langle \Gamma; C \rangle \vdash (L_1.e_1, \ldots, L_n.e_n)\{v/x\} : \tau \triangleright H$ hold.

- case TSEND. By the rule TSEND we have that $\langle \Gamma, x : \tau'; C \rangle \vdash \mathbf{send}_\tau(e_1) : \tau \triangleright H'$ holds with $H = H' \cdot send_\tau$. By Definition 3.3.3 $\mathbf{send}_\tau(e_1)\{v/x\} = \mathbf{send}_\tau(e_1\{v/x\})$ and by inductive hypothesis that $\langle \Gamma; C \rangle \vdash e_1\{v/x\} : \tau \triangleright H'$. So by Definition 3.3.3 and by the rule TSEND we can conclude that $\langle \Gamma; C \rangle \vdash \mathbf{send}_\tau(e_1)\{v/x\} : \tau \triangleright H$.

- The other cases are standard and their proofs are similar to the one of Lemma 2.3.5. We do not show them.

□

The following property states that the order over history expressions is preserved by the concatenation:

Property 3.3.10. *Given the histories expressions H_1, H_2, H_1' and H_2' such that $H_1 \sqsubseteq H_1'$ and $H_2 \sqsubseteq H_2'$ then $H_1 \cdot H_2 \sqsubseteq H_1' \cdot H_2'$*

Proof. By the definition of $\sqsubseteq$ over the history expressions we know that $H_1' = H_1 + H_3$ for some H_3 and $H_2' = H_2 + H_4$ for some H_4. By exploiting the equational theory of Definition 3.3.2.

$$\begin{aligned} H_1' \cdot H_2' &= (H_1 + H_3) \cdot (H_2 + H_4) \\ &= (H_1 \cdot (H_2 + H_4)) + (H_3 \cdot (H_2 + H_4)) \\ &= H_1 \cdot H_2 + H_1 \cdot H_4 + H_3 \cdot H_2 + H_3 \cdot H_4 \end{aligned}$$

So we can conclude that $H_1 \cdot H_2 \sqsubseteq H_1' \cdot H_2'$. □

By the following lemma we can type-check values in every type environment and context (exploited in Lemma 3.3.1).

Lemma 3.3.11. *If $\langle \Gamma; C \rangle \vdash v : \tau \triangleright H$, with v a value, then for all C' we have that $\langle \Gamma; C' \rangle \vdash v : \tau \triangleright H$.*

Proof. Similar to the one of Lemma 2.3.6. □

The lemma below ensures us that we can always restrict the effect of the values to the empty one (used in Lemma 3.3.1).

Lemma 3.3.12. *If $\langle \Gamma; C \rangle \vdash v : \tau \triangleright H$ then $\langle \Gamma; C \rangle \vdash v : \tau \triangleright \epsilon$*

Proof. In the typing derivation for the judgement $\langle \Gamma; C \rangle \vdash v : \tau \triangleright H$ there is a subderivation with conclusion $\langle \Gamma; C \rangle \vdash v : \tau' \triangleright \epsilon$ for some τ'. This conclusion is obtained by applying one of typing rules for values. Since v is a value we can obtain $\langle \Gamma; C \rangle \vdash v : \tau \triangleright H$ 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 construct a derivation for the judgement $\langle \Gamma; C \rangle \vdash v : \tau \triangleright \epsilon$. □

Lemma 3.3.1 (Preservation). *Let e be a closed expression, if* $\langle \Gamma; C \rangle \vdash e : \tau \triangleright H$ *and* $C \vdash \eta, e \to \eta', e'$*, then* $\langle \Gamma; C \rangle \vdash e' : \tau \triangleright H'$ *with* $\eta H \sqsupseteq \eta' H'$.

Proof. By induction on the depth of the typing derivation, and then by cases on the last rule applied. Below for each case we report the premises of the corresponding rule.

- case Tconst or Tly or or Tvar Tfun
  In this case we know that e_s is a value (or a variable in the case of rule Tvar), then it cannot be the case $C \vdash \eta, e_s \to \eta', e_s'$ for any eta' and e_s', so the thesis vacuously holds.

- case Tsub $\quad \langle \Gamma; C \rangle \vdash e : \tau' \triangleright H' \quad H' \sqsubseteq H$
  If $C \vdash \eta, e \to \eta', e'$, then by inductive hypothesis and by the premises of the typing rule we have that $\langle \Gamma; C \rangle \vdash e' : \tau' \triangleright H''$ with $\eta H' \sqsupseteq \eta' H''$. Thesis follows because $H \sqsupseteq H'$ implies $\eta H \sqsupseteq \eta H'$. Then $\langle \Gamma; C \rangle \vdash e' : \tau \triangleright \eta' H''$ and $\eta H \sqsupseteq \eta' H''$.

- case TApp $\quad \langle \Gamma; C \rangle \vdash e_1 : \tau_1 \xrightarrow{\mathbb{P}|H} \tau_2 \triangleright H_1 \quad \langle \Gamma; C \rangle \vdash e_2 : \tau_1 \triangleright H_2 \quad \exists v \in \mathbb{P}. v \subseteq |C|$
  There are three semantic rules from which $C \vdash \eta, e_s \to \eta', e_s'$ can be derived:

  - case app$_1$
    The premise of the semantic rule tell us that e_2 reduces to e_2' ($C \vdash \eta, e_1 \to \eta', e_1'$) and so $e_s' = e_1 e_2'$. By the induction hypothesis $\langle \Gamma; C \rangle \vdash e_2' : \tau_1 \xrightarrow{\mathbb{P}|H} \tau_2 \triangleright H_2'$ with $\eta H_2 \sqsupseteq \eta' H_2'$. The thesis follows because $\langle \Gamma; C \rangle \vdash e_s' : \tau_2 \triangleright H_2' \cdot H_1 \cdot H$ and by 3.3.10 $\eta H_2 \cdot H_1 \cdot H \sqsupseteq \eta' H_2' \cdot H_1 \cdot H$.

  - case app$_2$
    The premise of the semantic rule ensures us that e_1 reduces ($C \vdash \eta, e_1 \to \eta', e_1'$) and $e_s' = e_1' v$. We prove that $\langle \Gamma; C \rangle \vdash e_s' : \tau_2 \triangleright H'$ with $\eta H_2 \cdot H_1 \cdot H \sqsupseteq \eta' H'$. By the inductive hypothesis $\langle \Gamma; C \rangle \vdash e_1' : \tau_1 \xrightarrow{\mathbb{P}|H} \tau_2 \triangleright H_1'$ with $\eta H_1 \sqsupseteq \eta' H_1'$. This implies the thesis because then $\langle \Gamma; C \rangle \vdash e_s' : \tau_2 \triangleright H_2 \cdot H_1' \cdot H$ and by 3.3.10 $\eta H_2 \cdot H_1 \cdot H \sqsupseteq \eta' H_2 \cdot H_1' \cdot H$.

  - case app$_3$
    From the semantic rule we have $e_s' = e\{v/x, e_1/f\}$. We have to prove that $\langle \Gamma; C \rangle \vdash e_s' : \tau_2 \triangleright H'$ with $\eta H_2 \cdot H_1 \cdot H \sqsupseteq \eta H'$. By the typing rule and Lemma 3.3.8 we have that $H_1 = H_2 = \epsilon$ (since e_1 and v are values) and $\langle \Gamma, x : \tau_1, f : \tau_1 \xrightarrow{\mathbb{P}|H} \tau_2; C \rangle \vdash e : \tau_2 \triangleright H$. The thesis follows because we have that $\langle \Gamma; C \rangle \vdash e_s' : \tau_2 \triangleright H$ by Lemma 3.3.9 and $\eta H_2 \cdot H_1 \cdot H \sqsupseteq \eta\epsilon \cdot H \sqsupseteq \eta H$ because of $H_2 \cdot H_1 = \epsilon \cdot \epsilon = \epsilon \sqsupseteq \epsilon$.

- case Tif $\langle\Gamma;C\rangle \vdash e_0 : \mathit{bool} \triangleright H \quad \langle\Gamma;C\rangle \vdash e_1 : \tau \triangleright H_1 \quad \langle\Gamma;C\rangle \vdash e_2 : \tau \triangleright H_2$
  There are semantic rules which drive the reduction of e_s:
  - case if$_1$
    From the premise of the semantic rule it holds $C \vdash \eta, e_0 \rightarrow \eta', e_0'$, hence $e_s' =$ **if** e_0' **then** e_1 **else** e_2. By induction hypothesis it holds $\langle\Gamma;C\rangle \vdash e_0' : \mathit{bool} \triangleright H'$ with $\eta H \sqsupseteq \eta' H'$. The thesis holds because we have $\langle\Gamma;C\rangle \vdash e_s' : \tau \triangleright H' \cdot (H_1 + H_2)$ by exploiting the typing rule and $\eta H \cdot (H_1 + H_2) \sqsupseteq \eta' H' \cdot (H_1 + H_2)$ by 3.3.10.
  - case if$_2$
    The transitions lead to e_1 without modifying history η. The thesis follows from the hypothesis of the typing rule and because $H \cdot (H_1 + H_2) = H_1 + H_2 \sqsupseteq H_1$ since $H = \epsilon \sqsupseteq \epsilon$.
  - case if$_3$
    Similar to the previous case.
- case Tlet
  The transition can be carried out by two semantic rules:
  - case let$_1$
    From the premise of the semantic rule e_1 reduces to e_1' ($C \vdash \eta, e_1 \rightarrow \eta', e_1'$). By induction hypothesis it holds $\langle\Gamma;C\rangle \vdash e_1' : \tau_1 \triangleright H_1'$ with $\eta H_1 \sqsupseteq \eta' H_1'$. The thesis follows because $\langle\Gamma;C\rangle \vdash e_s' : \tau_2 \triangleright H_1' \cdot H_2$ holds by the typing rule and because $\eta H_1 \cdot H_2 \sqsupseteq \eta' H_1' \cdot H_2$ by 3.3.10.
  - case let$_2$
    By Lemma 3.3.12 we have $\langle\Gamma;C\rangle \vdash v : \tau_1 \triangleright \epsilon$ and by Lemma 3.3.9 $\langle\Gamma;C\rangle \vdash e\{v/x\} : \tau_2 \triangleright H_2$. The thesis follows because it holds $H_1 = \epsilon \sqsupseteq \epsilon$ and $\eta \cdot \epsilon \cdot H_2 = \eta H_2 \sqsupseteq \eta H_2$.
- case Top
  Quite standard. The proof consists mainly of using the induction hypothesis and 3.3.10 for the cases op$_1$ and op$_2$. As regards the case op$_3$, it can be proved by applying the rule Tconst.
- case TWith $\langle\Gamma;C\rangle \vdash e_1 : ly_\phi \triangleright H' \quad \forall L_i \in \phi. \langle\Gamma; L_i :: C\rangle \vdash e_2 : \tau \triangleright H_i$
  The transition is driven by two semantic rules:
  - case with$_1$
    From the premise of the semantic rule we know that e_1 reduces to e_1' ($C \vdash \eta, e_1 \rightarrow \eta', e_1'$), hence $e_s' =$ **with**(e_1') **in** e_2. By induction hypothesis it holds $\langle\Gamma;C\rangle \vdash e_1 : ly_\phi \triangleright H''$ with $\eta H' \sqsupseteq \eta' H''$. Thesis follows because $\langle\Gamma;C\rangle \vdash e_s' : \tau \triangleright H'' \cdot \sum_{L_i} \llparenthesis_{L_i} \cdot H_i \cdot \rrparenthesis_{L_i}$ with $\eta H' \cdot \sum_{L_i} \llparenthesis_{L_i} \cdot H_i \cdot \rrparenthesis_{L_i} \sqsupseteq \eta' H'' \cdot \sum_{L_i} \llparenthesis_{L_i} \cdot H_i \cdot \rrparenthesis_{L_i}$ by 3.3.10.
  - case with$_2$
    By Lemma 3.3.13 it is only the case that $L = L_i$ for some $L_i \in \{L_1, \ldots, L_n\}$. The thesis follows because it holds $\langle\Gamma;C\rangle \vdash$ **with**$(\overline{L_i})$ **in** $e_2 : \tau \triangleright \cdot H \cdot \rrparenthesis_{L_i}$ by Tbwith it holds $H = \epsilon$ and $\eta H \cdot \sum_{L_i} \llparenthesis_{L_i} \cdot H_i \cdot \rrparenthesis_{L_i} = \eta \sum_{L_i} \llparenthesis_{L_i} \cdot H_i \cdot \rrparenthesis_{L_i} \sqsupseteq \eta \llparenthesis_{L_i} \cdot H_i \cdot \rrparenthesis_{L_i}$.

- case Twithout
  Similar to the case of Twith rule.

- case Tlexp $\forall i. \langle \Gamma; C \rangle \vdash e_i : \tau \triangleright H_i \quad L_1 \in |C| \vee \cdots \vee L_n \in |C|$
  From the semantic rule lexp we have the transition $C \vdash \eta, e_s \rightarrow \eta\ \mathsf{Disp}(L_i), e_i$. From the premise of the typing rule we know $\langle \Gamma; C \rangle \vdash e_i : \tau : H_i$. The thesis follows because $\eta \sum_{L_i \in \{L_1, \ldots, L_n\}} \mathsf{Disp}(L_i) \cdot H_i \sqsupseteq \eta \mathsf{Disp}(L_i) H_i$ by 3.3.10.

- case Talpha
  By using the rule action we deduce the transition $C \vdash \eta, \alpha(r) \rightarrow \eta\alpha(r), ()$. Since $\langle \Gamma; C \rangle \vdash () : \texttt{unit} \triangleright \epsilon$ by rule Tconst, we can conclude that $\eta\alpha(r) \sqsupseteq \eta\alpha(r) \cdot \epsilon$.

- case Tphi $\langle \Gamma; C \rangle \vdash e : \tau \triangleright H$
  From the semantic rule framing$_1$ we deduce the transition $C \vdash \eta, \phi[e] \rightarrow \eta[_\phi, \overline{\phi}[e]$. By using the premises of the typing rule Tphi and by applying the rule Tbphi we get that $\langle \Gamma; C \rangle \vdash \overline{\phi}[e] : \tau \triangleright H \cdot]_\phi$. The thesis follows from 3.3.10 $\eta[_\phi \cdot H \cdot]_\phi \sqsupseteq \eta[_\phi \cdot H \cdot]_\phi$.

- case Trec
  Similar to the case of the rule Talpha where $receive_\tau$ substitutes $\alpha(r)$.

- case Tsend $\langle \Gamma; C \rangle \vdash e : \tau \triangleright H$
  The reduction of e_s is driven by two rules:

  - case send$_1$
    From the premise of the semantic rule we have the transition $C \vdash \eta, e \rightarrow \eta', e'$. By induction hypothesis we have $\langle \Gamma; C \rangle \vdash e' : \tau \triangleright H'$ with $\eta H \sqsupseteq \eta' H'$. Then the thesis follows because $\langle \Gamma; C \rangle \vdash e'_s : unit \triangleright H' \cdot send_\tau$ by the rule Tsend and $\eta H \cdot send_\tau \sqsupseteq \eta' H' \cdot send_\tau$ by 3.3.10.

  - case send$_2$
    We have the e_s reduces to the value unit ($C \vdash \eta, \mathbf{send}_\tau(v) \rightarrow \eta send_\tau, ()$). The thesis follows because $\langle \Gamma; C \rangle \vdash () \triangleright \epsilon$ by exploiting the rule Tconst and since $H = \epsilon$ we have $\eta H \cdot send_\tau = \eta\epsilon \cdot send_\tau = \eta send_\tau \sqsupseteq \eta send_\tau$.

- case Tbwith $\langle \Gamma; L :: C \rangle \vdash e_2 : \tau \triangleright H$
  There are two semantic rules driving the reduction of e_s:

  - case with$_3$
    From the premise of the semantic rule we know e_2 reduces to e'_2 ($L :: C \vdash \eta, e_2 \rightarrow \eta', e'_2$). By inductive hypothesis we have that $\langle \Gamma; L :: C \rangle \vdash e'_2 : \tau \triangleright H'$ with $\eta H \sqsupseteq \eta' H'$. The thesis follows because $\langle \Gamma; C \rangle \vdash \mathbf{with}(\overline{L})\ \mathbf{in}\ e'_2 : \tau \triangleright H'$ and $\eta H \rrparenthesis \sqsupseteq \eta' H' \rrparenthesis$ by 3.3.10.

  - case with$_4$
    Immediate by Lemma 3.3.11.

- case Tbwith
  Similar to the previous case.

- case Tbphi $\langle\Gamma;C\rangle \vdash e : \tau \triangleright H$
  We can carry out the transition through two semantic rules:

  - case framing$_2$
    Since by the premises of the rule framing$_1$ we have that $C \vdash \eta, e \rightarrow \eta', e'$, we can use the inductive hypothesis so that we have $\langle\Gamma;C\rangle \vdash e' : \tau \triangleright H'$ such that $\eta H \sqsupseteq \eta' H'$. By 3.3.10 we have that $\eta H]_\phi \sqsupseteq \eta' H']_\phi$.
  - case framing$_3$
    In this case we have $e = v$. By the premise of the typing rule and by Lemma 3.3.12 we have $\langle\Gamma;C\rangle \vdash v : \tau \triangleright \epsilon$. We have to prove that $\eta H]_\phi \sqsupseteq \eta]_\phi \cdot \epsilon = \eta\epsilon\cdot]_\phi$. Since $H = \epsilon$, $\eta H]_\phi = \eta\epsilon\cdot]_\phi \sqsupseteq \eta\epsilon\cdot]_\phi$ holds.

□

The lemma below allows us to deduce the syntactic form of a value from its type (exploited in Lemma 3.3.2).

Lemma 3.3.13 (Canonical form). *Let v be a value*

1. *If $\langle\Gamma;C\rangle \vdash v : \tau_c \triangleright H$ then $v = c$ for some c.*
2. *If $\langle\Gamma;C\rangle \vdash v : \tau_1 \xrightarrow{\mathbb{P}|H} \tau_2 \triangleright H'$, then $v =$ **fun** $f x \Rightarrow e$ for some f, x, e.*
3. *If $\langle\Gamma;C\rangle \vdash v : ly_{\{L_1,\ldots,L_n\}} \triangleright H$, then $v \in \{L_1, \ldots, L_n\}$.*

Proof. The proof is similar to the one of Lemma 2.3.7. □

Proposition 3.3.3 (Over-approximation). *If $\langle\Gamma;C\rangle \vdash e : \tau \triangleright H$ and $C \vdash \epsilon, e \rightarrow^* \eta, e'$, then $\eta \in [\![H]\!]$.*

Proof. By induction on the length i of the computation, by repeatedly applying Lemma 3.3.1 we can prove that $\langle\Gamma;C\rangle \vdash e' : \tau \triangleright H'$ with $H \sqsupseteq \eta H'$. Since by definition $\eta \in [\![\eta H']\!]$ and since $H' \sqsubseteq H_1 \Rightarrow [\![H']\!] \sqsubseteq [\![H_1]\!]$, we have that $\eta \in [\![H]\!]$. □

Lemma 3.3.2 (Progress). *Let e be a closed expression such that $\langle\Gamma;C\rangle \vdash e : \tau \triangleright H$. If $C \vdash \eta, e \nrightarrow$ and ηH is valid and it is compliant with the communication protocol, then e is a value.*

Proof. By induction on the depth of the typing derivation, and then by cases on the last rule applied. Most of cases are equal to the ones of Lemma 2.2.2. Here, we consider only the cases for new constructs (the case for the rule Twithout is similar to the one of rule Twith).

- case Talpha $e_s = \alpha(r)$
  The semantic rule action is an axiom and we always can apply when $e_s = \alpha(r)$ to reduce e_s (contradiction).

- case Tphi $e_s = \phi[e]$
  If $\phi[e]$ is stuck, then it is only the case that it does not hold $\eta^{-[]} \vDash \phi$. Since $\eta[_\phi \cdot H \cdot]_\phi$ is valid, then $\eta[_\phi \in [\![\eta[_\phi \cdot H \cdot]_\phi]\!]$ (remember that the semantics of a history is prefix closed), hence $\eta[_\phi$ is valid and $\eta^{-[]} \vDash \phi$. So we can apply the semantic rule framing$_1$ (contradiction).

- case TREC $e_s = \mathbf{receive}_\tau$
  The semantic rule RECEIVE is an axiom and we always can apply when $e_s = \mathbf{receive}_\tau$ to reduce e_s (contradiction).

- case TSEND $e_s = \mathbf{send}_\tau(e)$
  If e_s is stuck, then it is only the case that e is stuck. By induction hypothesis e is a value v. Then, e_s reduces to v by the rule SEND2 (contradiction).

- case TBWITH $e_s = \mathbf{with}(\overline{L})$ **in** e_2
  If e_s is stuck, then e_2 is stuck. By induction hypothesis e_2 is a value v, so we can apply the rule WITH4 (contradiction).

- case TBWITHOUT $e_s = \mathbf{without}(\overline{L})$ **in** e_2
  Similar to the previous case.

- case TBPHI $e_s = \overline{\phi}[e]$
  If e_s is stuck, then either e is stuck or it does not hold $\eta^{-[]} \models \phi$.

  1. case e
     By induction hypothesis e is a value v, so we have the case (2).
  2. case $\eta^{-[]} \not\models \phi$.
     By Lemma 3.3.1 $\langle \Gamma; C \rangle \vdash e' : \tau \triangleright H'$ with $\eta H \sqsupseteq \eta' H'$. Since $\eta H \cdot]_\phi \sqsupseteq \eta' H' \cdot]_\phi$ and $\eta H \cdot]_\phi$ is valid and balanced then $\eta' H' \cdot]_\phi$ is valid and balanced. In particular, there is a history $\eta\eta' \in [\![\eta' H']_\phi]\!]$ with such an unmatched $[_\phi$. Since $\eta'\eta''$ is valid then $(\eta'\eta'')^{-[]} \models \phi$, hence $\eta'^{-[]} \models \phi$ by 3.2.1 (contradiction).

□

Theorem 3.3.4. *If $\langle \emptyset; C \rangle \vdash e : \tau \triangleright H$ and H is valid with balanced policy framings and it is compliance with the communication protocol, then either the computation terminates by yielding a value ($C \vdash \epsilon, e \rightarrow^* \eta', v$) or it diverges, but it never gets stuck.*

Proof. Assume that $C \vdash \epsilon, e \rightarrow^i \eta', e'_s \not\rightarrow$ for some $i \in \mathbb{N}$ where e' a non-value stuck expression. By induction on the length i of the computation, by repeatedly applying Preservation Lemma, we have that $\langle \Gamma; C \rangle \vdash e' : \tau \triangleright H'$ and $\eta' H' \sqsubseteq H$. Since H is valid and balanced also $\eta' H'$ is valid and balanced. Then the Progress Lemma suffices to show that e'_s is a value (contradiction). □

3.4 Model Checking

In this section we introduce a model-checking machinery for verifying whether a history expression is compliant with a policy ϕ and a protocol P. The idea is that the environment specifies P, and it accepts a component to join only if the component follows P during the communication.

3.4.1 Safety checking

To formalize our notion of safety, we adopt the approach introduced in [HMS06], where a security policy φ is actually a safety property, expressing that nothing bad will occur during a computation. Policies are expressed through standard Finite State Automata (FSA) and we assume a *default-accept* paradigm, i.e. only the unwanted behaviour is explicitly mentioned. Consequently, the language of φ is the set of *unwanted traces*, hence an accepting state is considered as offending. Below we denote with $L(\varphi)$ the language of φ.

Example 3.4.1. Returning to our running example in Section 3.1, consider the policy φ in the function `serve` that forbids to write action on the library. The automaton specifying this policy is displayed in Figure 3.11a. If the action `write(library)` is carried out, the automaton performs the transition toward the offending state q_1 to indicate a policy violation. The self loop indicates that for each event different from `write(library)` the automaton consumes the symbol and remains in the same state. Note that the alphabet of the automaton is the set of the events $\Sigma \subseteq ev$ occurring in the program. Obviously, this set is computable from the code and it is finite.

As a further example of FSA implementing a policy, consider Figure 3.11b where we depict a simple policy φ_2. It prevents the occurrence of an action `write` after an action `read` on a file system, say the resource `fs`, at the beginning of the computation. The language accepted by this automaton consists of all strings containing first a `read(fs)` and then `write(fs)` (unnecessarily consecutive). Also in this case Σ denotes the events may occur in the program and reaching the accepting state indicates that the program performed the forbidden actions on the resource `fs` violating φ_2.

We now complete the definition of validity of a history ($\eta \models \varphi$) introduced in Section 3.2.3:

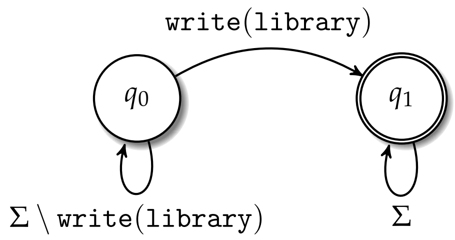
Definition 3.4.1 (Safety)**.** Let η be a history without framing events, then $\eta \models \varphi$ iff $\eta \notin L(\varphi)$.

The semantics of a history expression may contain histories with redundant framings, i.e. nesting of the same policy. For instance, $\mu h.\, (\varphi[\alpha(r)h] + \epsilon)$ generates $[_\varphi\alpha(r)[_\varphi\alpha(r)]_\varphi]_\varphi$. Intuitively, a history η has *redundant framing* whenever the active policies $ap(\eta')$ contain a duplicate φ for some prefix η' of η.

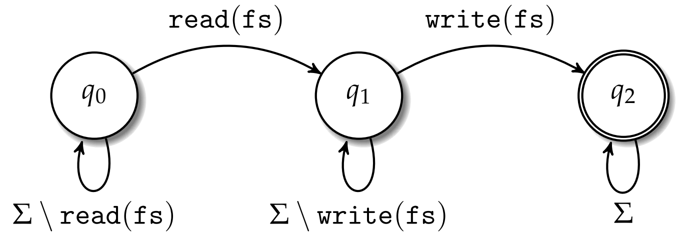
In [BDFZ09] they proved that redundant framing can be eliminated without affecting validity. This is because the expressions monitored by the inner-framings are already under the scope of the outermost one. For this reason the definition of validity uses $\eta^{-[]}$, i.e. the history purged from framings. To do that, in [BDFZ09] there is a *regularisation* algorithm that given a history expression H returns its regularized version $H{\downarrow}$, satisfying the followings

1. each history in $[\![H{\downarrow}]\!]$ has no redundant framing;

2. $H{\downarrow}$ is valid if and only if H is valid.



(a) The automaton of the security policy ϕ of our motivating example.



(b) The automaton implementing the security policy ϕ_2 which forbids an action `write` after `read`.

Figure 3.11: Example of automata defining security policies.

Hence, we can reduce the problem of checking validity of a history expression H to the corresponding one for a history expression $H{\downarrow}$ without redundant framings.

Our approach fits into the standard *automata based* model checking [VW86]. Indeed, there is an efficient and fully automated method for checking the $\models$ relation for a regularized history expression H. Let $\{\phi_i\}$ be the set of all policies ϕ_i occurring in H. From each ϕ_i it is possible to obtain a *framed automaton* $\phi_i^{[]}$ such that η is valid iff $\eta \notin L(\bigcup \phi_i^{[]})$. The detailed construction of framed automata is in [BDFZ09]. Roughly the framed automaton for the policy ϕ has two copies of ϕ. The first copy has no offending states, the second has the same offending states of ϕ. Intuitively, one uses the first copy when the actions are made while the policy is not active. The second copy is reached when the policy is activated by a framing event. Indeed, there are edges labelled with $[_\phi$ from the first copy to the second one and with $]_\phi$ in the opposite direction. So when a framing gets activated we can also reach an offending state.

Example 3.4.2. Figure 3.12a displays the framed automaton for the policy ϕ of Example 3.4.1. This automaton has two new states q_0' and q_1' (accepting) that mimic their counterparts. These new states are reached when the policy ϕ is activated as the transition labelled by the event $[_\phi$ signals. Note that when the policy is not active a possible action *write*(*library*) causes a transition in the automaton from the state q_0 to q_1, without causing any violations (q_1 does not accept). When we activate ϕ, the transition from q_1 to q_1' is performed and there is a security violation because q_1' is accepting. As a further example, in Figure 3.12b we show the framed automaton for the security policy ϕ_2 of Example 3.4.1. Also in this case we create new states mimicking the existent ones and setting transitions to manage the activation and the deactivation of the policy.

In [BDF09] Bartoletti et al. proved that the semantics of a history expression H is a context-free language because there exists a pushdown automaton recognising $[\![H]\!]$. Then, validating a regularized history expression H amounts to verifying that $[\![H]\!] \cap \bigcup L(\phi_i^{[]})$ is empty. We can state the following:

Theorem 3.4.1 (Model checking policies)**.** *A given history expression H is valid if and only if* $L(H{\downarrow}) \cap \bigcup L(\phi_i^{[]}) = \emptyset$.

Since regular languages are closed by union, context-free languages are closed by intersection with a regular language and the emptiness of context-free languages is decidable the above problem is decidable and we can use standard algorithms to solve it [Hop79].

Example 3.4.3. As an example of checking policies, consider the history expression H_s of the function `serve`, its semantics $[\![H_s]\!]$ in Example 3.3.1 and assume that the subterm H never generates string with the event `write(library)`. Consider also the automaton specifying the policy ϕ and its language $L(\phi)$ whose strings have the form $(ev \setminus \texttt{write(library)})^\star\texttt{write(library)}ev^\star$. Since the strings in $L(\phi)$ always contains at least one `write(library)` event, whereas the strings in $[\![H_s]\!]$ do not, there are no strings in $[\![H_s]\!]$ which are also in $L(\phi)$ (their intersection is empty), then our function `serve` is compliant to ϕ.

3.4.2 Protocol compliance

In this subsection we described how checking whether a program well-behaves when interacting with other parties through the bus. We assume that a protocol P is a (regular) sequence S of $send_\tau$ and $receive_\tau$ actions possibly repeated. Formally, our protocols are specified by the following grammar:

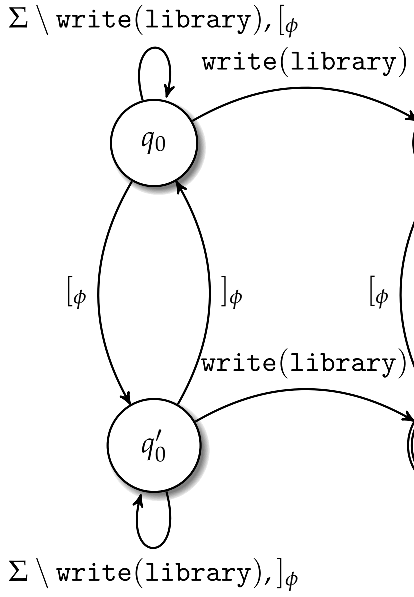
$$P ::= S \mid S^* \qquad\qquad S ::= \epsilon \mid send_\tau.S \mid receive_\tau.S$$

The symbol S^* indicates that the action can be repeated. We require a program to interact with the bus respecting the protocol, but we do not force the program to do the whole interaction specified, this means that a program may perform only a prefix of the specified sequence. For this motivation the language $L(P)$ of P is a prefix closed set of actions, obtained by considering all the prefixes of the sequences defined by P. Then our protocol compliance procedure only requires that all the histories generated by a program (projected so that only $send_\tau$ and $receive_\tau$ appear) belong to $L(P)$.

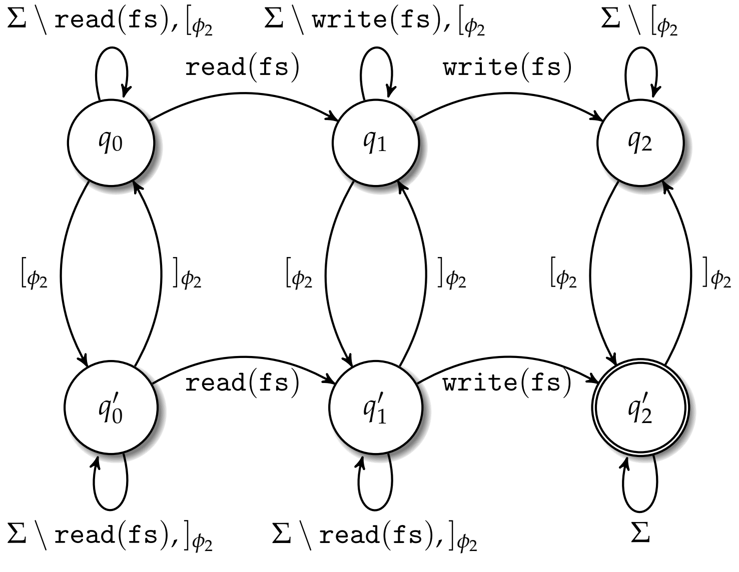
Let H^{sr} be a projected history expression where all non $send_\tau$, $receive_\tau$ events have been removed. Then we define compliance to be:

Definition 3.4.2 (Protocol compliance)**.** Let e be an expression such that $\langle \Gamma, C\rangle \vdash e : \tau \triangleright H$, then e is compliant with P if $[\![H^{sr}]\!] \subseteq L(P)$.

The following theorem provides us with a decidable model checking procedure to establish protocol compliance. In its statement we write $\overline{L(P)}$ for the complement of $L(P)$. Note that the types annotating $send_\tau/receive_\tau$ can be kept finite in both $L(P)$ and $\overline{L(P)}$, because we only take the types occurring in the effect H under checking.


$\Sigma \setminus \texttt{write(library)}, [_\phi$
$\Sigma \setminus [_\phi$
$\texttt{write(library)}$
q_0
q_1
$[_\phi$
$]_\phi$
$[_\phi$
$]_\phi$
$\texttt{write(library)}$
q'_0
q'_1
$\Sigma \setminus \texttt{write(library)},]_\phi$
Σ

(a) The framed automaton of the policy ϕ in our motivating example.


$\Sigma \setminus \texttt{read(fs)}, [_{\phi_2}$
$\Sigma \setminus \texttt{write(fs)}, [_{\phi_2}$
$\Sigma \setminus [_{\phi_2}$
$\texttt{read(fs)}$
$\texttt{write(fs)}$
q_0
q_1
q_2
$[_{\phi_2}$
$]_{\phi_2}$
$[_{\phi_2}$
$]_{\phi_2}$
$[_{\phi_2}$
$]_{\phi_2}$
$\texttt{read(fs)}$
$\texttt{write(fs)}$
q'_0
q'_1
q'_2
$\Sigma \setminus \texttt{read(fs)},]_{\phi_2}$
$\Sigma \setminus \texttt{read(fs)},]_{\phi_2}$
Σ

(b) The framed automaton for the security policy ϕ_2.

Figure 3.12: Examples of framed automata

Theorem 3.4.2 (Model checking protocols)**.** *Let e be an expression such that* $\langle \Gamma, C \rangle \vdash e : \tau \triangleright H$*, then e is compliant with P iff*

$$L(H^{sr}) \cap \overline{L(P)} = \emptyset$$

Also in this case, we reduce our model checking procedure to verify the emptiness property of context-free languages. We remark that, in our model, protocol compliance cannot be expressed only through security policies introduced above. As a matter of fact, $L(H^{sr}) \cap \overline{L(P)} = \emptyset$ expresses that H has no forbidden communication patterns, and this is a requirement much similar to a default-accept policy. Recall that $[\![H^{sr}]\!] \subseteq L(P)$ requires that some communication pattern in compliance with P *must* be done. This highlights the different nature of security policies and protocols in our framework and explains why we keep them distinct and we need two different model-checking procedures.

Example 3.4.4. Back to our running example, consider the protocol

$$P = (send_{\tau}.send_{\tau'}.receive_{\tau''})^*$$

and the reduced history expression of the body of the function `ftsearch`

$$H_{sr} = send_{\tau} \cdot send_{\tau'} \cdot receive_{\tau''}.$$

It is easy to verify that the app running on the smartphone is compliant with the protocol because $[\![H_{sr}]\!] \cap \overline{L(P)} = \emptyset$.

3.5 Remarks

The material of this chapter is an extension of the one in Chapter 2, thus all remarks we made at the end of Section 2.2.2 about the relationships with [HIM11, CS09, CCT09] are still valid. Here, we introduced constructs for resource manipulation, enforcement of security policies and communication and we defined a static mechanism that, to the best of our knowledge, was not tackled in the area COP languages.

Chapter 4

A Declarative approach to Context-Oriented Programming

As discussed in Section 1.1.6, there are some open issues in context-oriented languages. In particular, the primitives for describing and querying the context are at a too low level, which make programming complex adaptive applications difficult. As matter of fact the behavioural variations implemented as partial definition of classes, procedures and modules are not able to express dynamic adaptation patterns. ContextML introduced in Chapter 3 suffers from these limitations as well. Indeed, layers are binary predicates so we cannot use them to represent structured pieces of information, capable of affecting the dispatching mechanism. Furthermore, layered expressions are not flexible enough to express involved adaptation patterns.

For these reasons, we introduced a new ML-based language, ML_{CoDa}, equipped with linguistic primitives for context-awareness, coupled with a logical language for context definition and management. In particular, in our proposal we suggest to use structured knowledge bases as contexts and behavioural variations are first class constructs. As a by-product, the context can be dynamically constrained to requirements that may change over time, thus making a fine control of adaptation possible. For the time being, ML_{CoDa} is sequential, and we offer no construct for event-based adaptation.

As ContextML, ML_{CoDa} is designed to support mechanized verification based on static analysis techniques. Differently from ContextML, the verification of ML_{CoDa} programs occurs into two stages: a type and effect system (at compile time) to compute an abstraction (history expression) and a control-flow analysis (at loading time) to verify that the various contexts hosting the program a runtime have the required capabilities.

First we describe the design of ML_{CoDa} (Section 4.1) and a running example (Section 4.2). Then, we introduce the syntax and the semantics of the language (Section 4.3) and our static analysis (Section 4.4 and Section 4.6).

4.1 The Design of ML_{CoDa}

Our first concern in the design of our COP language is the context. We follow the line proposed by Jackson and Zave [ZJ97] who defined the context as the part of the real

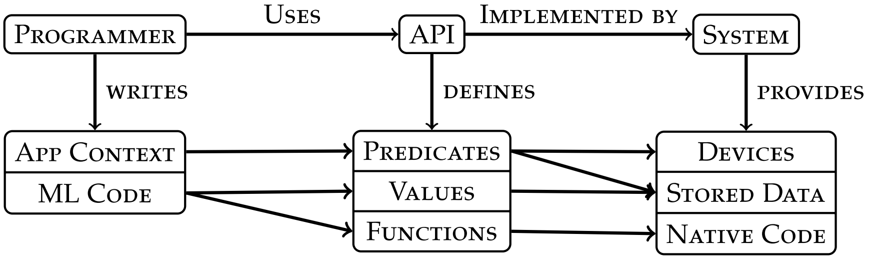


Figure 4.1: The programming model of ML_{CoDa}

world which interacts with (and is affected by) a program and which potentially evolves independently. They pointed out the importance of the context in the development process because it includes crucial information, such as the requirements and the domain knowledge (assumptions and properties). In this setting, defining and modelling the context requires skills different from those needed for programming applications and usually this task is performed by specialized engineers, namely *requirements engineers*. These observations, as well as separation of concerns, have motivated us to define ML_{CoDa} as a two-component language: a declarative constituent for describing 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 [OT11, Lok04]. This representation lets adaptive programs query the context by simply verifying whether a given property holds in it, i.e. by checking a Datalog goal. During the needed deductions the relevant information is also retrieved.

As for programming adaptation, we propose two mechanisms. The first one takes care of those program variables that assume different values depending on the different properties of the current context. To make that explicit, we introduce the notion of *context-dependent binding*, a sort of dynamic binding.

The second mechanism is a more powerful kind of behavioural variation. As said, behavioural variations are not first class constructs in COP languages, or rather, they are expressed as partial definition of procedures or classes or methods or modules. In ContextML we express them as layered expressions (see Chapter 2). ML_{CoDa} extends layered expressions by introducing Datalog goals and by making them first class citizens, so that they can be bound to identifiers, passed as arguments to, and returned by functions. This extension provides us with more natural means for programming dynamic adaptation patterns, as well as reusable and modular code, than the previous proposals where only layers are values. For example, in ML_{CoDa} a behavioural variation can be supplied by the context, and then composed with existing ones, so making easier implementing the notion of autonomic element. For example, see the definition of the function `addDefault` later on in this section.

We now discuss the programming model of ML_{CoDa}, displayed in Figure 4.1. It assumes that the virtual machine of the language provides its users with a collection of system variables, values, functions and predicates through a predefined API. Con-

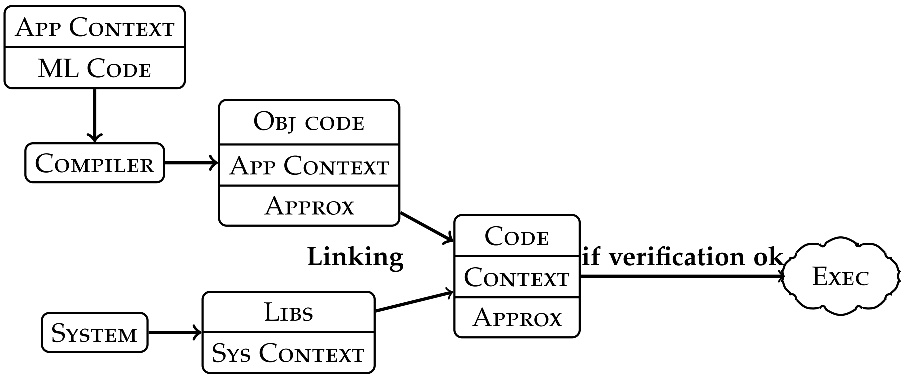


Figure 4.2: Execution model of $\mathrm{ML_{CoDa}}$

sequently, the context is split in two parts: the *system* and the *application* context. The first one is provided by the virtual machine through its API (system predicates) and contains information about the device running the application. Obviously, the actual values of system predicates are only available at runtime. Whereas the application context stores specific knowledge of the application, and its contents are initially filled in by the programmer.

The execution model of $\mathrm{ML_{CoDa}}$ is shown in Figure 4.2. We assume that the compiler produces a triple (C, e, H), where C is the application context, e is the object code and H is an approximation of e, used to verify properties about the program (we adopt the approach described in Section 1.2.5). Given such a triple, at loading-time the virtual machine performs a *linking* and a *verification* phase. The linking phase resolves system variables and links the application context to the system one, so obtaining the initial context which, of course, is checked for consistency. Note that linking itself makes a first adaptive step, because it enables the application to use the capabilities of the host system, be they resources, data or code. In the spirit of Proof-Carrying code [NL98] and of the Java Bytecode Verifier [LYBB13, Ros04], the verification phase exploits the approximation H to check that the program e will adapt to *all* contexts that may arise at runtime. If both phases succeed, program evaluation begins, otherwise it is aborted.

4.2 A Guided Tour of $\mathrm{ML_{CoDa}}$ Features

In this section we illustrate and discuss the main features of $\mathrm{ML_{CoDa}}$ through a running example. 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.

4.2.1 A Multimedia Guide to 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, which 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; for instance, if he is a European citizen and is 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.

4.2.2 The Context

The most important design choice in the development of a context-aware application is deciding the contextual information, i.e. what the context is. In our example, the context will certainly contain information about the smartphone capabilities, both hardware and software; data about the physical environment, e.g. in which museum room the user is; and the user profile and preferences. These data come from different sources and have different representations. Some of these data are application independent, like those about smartphone capabilities and about the physical environment; other data are application-dependent like user preferences.

We argued that the context is conveniently structured in two parts: the *system* and the *application context*, and this is how $\mathrm{ML}_{\mathrm{CoDa}}$ handles this notion. The system context pertains to the environment running the application, for example to the virtual machine of the language. As said, we assume this part of the context is accessible through a predefined API, and its actual data is only available at runtime. The application context stores specific knowledge of the application, and its contents are filled in by the programmer. Of course, this context can use information from the system context, using the API. At runtime, the actual execution context results from linking the system and the application contexts.

Tools are clearly needed to access and manipulate all kinds of contextual data in an easy and uniform way. As discussed in the Section 4.1, we foster a programming methodology where context management is neatly separated from program code. For that, one of the components of $\mathrm{ML}_{\mathrm{CoDa}}$ is Datalog, which provides programmers with well-established tools for declaratively describing the context and reasoning on it.

As for the system context, Datalog predicates and facts represent properties and capabilities of devices, and come with the abstract machine of the language that natively implements them.

For example, the API of the system context can offer the *(system) predicate* `device` to test if a specific sensor or hardware is available. In our running example, if the smartphone is equipped by an accelerometer, the fact

```
device(accelerometer)
```

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

Predicates can also encapsulate native code for more complex tests requiring complicated computations. Assume that the API provides us with the predicate `headphones` to test whether headphones are plugged in or not. If plugged in, the following fact will hold

```
headphone(plugged).
```

Furthermore, the predicate `level_noise` encapsulates a routine computing the level of noise in the room where the smartphone is, through complex operations, like interacting with the microphone and executing a numerical algorithm. One can check if the value returned by `level_noise` exceeds a given `noise_threshold` by the following

```
level_noise(x), x > noise_threshold.
```

In the previous snippet of code, the place-holder x receives a value, but an argument of a predicate can also be a constant (e.g. `accelerometer` above) so passing a value. Note that this mechanism for both passing and receiving values makes it easier to interact with the context, and supports our choice of having a two-component language.

Predicates not only represent devices (and their implementations), but also provide the programmer with a friendly interface for interacting with the software services offered by the system. For example, a user with disabilities may have a smartphone with a speech synthesizer helping him to use some applications. In ML$_{CoDa}$, we can test the availability of this service, typically implemented thought a mix of hardware and software, with the predicate

```
speech_synthetizer(y).
```

If a synthesizer is available, the variable `y` will be bound to the *handle* through which the service can be invoked. Note that a synthesizer is an object of the system, supplied as a software service.

Besides system predicates that are independent of the specific application, the programmer specifies predicates and clauses that constitute the other component of the context, namely the *application* context.

In our example, the multimedia guide stores and accesses data about the user's profile. The programmer defines then the following predicates

```
user_age(n)
user_country(c)
user_work(job)
```

that the application running on the smartphone uses to retrieve the age, the country and the job, respectively, as declared by the visitor at the registration desk and stored in the museum database. Similarly for preferences and accessibility options. 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 and their retrieval may require some deduction, because the contextual information comes from different sources. To this aim, Datalog clauses and its deduction machinery are clearly an asset of our proposal.

For example, our multimedia guide will download the images of an exhibit, selecting the size more appropriate for the dimension of the smartphone screen, classified, e.g. as small, medium or large, regardless of its exact size. This kind of aggregate information is given by the predicate `sscreen`, easily defined by the following clauses (`max_resolution` and `min_resolution` have the obvious meaning and are predefined)

```
sscreen(small) ← max_resolution(320,180).
sscreen(medium) ← max_resolution(1024,480).
sscreen(large) ← min_resolution(1024,480).
```

Furthermore, the orientation of the device (portrait/landscape) drives the optimal layout to visualize information. Also, this information results from aggregating low-level data:

```
screen_orientation(portrait) ← device(accelerometer),
                               accelerometer(x,y,z),
                               portrait_pos(x,y,z).
screen_orientation(landscape) ← device(accelerometer),
                                accelerometer(x,y,z),
                                landscape_pos(x,y,z).
```

The screen orientation is deduced by interacting with the accelerometer and by retrieving and testing the values of axes, through system predicates. Note that the programmer only uses `screen_orientation` and is shielded by its actual definition (alternative definitions are of course possible, e.g. when the smartphone has no accelerometer).

Datalog clauses can also express business logic rules taking into account the context. For example, a ticket has different prices, depending on the information the user supplies during registration, as follows:

```
ticket(reduced) ← user_age(x), x < 10,
                  user_country(y),
                  Europe(y).
ticket(reduced) ← user_age(x), x > 65,
                  user_country(y),
                  Europe(y).
ticket(reduced) ← user_work(student).
ticket(free) ← user_work(student),
               user_study(art).
```

The first two clauses grant a reduced ticket to under 10 or over 65 users from a European country; the third clause does the same to students; additionally if they study art, entrance is free. The predicates `user_age`, `user_country`, `user_work` and `user_study` retrieve data from the user profile and we check if they satisfy the given constraints, e.g. $\texttt{x} < \texttt{10}$ or Europe(y).

We list further clauses, part of the application context of our running example. Predicates only_text and only_speech concern which media providing 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 she declared hearing impairments. Note that in the third clause, we use the system predicates about noise, described above.

The appropriate video format HD is defined below, and requires the smartphone to support high definition, to run a codec, and to have enough battery:

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

The last group of predicates describes capabilities of the smartphone for retrieving URLs. The first predicate holds when the smartphone can decode QR code; the second one if direct communication is possible:

```
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).
```

4.2.3 The application behaviour

We now discuss the two main constructs, *context-dependent binding* and *first class behavioural variations*, that distinguish ML$_{\text{CoDa}}$. They are used to program how applications adapt to changes in the context. We assume that programmers only interact with the system running the application via the API, which 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. where it is taken from) 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.

In our example, the input of the multimedia guide consists of the interactions between the user and the program GUI and of the downloads from the exhibit pages. Contextual data, such as the predicates `only_text` and `only_speech`, affect both the downloads and the way they are presented to users: if `only_text` holds the application will not download videos and audio files.

Context-dependent binding is the 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 a clause `when` requiring that a given goal hold in the context. The variable declared (called *parameter* hereafter) may denote different objects, with different behaviour, which is a major aspect of adaptivity. A parameter is *dynamically* bound, according to which goal is satisfied by the current context, as exemplified below.

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` will be called and the returned value will be bound to the current occurrence of `txt_label`. Note that `getLabel` is *not* called when `txt_label` is declared, but when it *is used*, in a sort of call-by-name style. To better illustrate this mechanism, consider the following snippet of code, that adds components to the main window of the application

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

If the goal ¬`only_speech`() holds in the context where `setMainWindow` is running, then `getLabel`() will be evaluated and the returned value will be bound to `txt_label`.

We now exemplify the multiple declarations 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 multiple declarations, and the appropriate one will be selected when the above `addComponent window vcanvas` is run

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

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

```
(* definition of setMainWindow and 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 `video(hd)`, ¬ `only_text()`, but not `video(low)`, ¬ `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`.

The other main feature to express context-dependency is that of *behavioural variation*, which as usual alters the flow of application depending on the context. The construct of ML$_{\text{CoDa}}$ extends layered expression of ContextML, indeed, roughly it is a list of pairs (goal, expression), but with goals in place of layers. Additionally, ML$_{\text{CoDa}}$ behavioural variations have parameters and they are *first-class values*, so we can bind them to variables, pass them to functions, manipulate them with ad hoc operators, and so on. Note that they are also similar to functional abstractions: we need to apply them in order to evaluate them. As in ContextML, the application triggers a *dispatching mechanism* that queries the context at runtime and selects the first expression whose goal holds.
We introduce the behavioural variation `url` which has a not used dummy argument "_"

```
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
```

Depending on the smartphone capabilities, this behavioural variation 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 with 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 variations application represented by #; for details see Section 4.3).

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 smartphone display capabilities.

It is worth noting that behavioural variations are values, so like other values we can write code that manipulate them. This allows programmers to implement adaptation patterns they think more appropriate for their applications and write more modular and extendible code. To manipulate behavioural variations we equip $\mathrm{ML}_{\mathrm{CoDa}}$ with the binary operator ∪ that allows extending 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 ∪ (w){ True. y }
```

It takes as arguments a behavioural variation `bv` and a value `dv`, and it extends `bv` by adding a default case which is always selected when no other case applies. Note that this function implements a sort of program extension pattern that may requires an intricate definition with available COP mechanisms. In particular, the above `addDefault` allows $\mathrm{ML}_{\mathrm{CoDa}}$ programmer to easily implement the standard notion of *basic behaviour* of COP languages.

Besides the features that describe and query the context, and those that adapt program behaviour, $\mathrm{ML}_{\mathrm{CoDa}}$ is also equipped with constructs that update the context by adding facts (`tell`) and removing them (`retract`) (a sort of *indefinite activation mechanism*, see Section 1.1.3). For example, the function below inserts into the context the fact returned by `getCheckedOption` applied to the argument `accRadioButton`. We assume this application will yield either the fact `user_acc_opt`(`deaf`) or `user_acc_opt`(`blind`)

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

Note that `tell` and `retract` belong to the functional component of ML$_{\text{CoDa}}$, 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, through these constructs the programmer can insert or remove facts of the application context, only. Instead, the system context can only be updated through the provided API, so a control is possible for preventing the programmer to drive the system in an unsafe state, e.g. by inhibiting access to a hardware device if the level of battery is too low.

An application can fail to adapt to a context, both because the programmer assumed the presence of functionalities that the current system context does not support, or because of some programming errors causing some crucial facts to be removed. This happens when a parameter cannot be resolved in a context-dependent binding or when a behavioural variation gets stuck, i.e. the dispatching mechanism fails. In both cases the reason is that the goals occurring in these constructs do not hold in the current context.

Back to our example, consider the function `setMainWindow`. For evaluating it, the function `addComponent` 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 runtime error occurs.

As an example of the dispatching mechanism failure, consider evaluating the function `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 and an error is triggered.

To avoid these runtime errors from happening, we equip ML$_{\text{CoDa}}$ with a two-phase static analysis that detects if an application will be able to adapt to its execution contexts.

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

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 retuned by the invocation of `getRemoteData`. The annotation H_1 is

$$H_1 = ask\, G_1 \otimes ask\, G_2,$$

where

$$G_1 = \leftarrow \texttt{direct\_comm}()$$
$$G_2 = \leftarrow \texttt{use\_qrcode}(\texttt{decoder}), \texttt{camera}(\texttt{cam})$$

are the goals of `url` (for readability, here we use a simplified syntax for type annotations; for details see Section 4.4). Intuitively, H_1 says that one of G_1 or G_2 must be satisfied by the context in order to successfully apply the function `getExhibitData`.

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

$$H_2 = \texttt{tell}(\texttt{user\_acc\_opt}(\texttt{deaf})) + \texttt{tell}(\texttt{user\_acc\_opt}(\texttt{blind}))$$

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

The effects are exploited at *loading time* to verify that the application can adapt to all contexts arising at runtime (Section 4.6). To do that, our control-flow analysis first builds a graph to trace how the initial context evolves during execution. Then, for all possible invocations of the dispatching mechanism, the analysis controls if it will succeed, i.e., if the contexts in the graph satisfy at least one of the goals of the behavioural variation involved.

4.3 The Abstract Syntax and the Semantics of $\mathbf{ML_{CoDa}}$

As discussed above, $\mathrm{ML_{CoDa}}$ consists of two components: Datalog with negation to describe the context and a core of ML extended with COP features. Its syntax and its structural operational semantics follow.

4.3.1 Syntax

First, we intuitively introduce our Datalog dialect. Let *Var* (ranged over by $x, y, ...$), *Const* (ranged over by $c, n, ...$) and *Predicate* (ranged over by $P, ...$) be a set of variables, of basic constants and of predicate symbols, respectively. The syntax is

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

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

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

A Datalog program is a finite set of facts and clauses. In the following we assume that each Datalog program is safe [CGT89], 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) each variable occurring in a negative literal must also occur in a positive literal of the same clause. Predicates are split into two categories: extensional and intensional. The first one represents concrete context objects, such as resources, user preferences and data. The intensional predicates instead describe relationships and properties about context objects. For instance, the predicate `user_prefer` of Section 4.2 is extentional, because it describes a user preference; whereas `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. Also we require that given a clause $A \leftarrow B$, the head A only contains intensional predicates, and the body B can contain all kinds of predicates, if non-empty. Also this requirement can be enforced by a syntactic analysis of a program.

To deal with negation, we assume our world closed and we admit stratified programs, i.e. only *Stratified Datalog* [CGT89].

Since our context is split into the system and the application context, we divide each of our syntactic categories into two disjoint classes: those of the system and those of the application. The system ones, supplied by the API, are marked through a bar over their name; the second ones are defined by the programmer. For example, $\overline{x}$ is a system variable and A is an application atom. Furthermore, given a clause $A \leftarrow B$, we require that A is an application atom while the body B can contain both application and system literals. Also in this case we can easily ensure this requirement using a syntactic analysis.

The functional part inherits most of the ML (and ContextML) constructs. Specially, the behavioural variation of ML$_{\text{CoDa}}$ are an extension of the layered expressions of ContextML. The syntax follows:

$$\tilde{x} \in DynVar\ (Var \cap DynVar = \emptyset) \qquad C, C_p \in Context$$

$$
\begin{array}{rlcl}
\underline{Va} & ::= & \underline{\textit{variations}} & \\
 & \mid \underline{G.e} & & \\
 & \mid \underline{G.e, Va} & & \\
v & ::= & \textit{values} & \\
 & \mid c & \textit{constants} & \text{e.g. } true, (), 1, 2, \ldots \\
 & \mid \textbf{fun}\ f\,x \Rightarrow e & \textit{functions} & \\
 & \mid \underline{(x)\{Va\}} & \underline{\textit{behavioural variations}} & \\
 & \mid \underline{F} & \underline{\textit{Datalog facts}} & \\
e & ::= & \textit{expressions} & \\
 & \ \ v & \textit{values} & \\
 & \mid x & \textit{variables} & \\
 & \mid \tilde{x} & \textit{parameters} & \\
 & \mid e_1\, e_2 & \textit{functional application} & \\
 & \mid \textbf{let}\ x\ =\ e_1\ \textbf{in}\ e_2 & \textit{declaration} & \\
 & \mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 & \textit{conditional} & \\
 & \mid \underline{\textbf{dlet}\ \tilde{x}\ =\ e_1\ \textbf{when}\ G\ \textbf{in}\ e_2} & \underline{\textit{context-dependent binding}} & \\
 & \mid \underline{\textbf{tell}(e_1)} & \underline{\textit{asserting facts}} & \\
 & \mid \underline{\textbf{retract}(e_1)} & \underline{\textit{retracting facts}} & \\
 & \mid \underline{\#(e_1, e_2)} & \underline{\textit{behavioural variation application}} & \\
 & \mid \underline{e_1 \cup e_2} & \underline{\textit{behavioural variation concatenation}} &
\end{array}
$$

In addition to the usual constructs (the new ones are underlined and in blue), our values include Datalog facts F and behavioural variations. Also, we introduce the set $\tilde{x} \in DynVar$ of *parameters*, i.e., variables assuming values depending on the properties of the running context, whereas Var are standard identifiers. Our COP constructs include behavioural variations $(x)\{Va\}$, each consisting of a variation Va, i.e. a list of expressions

If1

$$\frac{\rho \vdash C,\, e_1 \rightarrow C',\, e_1'}{\rho \vdash C,\, \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \rightarrow C',\, \textbf{if}\ e_1'\ \textbf{then}\ e_2\ \textbf{else}\ e_3}$$

If2

$$\frac{}{\rho \vdash C, \textbf{if}\ \mathit{true}\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \rightarrow C',\, e_2}$$

If3

$$\frac{}{\rho \vdash C,\, \textbf{if}\ \mathit{false}\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \rightarrow C',\, e_3}$$

let1

$$\frac{\rho \vdash C,\, e_1 \rightarrow C',\, e_1'}{\rho \vdash C, \textbf{let}\ x\ =\ e_1\ \textbf{in}\ e_2 \rightarrow C',\, \textbf{let}\ x\ =\ e_1'\ \textbf{in}\ \ e_2}$$

let2

$$\frac{}{\rho \vdash C,\, \textbf{let}\ x\ =\ v\ \textbf{in}\ e_2 \rightarrow C,\, e_2\{v/x\}}$$

App1

$$\frac{\rho \vdash C,\, e_1 \rightarrow C',\, e_1'}{\rho \vdash C,\, e_1\, e_2 \rightarrow C',\, e_1'\, e_2}$$

App2

$$\frac{\rho \vdash C,\, e_2 \rightarrow C',\, e_2'}{\rho \vdash C,\, (\textbf{fun}\ f x \Rightarrow e)\, e_2 \rightarrow C',\, (\textbf{fun}\ f x \Rightarrow e)\, e_2'}$$

App3

$$\frac{}{\rho \vdash C,\, (\textbf{fun}\ f x \Rightarrow e)\, v \rightarrow C,\, e\{v/x, (\textbf{fun}\ f x \Rightarrow e)/f\}}$$

Figure 4.3: Part of semantic rules for $\mathrm{ML}_{\mathrm{CoDa}}$

$G_1.e_1, \ldots, G_n.e_n$ guarded by Datalog goals G_i. The variable x can freely occur in the expressions e_i. At runtime, the first goal G_i satisfied by the context determines the expression e_i to be selected (*dispatching*). The **dlet** construct implements the context-dependent binding of a parameter $\tilde{x}$ to a variation Va. When $\tilde{x}$ is referred to, one of e_i is selected depending on the first goal G_i in the list the actual context satisfies. The **tell** and **retract** constructs update the context by asserting and retracting facts. The application of a behavioural variation $\#(e_1, e_2)$ applies e_1 to its argument e_2. To do that, the dispatching mechanism is triggered to query the context and to select from e_1 the expression to run, if any. The append operator $e_1 \cup e_2$ dynamically concatenates behavioural variations, so providing a further adaptation mechanism.

Note that we could add constructs for communication, resource manipulation and security as we did in Chapter 3 for ContextML, but we prefer to omit them to keep the formal development manageable.

4.3.2 Semantics of $\mathrm{ML}_{\mathrm{CoDa}}$

We endow $\mathrm{ML}_{\mathrm{CoDa}}$ with a small-step operational semantics. Recall (Section 4.1) that the evaluation of a program only starts after the virtual machine has completed the linking and verification phases.

For the Datalog evaluation we adopt the top-down standard semantics for stratified programs [CGT89]. Given a context C and a goal G, $C \vDash G\, \mathit{with}\, \theta$ means that there exists a substitution θ, replacing constants for variables, such that the goal G is satisfied in the context C.

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

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

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

$$\textsc{Dlet2}\quad \frac{}{\rho \vdash C,\, \mathbf{dlet}\ \ \tilde{x} = e_1\ \mathbf{when}\ G\ \ \mathbf{in}\ v \rightarrow C,\, v} \qquad \textsc{Dyvar}\quad \frac{\rho(\tilde{x}) = Va \qquad dsp(C,\, Va) = (e,\, \{\vec{c}/\vec{y}\})}{\rho \vdash C,\, \tilde{x} \rightarrow C,\, e\{\vec{c}/\vec{y}\}}$$

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

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

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

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

Figure 4.4: Semantic rules of the new constructs for adaptation of ML$_{\text{CoDa}}$

The $\mathrm{ML}_{\mathrm{CoDa}}$ semantics is inductively defined for expressions with no free variable, but possibly with free parameters. These will take a value in an environment ρ, i.e. a function mapping parameters to variations $DynVar \rightarrow Va$. Environments are updated as usual, given an environment ρ we define the update operation as follow

$$\rho[Va/\tilde{x}](\tilde{x}') = \begin{cases} Va & \tilde{x} = \tilde{x}' \\ \rho(\tilde{x}') & \text{otherwise} \end{cases}$$

A transition $\rho \vdash C,\, e \rightarrow C',\, e'$ says that in the environment ρ, the expression e is evaluated in the context C and reduces to e' changing the context C to C'. We assume that the initial configuration is $\rho_0 \vdash C,\, e_p$ where ρ_0 contains the bindings for all system parameters, and C results from the linking of the system context $\overline{C}$ and of the application context C_p, as illustrated in Figure 4.2.

The semantics rules are displayed in Figure 4.3 and Figure 4.4. Since most of them are inherited from ML, below we only comment on those for new constructs (Figure 4.4).

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).

Example 4.3.1. In the following we show the reduction of a **tell** construct: we apply the function `getAccessibilityOpt` to unit, assuming that `getCheckedOption` returns the fact `user_acc_opt(blind)`:

$$\begin{aligned} \rho \vdash & C,\, \texttt{setAccessibilityOpt}\,() \rightarrow \\ & C,\, \texttt{tell(getCheckedOption accRadioButton)} \rightarrow^{\star} \\ & C,\, \texttt{tell(user\_acc\_opt(blind))} \rightarrow C \cup \{\texttt{user\_acc\_opt(blind)}\},\, (). \end{aligned}$$

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 rule Dlet1 extends the environment ρ by appending $G.e_1$ in front of the existent binding for $\tilde{x}$. Then, e_2 is evaluated under the updated environment. Notice that the **dlet** reduction rules do not evaluate e_1 but only record it in the environment. The rule Dlet2 is standard: the **dlet** reduces to the value eventually computed by e_2.

The Par rule looks up the variation Va bound to $\tilde{x}$ in ρ. Then the dispatching mechanism selects the expression to which $\tilde{x}$ reduces.

Definition 4.3.1 (Dispatching Mechanism)**.** The dispatching mechanism of $\mathrm{ML}_{\mathrm{CoDa}}$ is defined on the line of the one of ContextML (Definition 2.2.3) and it is implemented by the partial function dsp, as

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

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 $\tilde{x}$ 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 since the program cannot adapt to the current context.

Example 4.3.2. Consider the function `setMainWindow` defined in Section 4.2 and apply it to a value w. Assume that the environment ρ binds the parameters `txt_label` and `vcanvas` as done in Section 4.2. Furthermore, assume that context C satisfies the goals $\leftarrow \neg\texttt{only\_speech}()$ and $\leftarrow \texttt{video}(\texttt{hd}), \neg\texttt{only\_text}()$ but not $\leftarrow \texttt{video}(\texttt{low}), \neg\texttt{only\_text}()$.

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

In the computation from the second to the third configuration, we retrieve the binding for `txt_label` and apply dsp to C and $\rho(\texttt{txt\_label})$:

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

where ι is the empty substitution. The same happens from the fourth to the fifth configuration: we apply dsp to C and $\rho(\texttt{vcanvas})$ obtaining $\texttt{getHDCanvas}()$.

The rules for $e_1 \cup e_2$ sequentially evaluate e_1 and e_2 until they reduce to behavioural variations (rules Append1 and Append2). Then, they are concatenated together by renaming bound variables to avoid name captures (rule Append3).

Example 4.3.3. As an example, consider the function `addDefault` of Section 4.2. 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):

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

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 is substituted for x. Also in this case the computation gets stuck, if the dispatching mechanism fails.

Example 4.3.4. As an example consider the function `getExhibitData` and apply it to unit. The computation is

$$
\begin{aligned}
&\rho \vdash C,\ \texttt{getExhibitData}\,() \to^{\star} C,\ \texttt{getRemoteData}\,\#(u, ()) \to^{\star}\\
&\quad C,\ \texttt{getRemoteData}(\texttt{receiveData}\,n)
\end{aligned}
$$

$(* \; n$ is returned by `getChannel` $*)$

If the context C satisfies the goal $\leftarrow$ `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`).

4.4 Type and Effect System

We now associate $\mathrm{ML}_{\mathrm{CoDa}}$ expressions with an annotated type and an history expression. The shape of history expression of $\mathrm{ML}_{\mathrm{CoDa}}$ are different from the ones of ContextML, although the underlying idea remains the same. During the verification phase the virtual machine uses this history expression to ensure that the dispatching mechanism will always succeed at runtime. First, we briefly present history expressions for $\mathrm{ML}_{\mathrm{CoDa}}$ and then the rules of our type and effect system.

4.4.1 History Expressions of $\mathrm{ML}_{\mathrm{CoDa}}$

Here, history expressions approximate the sequence of actions that a program may perform over the context at runtime, i.e., asserting/retracting facts and asking if a goal holds, as well as how behavioural variations will be "resolved".

The following syntax of history expressions is similar to the one introduced in Section 3.3 except for constructs (underlined and in <u>blue</u> below) to deal with the context:

$$\begin{array}{lll}
H \;\; ::= \; \epsilon & & \textit{empty} \\
\quad\;\; \mid h & & \textit{recursion variable} \\
\quad\;\; \mid \mu h.H & & \textit{recursion} \\
\quad\;\; \mid \underline{\textit{tell}\, F} & & \underline{\textit{asserting a fact } F} \\
\quad\;\; \mid \underline{\textit{retract}\, F} & & \underline{\textit{retracting a fact } F} \\
\quad\;\; \mid H_1 + H_2 & & \textit{non-deterministic sum} \\
\quad\;\; \mid H_1 \cdot H_2 & & \textit{sequence} \\
\quad\;\; \mid \underline{\Delta} & & \underline{\textit{abstract variation}} \\
\underline{\Delta} \;\; ::= & & \underline{\textit{abstract variations}} \\
\quad\;\; \mid \underline{\textit{ask}\, G.H \otimes \Delta} & & \underline{\textit{dispatch}} \\
\quad\;\; \mid \underline{\textit{fail}} & & \underline{\textit{failure}}
\end{array}$$

Here, the empty history expression ϵ abstracts programs which do not interact with the context; the "atomic" history expressions *tell* F and *retract* F are for the analogous expressions of $\mathrm{ML}_{\mathrm{CoDa}}$; Δ mimics our dispatching mechanism, where Δ is an *abstract variation*, defined as a list of history expressions, each element H_i of which is guarded by *ask* G_i; the other constructs are the similar to the ones of ContextML.

Given a context C, the behaviour of a history expression H is formalized by the transition system inductively defined in Figure 4.5. Configurations have the form $C, H \to C', H'$ meaning that H reduces to H' in the context C and yields the context C'. Most rules are similar to the ones in Section 3.3, and below we only comment on those dealing with the context.

$$\frac{}{C,\ \mathit{tell}\, F \rightarrow C \cup \{F\},\ \epsilon} \qquad \frac{}{C,\ \mathit{retract}\, F \rightarrow C \backslash \{F\},\ \epsilon} \qquad \frac{}{C,\ \epsilon \cdot H \rightarrow C,\ H}$$

$$\frac{C,\ H_1 \rightarrow C',\ H_1'}{C,\ H_1 \cdot H_2 \rightarrow C',\ H_1' \cdot H_2} \qquad \frac{C, H_1 \rightarrow C',\ H_1'}{C, H_1 + H_2 \rightarrow C',\ H_1'} \qquad \frac{C, H_2 \rightarrow C',\ H_2'}{C, H_1 + H_2 \rightarrow C',\ H_2'}$$

$$\frac{}{C, \mu h.H \rightarrow C, H[\mu h.H/h]} \qquad \frac{C \models G}{C,\ \mathit{ask}\, G.H \otimes \Delta \rightarrow C,\ H} \qquad \frac{C \not\models G}{C,\ \mathit{ask}\, G.H \otimes \Delta \rightarrow C,\ \Delta}$$

Figure 4.5: Semantics of History Expressions

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.

Note that differently from Section 3.3 we here are not concerned about the sequence of actions carried out on the initial context, but we are interested in the contexts which can be reached starting from it. For this reason, here the transition system has no labels and we introduce no language-theoretic semantics, as Definition 3.3.1. As we see later on, our loading time analysis provides us with an over-approximation of the reached contexts.

Example 4.4.1. Consider the initial context $C = \{F_2,\ F_5,\ F_8\}$ and the history expression

$$\begin{aligned} H_a = & \mathit{tell}\, F_1 \cdot \mathit{retract}\, F_2 \cdot \mathit{ask}\, F_8.\mathit{retract}\, F_5 \otimes \mathit{fail} + \\ & \mathit{ask}\, F_5.\mathit{retract}\, F_8 \otimes \mathit{ask}\, F_3.\mathit{retract}\, F_4 \otimes \mathit{fail}. \end{aligned}$$

For simplicity, we use a context and goals with only facts and we omit the symbols $\leftarrow$ from goals. A possible sequence of transitions for $C,\ H_a$ is:

$$\begin{aligned} & \{F_2,\ F_5,\ F_8\}, H_a \rightarrow^* \\ & \{F_1,\ F_2,\ F_5,\ F_8\}, \epsilon \cdot \mathit{retract}\, F_2 \cdot \mathit{ask}\, F_8.\mathit{retract}\, F_5 \otimes \mathit{fail} \rightarrow \\ & \{F_1,\ F_2,\ F_5,\ F_8\}, \mathit{retract}\, F_2 \cdot \mathit{ask}\, F_8.\mathit{retract}\, F_5 \otimes \mathit{fail} \rightarrow^* \\ & \{F_1,\ F_5,\ F_8\}, \mathit{ask}\, F_8.\mathit{retract}\, F_5 \otimes \mathit{fail} \rightarrow \\ & \{F_1,\ F_5,\ F_8\}, \mathit{retract}\, F_5 \rightarrow \\ & \{F_1,\ F_8\}, \epsilon \end{aligned}$$

When the reduction of H_a terminates, the empty history expression is reached, yielding the context $\{F_1,\ F_8\}$. As an example of failure, take $C' = \{F_2,\ F_5\}$ as initial context. In this case we reach the *fail* configuration as shown by the following reduction:

$$\{F_2,\ F_5\}, H_a \rightarrow^*$$

$$\begin{aligned}
&\{F_1,\ F_2,\ F_5\}, \epsilon \cdot \mathit{retract}\ F_2 \cdot \mathit{ask}\ F_8.\mathit{retract}\ F_5 \otimes \mathit{fail} \rightarrow \\
&\{F_1,\ F_2,\ F_5\}, \mathit{retract}\ F_2 \cdot \mathit{ask}\ F_8.\mathit{retract}\ F_5 \otimes \mathit{fail} \rightarrow^* \\
&\{F_1,\ F_5\}, \mathit{ask}\ F_8.\mathit{retract}\ F_5 \otimes \mathit{fail} \rightarrow \\
&\{F_1,\ F_5\}, \mathit{fail}
\end{aligned}$$

4.4.2 Typing rules

Here, we only give a logical presentation of our type and effect system, leaving the definition of an inference algorithm as a future work (see Section 5.1.1). 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, for example a light version of [MO84]. 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 variables x of G (e.g. in the Tvariation, Figure 4.7).

The syntax of types is

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

We have basic types (*int*, *bool*, *unit*), functional types, behavioural variations types, and facts. As in Section 3.3, some types are annotated for analysis reason. In the type $fact_\phi$, the set ϕ soundly contains the facts that an expression can be reduced to at runtime (see the semantic rules Tell2 and Retract2). In the type $\tau_1 \xrightarrow{K|H} \tau_2$ associated with a function f, the environment K is a precondition needed to apply f. Here, K stores the types and the abstract variations of parameters occurring inside the body of f. The history expression H is the latent effect of f, i.e. the sequence of actions which may be performed over the context during the function evaluation. Analogously, in the type $\tau_1 \xRightarrow{K|\Delta} \tau_2$ associated with the behavioural variation $bv = (x)\{Va\}$, K is a precondition for applying bv and Δ is an abstract variation representing the information that the dispatching mechanism uses at runtime to apply bv.

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

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

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

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

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

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

$$\textsc{Srefl}\ \frac{}{\tau \leq \tau} \qquad \textsc{Sfact}\ \frac{\phi \subseteq \phi'}{\mathit{fact}_{\phi} \leq \mathit{fact}_{\phi'}} \qquad \textsc{Sfun}\ \frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2' \qquad K \sqsubseteq K' \qquad H \sqsubseteq H'}{\tau_1 \xrightarrow{K|H} \tau_2 \leq \tau_1' \xrightarrow{K'|H'} \tau_2'}$$

$$\textsc{Sva}\ \frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2' \qquad K \sqsubseteq K' \qquad \Delta \sqsubseteq \Delta'}{\tau_1 \xRightarrow{K|\Delta} \tau_2 \leq \tau_1' \xRightarrow{K'|\Delta'} \tau_2'} \qquad \textsc{Tconst}\ \frac{}{\Gamma; K \vdash c \ : \ \tau_c \triangleright \epsilon} \qquad \textsc{Tvar}\ \frac{\Gamma(x) = \tau}{\Gamma; K \vdash x \ : \ \tau \triangleright \epsilon}$$

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

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

$$\textsc{Tabs}\ \frac{\Gamma, x : \tau_1, f : \tau_1 \xrightarrow{K'|H} \tau_2; K' \vdash e \ : \ \tau_2 \triangleright H}{\Gamma; K \vdash \textbf{fun } f x \Rightarrow e \ : \ \tau_1 \xrightarrow{K'|H} \tau_2 \triangleright \epsilon}$$

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

Figure 4.6: Typing rules for standard ML constructs

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.

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 $H_2 = H_1 + H_3$. We lift the operator $\otimes$ to abstract variation Δ as follows $\Delta_1 \sqsubseteq \Delta_2$ iff $\exists \Delta_3$ such that $\Delta_2 = \Delta_1 \otimes \Delta_3$ (note that we assume $\mathit{fail} \otimes \Delta = \Delta$, so the concatenation Δ_2 has an single trailing term *fail*); $K_1 \sqsubseteq K_2$ iff $(\,(\tilde{x}, \tau_1, \Delta_1) \in K_1$ implies $(\tilde{x}, \tau_2, \Delta_2) \in K_2 \wedge \tau_1 \leq \tau_2 \wedge \Delta_1 \sqsubseteq \Delta_2\,)$.

Most of the rules of our type and effect system are inherited from those of ML and from those presented in Section 3.3 (Figure 4.6). Those for the new constructs are in Figure 4.7. Comments on the typing rules for new constructs are in order.

We have rules for subtyping and subeffecting (Figure 4.6, top). As expected these rules say that subtyping relation is reflexive (rule Srefl); that a type fact_{ϕ} is a subtype of a type $\mathit{fact}_{\phi'}$ 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).

TSUB

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

TFACT

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

TPAR

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

TTELL

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

TRETRACT

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

TVARIATION

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

TVAPP

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

TAPPEND

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

(TDLET)

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

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

Figure 4.7: Typing rules for new constructs

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 (or Tretract) asserts that the expression $\mathbf{tell}(e)$ (or $\mathbf{retract}(e)$) 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 $\mathbf{tell}\,F$ (or $\mathbf{retract}\,F$) where F is any of the facts in the type of e.

Example 4.4.2. Take the body of the function `setAccessibilityOpt` of Section 4.2. We know that the function `getCheckedOption` (call it e_1) returns either the 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 up the type and the effect of the parameter $\tilde{x}$ in the environment K.

Example 4.4.3. Consider type-checking the application `addComponent window vcanvas` in the body of function `setMainWindow` in Section 4.2. Assume that `addComponent` has type $\mathtt{window\_t} \xrightarrow{H_a} \mathtt{component\_t} \xrightarrow{H_b} unit$ and empty effect, `window` has type `component_t` and empty effect and that the binding for `vcanvas` in K is $(\mathtt{component\_t}, \Delta)$ where

$$\Delta = ask\ \leftarrow \mathtt{video(hd)}, \neg\mathtt{only\_text()}.\ H_1 \otimes ask\ \leftarrow \mathtt{video(low)}.\ \mathtt{H}_2 \otimes fail, \neg\mathtt{only\_text()}.$$

The application `addComponent 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 $\vec{y_i}$ 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.

Example 4.4.4. As an example of typing of a behavioural variation, take the one defined inside the body of function `getExhibitData` of Section 4.2 (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

$$\Delta = ask\,\mathtt{direct\_comm()}.H_1 \otimes ask\,\mathtt{use\_qr\_code()}, \mathtt{camera(on)}.H_2 \otimes fail$$

and the effect will be empty.

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

Example 4.4.5. Take the above behavioural variation bv_1, which has type $unit \xrightarrow{K'|\Delta} \tau$. Assume having 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 $H_1 \cdot H_2$.

Example 4.4.6. Consider again the above bv_1 and its type $int \xrightarrow{K'|\Delta} \tau$; consider also the body of the function `addDefault` of Section 4.2 and 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 $\vec{y}$ of the goal G. Also, e_2 has to type-check in an environment K extended with the information for the parameter $\tilde{x}$. The type and the effect for the overall **dlet** expression are the same of e_2.

Our type and effect system ensures us that no type error occurs at runtime and that the effect is an over-approximation of the actions performed over the context. Our analysis is sound with respect to the operational semantics. To concisely state our formal results, it is convenient to introduce the following technical definitions. The following definition establishes a relation among the dynamic environment ρ used in the semantics and the typing environments Γ and K.

Definition 4.4.1 (Typing dynamic environment)**.** Given the type environments Γ and K, we say that the dynamic environment ρ has type K under Γ (in symbols $\Gamma \vdash \rho : K$) iff $dom(\rho) \subseteq dom(K)$ and $\forall \tilde{x} \in dom(\rho)\,.\ \rho(x) = G_1.e_1, \ldots, G_n.e_n\ K(\tilde{x}) = (\tau, \Delta)$ and $\forall i \in \{1, \ldots, n\}\,.\ \gamma(G_i) = \vec{y}_i : \vec{\tau}_i\ \Gamma, \vec{y}_i : \vec{\tau}_i; K \vdash e_i : \tau' \rhd H_i$ and $\tau' \leq \tau$ and $\bigotimes_{i \in \{1,\ldots,n\}} G_i.H_i \sqsubseteq \Delta$.

Definition 4.4.2. Given H_1, H_2 then $H_1 \preccurlyeq H_2$ iff one of the following case holds

(a) $H_1 \sqsubseteq H_2$;

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

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

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

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

Lemma 4.4.1 (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 $\Gamma; K \vdash e_s : \tau \rhd H_s$ and $\rho \vdash C, e_s \to C', e'_s$ then $\Gamma; K \vdash e'_s : \tau \rhd H'_s$ and there exist $\overline{H}$ such that $\overline{H} \cdot H'_s \preccurlyeq H_s$ and $C, \overline{H} \cdot H'_s \to^\star C', H'_s$.

The Progress Lemma assumes that the effect H is *viable* (see Definition 4.6.2), namely it does not reach *fail*, because the dispatching mechanism succeeds at runtime. To guarantee the viability we have defined the analysis of Section 4.6. In the following statements we write $\rho \vdash C,\, e \not\rightarrow$ to intend that there exists no transition.

Lemma 4.4.2 (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 $\rho \vdash C,\, e_s \not\rightarrow$ and H is viable for C, i.e. $C,\, H_s \not\rightarrow^{+} C',\, fail$, 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.

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

The following theorem ensures the correctness of our approach.

Theorem 4.4.3 (Correctness)**.** *Let e_s be a closed expression such that $\Gamma; K \vdash e_s : \tau \triangleright H_s$; let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e_s, and that $\Gamma \vdash \rho : K$; and let C be a context such that H_s is viable i.e. $C, H_s \not\rightarrow^{+} C',\, fail$. Then either the computation terminates yielding a value ($\rho \vdash C, e_s \rightarrow^{\star} C'', v$) or it diverges, but it never gets stuck.*

4.5 Properties of the Type and Effect System

In this section we prove Lemma 4.4.1, Lemma 4.4.2 and Corollary 1. We start giving some technical lemmata and definitions useful for the proofs. Roughly, the formal development follows the same schema of the ones presented in Chapter 2 and Chapter 3.

Definition 4.5.1 (Capture avoiding substitutions)**.** Given the expression e,e' and the variable x we define $e\{e'/x\}$ as following

$$
\begin{aligned}
&c\{e'/x\} = c\\
&F\{e'/x\} = F\\
&(\lambda_f x'.e)\{e'/x\} = \lambda_f x'.e\{e'/x\}\\
&\qquad\qquad\qquad \text{if } f \neq x \wedge x' \neq x \wedge f, x' \notin FV(e')\\
&(x')\{G_1.e_1, \dots, G_n.e_n\}\{e'/x\} =\\
&\qquad (x')\{G_1.e_1\{e'/x\}, \dots, G_n.e_n\{e'/x\}\}\\
&\qquad\qquad\qquad \text{if } x \neq x' \wedge x \in \bigcup_{i \in \{1,\dots,n\}} FV(G_i) \wedge\\
&\qquad\qquad\qquad \left(\{x'\} \cup \bigcup_{i \in \{1,\dots,n\}} FV(G_i) \right) \cap FV(e') = \emptyset\\
&x\{e'/x\} = e'
\end{aligned}
$$

$$
\begin{aligned}
& x'\{e'/x\} = x' \qquad\qquad \text{if } x \neq x' \\
& (e_1\, e_2)\{e'/x\} = e_1\{e'/x\}\, e_2\{e'/x\} \\
& (e_1\, op\, e_2)\{e'/x\} = e_1\{e'/x\}\, op\, e_2\{e'/x\} \\
& (if\, e_1\, then\, e_2\, else\, e_3)\{e'/x\} = \\
& \qquad\qquad if\, e_1\{e'/x\}\, then\, e_2\{e'/x\}\, else\, e_3\{e'/x\} \\
& (tell(e))\{e'/x\} = tell(e\{e'/x\}) \\
& (retract(e))\{e'/x\} = retract(e\{e'/x\}) \\
& (e_1 \cup e_2)\{e'/x\} = e_1\{e'/x\} \cup e_2\{e'/x\} \\
& \#(e_1, e_2)\{e'/x\} = \#(e_1\{e'/x\}, e_2\{e'/x\}) \\
& (let\, x' \ = \ e_1\, in\, e_2)\{e'/x\} = let\, x' \ = \ e_1\{e'/x\}\, in\, e_2\{e'/x\} \\
& \qquad\qquad\qquad\qquad \text{if } x \neq x' \wedge x' \in FV(e') \\
& (dlet\, \tilde{x} \ = \ e_1\, when\, G\, in\, e_2)\{e'/x\} = \\
& \qquad\qquad dlet\, \tilde{x} \ = \ e_1\{e'/x\}\, when\, G\, in\, e_2\{e'/x\} \\
& \qquad\qquad\qquad\qquad \text{if } x \notin FV(G) \wedge FV(G) \cap FV(e') = \emptyset.
\end{aligned}
$$

Lemma 4.5.1. *If $\Gamma \vdash \rho \ : \ K$ and $K \sqsubseteq K'$ then $\Gamma \vdash \rho \ : \ K'$.*

Proof. The thesis follows from Definition 4.4.1 and that of $K \sqsubseteq K'$. □

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

Lemma 4.5.2. *Given K and a parameter $\tilde{x}$*

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

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

Lemma 4.5.3. *If $\Gamma \vdash \rho \ : \ K$ and G and e are such that $\gamma(G) = \vec{y} : \vec{\tau}$ and $\Gamma, \vec{y} : \vec{\tau}; K \vdash e \ : \ \tau \triangleright H$*

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

Proof. The thesis follows by using the Definition 4.4.1 and that of $K \sqsubseteq K'$. □

Lemma 4.5.4. *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 4.5.5 (Weakening)**.**

1. *if $\Gamma; K \vdash e \ : \ \tau \triangleright H$ and x is a variable $x \notin dom(\Gamma)$ then $\Gamma, x \ : \ \tau'; K \vdash e \ : \ \tau \triangleright H$*

2. *if* $\Gamma; K \vdash e \,:\, \tau \triangleright H$ *and* $\tilde{x}$ *is a parameter* $\tilde{x} \notin dom(K)$ *then* $\Gamma; K, (\tilde{x}, \tau, \Delta) \vdash e \,:\, \tau \triangleright H$

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

Lemma 4.5.6 (Inclusion)**.**

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

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 Lemma 4.5.5 the thesis holds.
2. Similar to previous case.

□

Lemma 4.5.7 (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* $v =$ ***fun*** $f x \Rightarrow e$
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, $(x)\{Va\}$, **fun** $f x \Rightarrow e$ 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 4.5.8 (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* $\forall i \in \{1, \ldots, n\}$ $\Gamma, x : \tau_1, \vec{y}_i : \vec{\tau}_i; K \vdash e_i \,:\, \tau_2 \triangleright H_i$ *where* $\vec{y}_i : \vec{\tau}_i = \gamma(G_i)$

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 4.5.6.

2. By the premise of the rule TVARIATION we know that $\forall i \in \{1, \ldots, n\}\ \Gamma, \vec{y_i} : \vec{\tau_i}; K' \vdash e_i : \tau_2 \triangleright H_i$ and $\vec{y_i} : \vec{\tau_i} = \gamma(G_i)$ and $\Delta = \bigotimes_{i \in \{1,\ldots,n\}} ask\, G_i.H_i$. Since $K' \sqsubseteq K$ the thesis follows by Lemma 4.5.6(2).

□

Lemma 4.5.9 (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 4.4.1 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 $\Gamma, x : \tau'; K \vdash e_i : \tau_1 \overset{K'|\Delta_i}{\Longrightarrow} \tau_2 \triangleright H_i$ 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 and by Definition 4.4.1 we can 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 $\Gamma, x : \tau'; K \vdash e_1 : \tau_1 \overset{K'|\Delta}{\Longrightarrow} \tau_2 \triangleright H_1$ and $\Gamma, x : \tau'; K \vdash e_2 : \tau_1 \triangleright H_1$ 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 4.4.1 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 $\forall i \in \{1, \ldots, n\}\ \Gamma, x : \tau', x' : \tau_1, \vec{y_i} : \vec{\tau_i}; K' \vdash e_i : \tau_2 \triangleright H_i$ where $\vec{y_i} : \vec{\tau_i} = \gamma(G_i)$, $\Delta = \bigotimes_{i \in \{1,\ldots,n\}} ask\, G_i.H_i$. By Lemma 4.5.4 $\forall i \in \{1, \ldots, n\}\ \Gamma, x' : \tau_1 \vec{y_i} : \vec{\tau_i}, x : \tau'; K' \vdash e_i : \tau_2 \triangleright H_i$. 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 4.4.1 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 $\Gamma, x : \tau', \vec{y} : \vec{\tau}; K \vdash e_1 : \tau_1 \triangleright H_1$ and $\Gamma, x : \tau'; K, (\tilde{x}, \tau_1, \Delta) \vdash e_2 : \tau \triangleright H$ with $\vec{y} : \vec{\tau} = \gamma(G)$. By Lemma 4.5.4

$\Gamma, \vec{y} : \vec{\tau}, x : \tau'; K \vdash e_1 : \tau_1 \triangleright H_1$. By using the induction hypothesis we can claim that $\Gamma; K \vdash \mathit{dlet}\, \tilde{x} = e_1\{v/x\}\, \mathit{when}\, G\, \mathit{in}\, e_2\{v/x\} : \tau \triangleright H$ and by Definition 4.4.1 $\Gamma; K \vdash (\mathit{dlet}\, \tilde{x} = e_1\, \mathit{when}\, G\, \mathit{in}\, e_2)\{v/x\} : \tau \triangleright H$.

- rule Tconst, Tfact, Tdvar Since $e\{v/x\} = e$ by Definition 4.4.1 the result $\Gamma; K \vdash e : \tau \triangleright H$ is immediate, when $e = c$, $e = F$ and $e = \tilde{x}$.

- The other cases are standard.

□

Lemma 4.5.10. *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 4.5.9 □

Lemma 4.5.11. *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 4.5.12. *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. □

Definition 4.4.2. Given H_1, H_2 then $H_1 \preccurlyeq H_2$ iff one of the following case holds

(a) $H_1 \sqsubseteq H_2$;

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

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

Lemma 4.5.13. *Given the histories expressions* H_1*,* H_2*,* H_3 *and* H_4 *such that* $H_1 \preccurlyeq H_2$ *and* $H_4 = H_3 + \tilde{H}$ *then* $H_1 \cdot H_3 \preccurlyeq H_2 \cdot H_4$

Proof. There are three cases according to Definition 4.4.2.

1. $H_2 = H_1 + \overline{H}$

$$\begin{aligned} H_2 \cdot H_4 &= (H_1 + \overline{H}) \cdot (H_3 + \tilde{H}) \\ &= H_1 \cdot H_3 + H_1 \cdot \tilde{H} + \overline{H} \cdot H_3 + \overline{H} \cdot \tilde{H} \\ &\succcurlyeq H_1 \cdot H_3 \end{aligned}$$

2. $H_2 = \tilde{H} \cdot H_1$

$$\begin{aligned} H_2 \cdot H_4 &= \tilde{H} \cdot H_1 \cdot (H_3 + \tilde{H}) \\ &= \tilde{H} \cdot (H_1 \cdot H_3 + H_1 \cdot \tilde{H}) \\ &\succcurlyeq H_1 \cdot H_3 + H_1 \cdot \tilde{H} \\ &\succcurlyeq H_1 \cdot H_3 \end{aligned}$$

3. $H_2 = \bigotimes_{i\in\{1,\ldots,n\}} ask\, G_i.H_i' \otimes fail$ and $H_1 = H_i$ for some i

$$\begin{aligned} H_2 \cdot H_4 &= \left(\bigotimes_{i\in\{1,\ldots,n\}} ask\, G_i.H_i \otimes fail \right) \cdot H_4 \\ &= \bigotimes_{i\in\{1,\ldots,n\}} ask\, G_i.H_i \cdot H_4 \otimes fail \\ &\succcurlyeq H_i \cdot H_4 \\ &= H_i \cdot (H_3 + \tilde{H}) \\ &= H_i \cdot H_3 + H_i \cdot \tilde{H} \\ &\succcurlyeq H_i \cdot H_3 \\ &= H_1 \cdot H_3 \end{aligned}$$

□

Lemma 4.4.1 (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 $\rho \vdash C,\, e_s \rightarrow C',\, e_s'$ then $\Gamma;\, K \vdash e_s'\ :\ \tau \triangleright H_s'$ and there exist $\overline{H}$ such that $\overline{H} \cdot H_s' \preccurlyeq H_s$ and $C, \overline{H} \cdot H_s' \rightarrow^{\star} C',\, H_s'$.

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

- 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 it cannot be the case that $\rho \vdash C,\, e_s \rightarrow C',\, e_s'$ for any e_s' so the theorem holds vacuously.

- rule Ttell $\quad e_s = tell(e') \quad \Gamma; K \vdash e'\ :\ fact_\phi \triangleright H \quad H_s = H \cdot \sum_{F\in\phi} tell\, F$
  We have only two rules by which $\rho \vdash C,\, e_s \rightarrow C',\, e_s'$ can be derived.

  - rule tell1
    We know that e' is an expression and $e_s' = \mathbf{tell}(e'')$ and $\rho \vdash C,\, e' \rightarrow C',\, e''$ 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 there is a $\overline{H}''$ such that $H \sqsupseteq \overline{H}'' \cdot H''$ and $C,\, \overline{H}'' \cdot H'' \rightarrow^{\star} C',\, H''$. By using the rule Ttell we can

conclude that $\Gamma; K \vdash e'_s : \mathit{unit} \triangleright H'_s$ and $H'_s = H'' \cdot \sum_{F \in \phi} \mathit{tell}\, F$. It remains to prove that $H \cdot \sum_{F \in \phi} \mathit{tell}\, F \succcurlyeq \overline{H}'' \cdot H'' \cdot \sum_{F \in \phi} \mathit{tell}\, F$. Since $H \succcurlyeq \overline{H}'' \cdot H''$ the thesis follows by Lemma 4.5.13.

- rule TELL2

  We now that $e' = F$, $e'_s = ()$ and $C' = C \cup \{F\}$. We need to prove that $\Gamma; K \vdash e'_s : \mathit{unit} \triangleright H''$, but from the rule TCONST we know that this holds with $H'' = \epsilon$. It remains to show that there is $\overline{H}$ such that $H_s \sqsupseteq \overline{H} \cdot \epsilon$ and $C, \overline{H} \cdot \epsilon \rightarrow^\star C', \epsilon$. Choosing $\overline{H} = \mathit{tell}\, F$ and by using Lemma 4.5.7(4), the equational properties of history expressions (Definition 3.3.2) Lemma 4.5.13 we can conclude $H \cdot \sum_{F \in \phi} \mathit{tell}\, F \succcurlyeq H \cdot \mathit{tell}\, F \succcurlyeq \epsilon \cdot \mathit{tell}\, F = \mathit{tell}\, F = \mathit{tell}\, F \cdot \epsilon$ and $C, \mathit{tell}\, F \cdot \epsilon \rightarrow^\star C', \epsilon$.

- rule TRETRACT

  Similar to TTELL rule (**retract** substitutes **tell**)

- rule TAPPEND $\quad e_s = e_1 \cup e_2 \quad \Gamma; K \vdash e_1 : \tau_1 \xRightarrow{K'|\Delta_1} \tau_2 \triangleright H_1 \quad \Gamma; K \vdash e_2 : \tau_1 \xRightarrow{K'|\Delta_2} \tau_2 \triangleright H_2$

  There are three rules only by which $\rho \vdash C, e_s \rightarrow C', e'_s$ can be derived.

  - rule APPEND1

    We know that e_1 and e_2 are expressions 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 $H_1 \succcurlyeq \overline{H}'_1 \cdot H'_1$ and $C, \overline{H}'_1 \cdot H'_1 \rightarrow^\star C', H'_1$. By applying the TAPPEND rule we can conclude that $\Gamma; K \vdash e'_1 \cup e_2 : \tau_1 \xRightarrow{K'|\Delta_1 \otimes \Delta_2} \tau_2 \triangleright H'_1 \cdot H_2$ and since $H_1 \succcurlyeq \overline{H}'_1 \cdot H'_1$, $H_1 \cdot H_2 \succcurlyeq \overline{H}'_1 \cdot H'_1 \cdot H_2$ follows by Lemma 4.5.13.

  - 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 $H_2 \succcurlyeq \overline{H}'_2 \cdot H'_2$ and $C, \overline{H}'_2 \cdot H'_2 \rightarrow^\star C', H'_2$. By applying Lemma 4.5.11 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 $\Gamma; K \vdash (x)\{Va_1\} \cup e'_2 : \tau_1 \xRightarrow{K'|\Delta_1 \otimes \Delta_2} \tau_2 \triangleright \epsilon \cdot H'_2$. $H_1 \cdot H_2 \succcurlyeq \overline{H}'_2 \cdot (\epsilon \cdot H'_2)$ holds by applying the equational theory and Lemma 4.5.13.

  - rule APPEND3

    We know that e_s is

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

    and that e'_s is

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

    By the premise of the rule TVARIATION, we also know that $\forall i \in \{1, \ldots, n\}$ we have $\Gamma, x : \tau_1, \vec{y}_i : \vec{\tau}_i; K' \vdash e_i : \tau_2 \triangleright H_i$ and $\forall j \in \{1, \ldots, m\}$ we have $\Gamma, y :$

$\tau_1, \vec{y}_j : \vec{\tau}_j; K' \vdash e'_j : \tau_2 \triangleright H_j$. By Lemma 4.5.10 it holds that $\forall i \in \{1, \ldots, n\}$ $\Gamma, z : \tau_1, \vec{y}_i : \vec{\tau}_i; K' \vdash e_i\{z/x\} : \tau_2 \triangleright H_i$ and $\forall j \in \{1, \ldots, m\}$ $\Gamma, z : \tau_1, \vec{y}_j : \vec{\tau}_j; K' \vdash e'_j\{z/x\} : \tau_2 \triangleright H_j$. So by applying the rule Tvariation for all judgements indexed by i and j we can conclude that $\Gamma; K \vdash e'_s : \tau_1 \xRightarrow{K'|\Delta_1 \otimes \Delta_2} \tau_2 \triangleright \epsilon$. Since $H_s \succcurlyeq \epsilon$ and $C, \epsilon \rightarrow^\star C, \epsilon$ the thesis holds.

- rule Tvapp $\quad e_s = \#(e_1, e_2) \quad \Gamma; K \vdash e_1 : \tau_1 \xRightarrow{K'|\Delta} \tau_2 \triangleright H_1 \quad \Gamma; K \vdash e_2 : \tau_1 \triangleright H_2$ $K' \sqsubseteq K$

  There are three rules only by which $\rho \vdash C, e_s \rightarrow C', e'_s$ 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 $H_1 \succcurlyeq \overline{H}'_1 \cdot H'_1$ and $C, \overline{H}'_1 \cdot H'_1 \rightarrow^\star C', H'_1$. By Tvapp rule we have $\Gamma; K \vdash e'_s : \tau_2 \triangleright H'_1 \cdot H_2 \cdot \Delta$ and by Lemma 4.5.13 we can conclude $H_1 \cdot H_2 \cdot \Delta \succcurlyeq \overline{H}'_1 \cdot H'_1 \cdot H_2 \cdot \Delta$.

  - rule vapp2

    We know that $e'_s = \#((x)\{Va\}, e'_2)$. By using Lemma 4.5.11 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 $H_2 \succcurlyeq \overline{H}'_2 \cdot H'_2$ and $C, \overline{H}'_2 \cdot H'_2 \rightarrow^\star C', H'_2$. By Tvapp $\Gamma; K \vdash e'_s : \tau_2 \triangleright \epsilon \cdot H'_2 \cdot \Delta$ holds. Since $\overline{H}'_2 \cdot \epsilon \cdot H'_2 = \overline{H}'_2 \cdot H'_2$ by the equational theory, we can conclude by Lemma 4.5.13 $H_1 \cdot H_2 \cdot \Delta \succcurlyeq H_2 \cdot \Delta \succcurlyeq \overline{H}'_2 \cdot H'_2 \cdot \Delta = \epsilon \cdot \overline{H}'_2 \cdot H'_2 \cdot \Delta$.

  - rule vapp3

    We know that $e_s = \#((x)\{Va\}, v)$ where $Va = G_1.e_1, \ldots, G_n.e_n$, $e'_s = e_j\{v/x, \vec{c}/\vec{y}\}$ for $j \in \{1, \ldots, n\}$ and $\rho \vdash C, e_s \rightarrow C, e'_s$. From our hypothesis and from Lemma 4.5.8(2) we have that for all $i \in \{1, \ldots, n\}$ it holds $\Gamma, x : \tau_1, \vec{y}_i : \vec{t}_i; K \vdash e_i : \tau_2 \triangleright H_i$. By Lemma 4.5.11 we also know that $\Gamma; K \vdash v : \tau_1 \triangleright \epsilon$. So by Lemma 4.5.9 we have that for $i \in \{1, \ldots, n\}$ $\Gamma; K \vdash e_i\{v/x, \vec{\tau}/\vec{y}\} : \tau \triangleright H_i$. Since $\epsilon \cdot H_i = H_i$ and since $H_1 \cdot H_3 \cdot \Delta \succcurlyeq \Delta \succcurlyeq H_i$ for all $i \in \{1, \ldots, n\}$ the thesis holds with $\overline{H} = \epsilon$.

- rule Tdlet $\quad \gamma(G) = \vec{y} : \vec{\tau} \quad \Gamma, \vec{y} : \vec{\tau}; K \vdash e_1 : \tau_1 \triangleright H_1 \quad \Gamma; K_{\tilde{x}}, (\tilde{x}, \tau_1\, \Delta') \vdash e_2 : \tau \triangleright H$

  If the last rule in the derivation is Tdlet we know that $\Delta' = askG.H_1$ when $\tilde{x} \notin dom(K)$ or $\Delta' = askG.H_1 \otimes \Delta$ when $K(\tilde{x}) = (\tau_1, \Delta)$. There are two rules by which $\rho \vdash C, e_s \rightarrow C', e'_s$ can be derived.

  - rule dlet1

    We know that $e'_s = dlet\, \tilde{x} = e_1\, when\, G\, in\, e'_2$ and $\rho' \vdash C, e_2 \rightarrow C', e'_s$ with $\rho' = \rho[G.e, \rho(\tilde{x})/\tilde{x}]$. By Lemma 4.5.1 $\Gamma \vdash \rho : K'$ with $K' = K_{\tilde{x}}, (\tilde{x}, \tau, \Delta')$ and by Lemma 4.5.2 we know that $\Gamma \vdash \rho' : K'$. So by induction hypothesis $\Gamma; K' \vdash e'_2\, \tau \triangleright H'$ with $H \succcurlyeq \overline{H}' \cdot H'$ and $C, \overline{H}' \cdot H' \rightarrow^\star C', H'$. 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 $\rho \vdash C, e_s \rightarrow C, e'_s$. By hypothesis we know that

$\Gamma; K_{\tilde{x}}, (\tilde{x}, \tau_1, \Delta') \vdash v : \tau \triangleright H$ and by the Lemma 4.5.12 we have $\Gamma; K \vdash v : \tau \triangleright H$ and the thesis follows by choosing $\overline{H} = \epsilon$.

- rule TDVAR $\quad K(\tilde{x}) = (\tau, \Delta)$
  We assume that $\Delta = \bigotimes_{i \in \{1,\dots,n\}} \mathit{ask}\, G_i.H_i \otimes \mathit{fail}$. We have to prove that if $\rho \vdash C, \tilde{x} \to C', e$ then $\Gamma; K \vdash e, : \tau \triangleright H'$. By the premise of the rule we know that $\rho(\tilde{x}) = G_1.e_1, \dots, G_n.e_n$ and that there exists a $j \in \{1, \dots, n\}$ such that $e = e_j$. Since $\Gamma \vdash \rho : K$ we have that for all $i \in \{1, \dots, n\}$ it holds $\Gamma, \vec{y}_i : \vec{t}_i \vdash e_i : \tau \triangleright H_i$ where $\gamma(G_i) = \vec{y}_i : \vec{t}_i$ and by Lemma 4.5.9 we conclude that $\Gamma; K \vdash e_i\{\vec{t}_i / \vec{y}_i\} : e_i \triangleright H_i$ for all $i \in \{1, \dots, n\}$. Since $\epsilon \cdot H_i = H_i$ and $\bigotimes_{\forall i \in \{1,\dots,n\}} \mathit{ask}\, G_i.H_i \succcurlyeq H_i$ for all $i \in \{1, \dots, n\}$ the thesis holds with $\overline{H} = \epsilon$.

- rule TAPP $\quad \Gamma; K \vdash e_1 : \tau_1 \xrightarrow{K'|H_3} \tau_2 \triangleright H_1 \quad \Gamma; K \vdash e_2 : \tau_1 \triangleright H_2 \quad K' \sqsubseteq K$
  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 $H_1 \succcurlyeq \overline{H}'_1 \cdot H'_1$ and $C, \overline{H}'_1 \cdot H'_1 \to^\star C', H'_1$. 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 4.5.13 we can conclude $H_1 \cdot H_2 \cdot H_3 \succcurlyeq \overline{H}'_1 \cdot H'_1 \cdot H_2 \cdot H_3$.
  - rule APP2
    We know that $e'_s = (\lambda_f x.e)\, e_2$. By using Lemma 4.5.11 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 $H_2 \succcurlyeq \overline{H}'_2 \cdot H'_2$ and $C, \overline{H}'_2 \cdot H'_2 \to^\star C', H'_2$. By TAPP $\Gamma; K \vdash e'_s : \tau_2 \triangleright \epsilon \cdot H'_2 \cdot H_3$ holds. Since $\overline{H}'_2 \cdot \epsilon \cdot H'_2 = \overline{H}'_2 \cdot H'_2$ by the equational theory, we can conclude by Lemma 4.5.13 $H_1 \cdot H_2 \cdot H_3 \succcurlyeq H_2 \cdot H_3 \succcurlyeq \overline{H}'_2 \cdot H'_2 \cdot H_3 = \epsilon \cdot \overline{H}'_2 \cdot H'_2 \cdot H_3$.
  - rule APP3
    We know that $e'_s = e\{v/x, (\lambda_f x.e)/f\}$ and $\rho \vdash C, e_s \to C, e'_s$. We prove that $\Gamma; K \vdash e\{v/x, (\lambda_f x.e)/f\} : \tau_2 \triangleright H_3$. By Lemma 4.5.11 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 4.5.8 we conclude that $\Gamma, x : \tau_1, f : \tau_1 \xrightarrow{K'|H_3} \tau_2; K \vdash e : \tau_2 \triangleright H_3$. The thesis follows because we have that $\Gamma; K \vdash e\{v/x, (\lambda_f x.e)/f\} : \tau_2 \triangleright H_3$ by Lemma 4.5.9 and that $H_1 \cdot H_2 \cdot H_3 \succcurlyeq \epsilon \cdot H_3$ and $C, \epsilon \to^\star C, \epsilon$.

- rule TLET $\quad e_s = \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \quad \Gamma; K \vdash e_1 : \tau_1 \triangleright H_1 \quad \Gamma, x : \tau_1; K \vdash e_2 : \tau_2 \triangleright H_2$
  There are only two rules by which $\rho \vdash C, e_s \to C', e'_s$ can be derived.

  - rule LET1
    We know that $e'_s = \textbf{let}\ x = e'_1\ \textbf{in}\ e_2$. By the induction hypothesis we have that $\Gamma; K \vdash e'_1 : \tau_1 \triangleright H'_1$ and $H_1 \succcurlyeq \overline{H}'_1 \cdot H'_1$ and $\Gamma; K \vdash e'_s : \tau_2 \triangleright H'_1 \cdot H_2$, since $H_1 \succcurlyeq \overline{H}'_1 \cdot H'_1$ from Lemma 4.5.13 we can conclude that $H_1 \cdot H_2 \succcurlyeq \overline{H}'_1 \cdot H'_1 \cdot H_2$ holds.

- rule LET2
  We know that $e'_s = e_2\{v/x\}$, $\rho \vdash C,\, e_s \rightarrow C,\, e'_s$, $\Gamma; K \vdash v \,:\, \tau_1 \triangleright H_1$ and $\Gamma, x : \tau_1; K \vdash e_2 \,:\, \tau_2 \triangleright H_2$. By Lemma 4.5.11 $\Gamma; K \vdash v \,:\, \tau_1 \triangleright \epsilon$ and by Lemma 4.5.9 $\Gamma; K \vdash e_2\{v/x\} \,:\, \tau_2 \triangleright H_2$. The thesis follows since by Lemma 4.5.13 $H_1 \cdot H_2 \succcurlyeq \epsilon \cdot H_2$ and $C,\, \epsilon \cdot H_2 \rightarrow^\star C,\, H_2$.

- rule TIF $\quad e_s = \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \quad \Gamma; K \vdash e_1 \,:\, bool \triangleright H_1 \quad \Gamma; K \vdash e_2 \,:\, \tau \triangleright H_2$ $\Gamma; K \vdash e_3 \,:\, \tau \triangleright H_3$
  There are three rules only by which $\rho \vdash C,\, e_s \rightarrow C',\, e'_s$ can be derived.

  - rule IF1
    We know that $e'_s = \textbf{if } e'_1 \textbf{ then } e_2 \textbf{ else } e_3$. By using the induction hypothesis we have that $\Gamma; K \vdash e'_1 \,:\, bool \triangleright H'_1$ with $H_1 \succcurlyeq \overline{H}'_1 \cdot H'_1$ and $C,\, \overline{H}'_1 \cdot H'_1 \rightarrow^\star C',\, H'_1$. So by rule TIF we conclude that $\Gamma; K \vdash e'_s \,:\, \tau \triangleright H'_1 \cdot (H_2 + H_3)$ and by Lemma 4.5.13 that $H_1 \cdot (H_2 + H_3) \succcurlyeq \overline{H}'_1 \cdot H'_1 \cdot (H_2 + H_3)$.
  - rule IF2
    We know that $e'_s = e_2$, $\rho \vdash C,\, e_s \rightarrow^\star C,\, e'_s$ and $\Gamma; K \vdash e_2 \,:\, \tau \triangleright H_2$. The thesis is immediate because $H_1 \cdot (H_2 + H_3) = H_1 \cdot H_2 + H_1 \cdot H_3 \succcurlyeq H_1 \cdot H_2 \succcurlyeq \epsilon \cdot H_2$ and $C,\, \epsilon \cdot H_2 \rightarrow^\star C',\, H_2$.
  - rule IF3
    Similar to rule IF2

- rule TSUB $\quad \Gamma; K \vdash e_s \,:\, \tau' \triangleright H' \quad \tau' \leq \tau_c \quad H_s = H' + \tilde{H}$ so $H' \preccurlyeq H_s$
  Then by the induction hypothesis $\Gamma; K \vdash e'_s \,:\, \tau' \triangleright H'_1$, $H' \succcurlyeq \overline{H}'_1 \cdot H'_1$ and $C,\, \overline{H}'_1 \cdot H'_1 \rightarrow C',\, H'_1$. Since $H_s \succcurlyeq H' \succcurlyeq \overline{H}'_1 \cdot H'_1$ the thesis holds.

□

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

Proof. We first prove the statement for a single computation step. By using Lemma 4.4.1 we have that $\Gamma; K \vdash e' \,:\, \tau \triangleright H'$ with $H \succcurlyeq \overline{H}' \cdot H'$ and $C,\, \overline{H}' \cdot H' \rightarrow^\star C',\, H'$. Consider now H, the form of which can be either cases:

- $H = \overline{H}' \cdot H' + \overline{H}$
$$C,\, \overline{H}' \cdot H' + \overline{H} \rightarrow C,\, \overline{H}' \cdot H' \rightarrow^\star C',\, H'$$

- $H = \overline{H} \cdot \overline{H}' \cdot H'$ It is sufficient to take $\overline{H} = \epsilon$ because $C,\, \overline{H} \rightarrow^\star C, \epsilon$, so that
$$C,\, \overline{H} \cdot \overline{H}' \cdot H' \rightarrow^\star C,\, \overline{H}' \cdot H' \rightarrow^\star C',\, H'$$

- $H = \bigotimes_{i \in \{1,\ldots,n\}} ask\, G_i.H_i \otimes fail$ and $H_i = \overline{H}' \cdot H'$ for some i
$$C,\, \bigotimes_{i \in \{1,\ldots,n\}} ask\, G_i.H_i \otimes fail \rightarrow^\star C,\, H_i \rightarrow^\star C',\, H'$$

An easy inductive reasoning on the length of the computation then suffices to prove the statement. □

Lemma 4.4.2 (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 $\rho \vdash C, e_s \nrightarrow$ and H is viable for C, i.e. $C, H_s \nrightarrow^{+} C', fail$, 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 $e_s = \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$
  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 4.5.7(1), either rule if2 or if3 applies, contradiction.

- rule Tlet $e_s = \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$
  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 $e_s = \mathbf{tell}(e)$
  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 4.5.7(4) $v = F$, so the rule tell2 applies, contradiction.

- rule Tretract $e_s = \mathbf{retract}(e)$
  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 4.5.7(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 4.5.7(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 4.5.7(2) $e_1 = \mathbf{fun}\ f x \Rightarrow e$. 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 $e_s = \tilde{x}$
  If e_s is stuck we can have two cases only. The first case is that $\tilde{x} \notin dom(\rho)$. But this

is not possible because $DFV(e_s) \subseteq dom(\rho)$ by our hypothesis. The second case is that $\rho(\tilde{x}) = Va$ and $dsp(C, Va)$ is not defined. But this is not possible because C, $H_s \not\twoheadrightarrow^{\star} C', fail$ by our hypothesis, so the $dsp(C, Va)$ is defined and dvar rule applies, contradiction.

- rule Tvapp $\quad e_s = \#(e_1, e_2)$
  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 4.5.7(3) $e_1 = (x)\{Va\}$. If e_2 reduces, rule VaApp2 applies, contradiction. If e_2 is stuck we are in case (2). By induction hypothesis e_2 is a value, hence, rule VaApp3 applies, contradiction.

- rule Tdlet $\quad e_s =$ **dlet** $\tilde{x} = e_1$ **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 Definition 4.4.1 $\Gamma \vdash \rho' : K'$ with $\rho' = \rho[G.e, \rho(\tilde{x})/\tilde{x}]$ and $K' = K_{\tilde{x}}, (\tilde{x}, \tau, \Delta')$. 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 4.4.3 (Correctness)**.** *Let e_s be a closed expression such that $\Gamma; K \vdash e_s : \tau \triangleright H_s$; let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e_s, and that $\Gamma \vdash \rho : K$; and let C be a context such that H_s is viable i.e. $C, H_s \not\twoheadrightarrow^{+} C', fail$. Then either the computation terminates yielding a value ($\rho \vdash C, e_s \rightarrow^{\star} C'', v$) or it diverges, but it never gets stuck.*

Proof (By contradiction). Assume that $\rho \vdash C, e_s \rightarrow^{i} C'', e_s^i \not\twoheadrightarrow$ for some $i \in \mathbb{N}$ where e_s^i is a non-value stuck expression. By Corollary 1 we have $\Gamma; K \vdash e_s^i : \tau \triangleright H_s^i$ and $C, H_s \rightarrow^{\star} C'', H_s^i$, and since $C, H_s \not\twoheadrightarrow C'$, fail we have also $C, H_s^i \not\twoheadrightarrow C'$, fail. Then, Lemma 4.4.2 suffices to show that e_s^i is a value (contradiction). □

4.6 Loading-time Analysis

As said in Section 4.1, in the execution model of ML$_{\text{CoDa}}$ the compiler produces a triple (C_p, e_p, H_p) made of the application context, the object code and the effect over-approximating the behaviour of the application. Using it, the virtual machine of ML$_{\text{CoDa}}$ performs a linking and verification phases at loading time. During the linking phase, system variables are resolved and the initial context C is constructed, combining C_p and the system context. Regarding the verification phase, we check whether applications adapt to all evolutions of C that may occur at runtime, 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 (e.g. see Section 4.8), we build a graph $\mathcal{G}$ describing all 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. Below, we first describe the specification of our analysis and then an algorithm for it.

$$\frac{}{C,\ (\backepsilon \cdot H)^l \to C,\ H} \qquad \frac{}{C,\ \epsilon^l \to C,\ \backepsilon} \qquad \frac{}{C,\ \mathit{tell}\,F^l \to C \cup \{F\},\ \backepsilon}$$

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

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

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

Figure 4.8: New semantics of History Expressions

4.6.1 Analysis Specification

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

To support the formal development, we assume that history expressions are uniquely labelled on a given set of *Lab* as follows

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

In addition, we introduce for technical reasons a new empty history expression $\backepsilon$ 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 ($\backepsilon$). The semantics of history expressions is accordingly modified, and Figure 4.8 displays it (apart from labels and $\backepsilon$, this semantics and the old one clearly coincide). Furthermore, without loss of generality, we assume that all 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 $\Sigma_\circ, \Sigma_\bullet \colon \mathit{Lab} \to \wp(\mathit{Context} \cup \{\bullet\})$ where $\bullet$ is the distinguished "failure" context representing a dispatching failure. For each label l, the set $\Sigma_\circ(l)$ over-approximates the set of contexts that may arise before evaluating H^l (call it *pre-set*); instead $\Sigma_\bullet(l)$ over-approximates the set of contexts that may result from the evaluation of H^l (call it *post-set*).

We define the specification of our analysis through the validity relation

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

where $\mathcal{AE} = (\mathit{Lab} \to \wp(\mathit{Context} \cup \{\bullet\}))^2$ 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 4.6.2 to check whether the history expression H_p, hence the expression e it is an abstraction of, will never fail in a given initial context C.

In Figure 4.9 we give the inference rules that inductively define the validity relation $\models$. Now, we briefly comment on them, where $\mathcal{E}$ denotes the estimate $(\Sigma_\circ, \Sigma_\bullet)$.

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

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

Definition 4.6.1 (Valid analysis estimate). Given $H_p^{l_p}$ and an initial context C, we say that a pair $(\Sigma_\circ, \Sigma_\bullet)$ is a valid analysis estimate for H_p and C iff $C \in \Sigma_\circ(l_p)$ and $(\Sigma_\circ, \Sigma_\bullet) \models H_p^{l_p}$.

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 [NN02].

Theorem 4.6.1 (Existence of solutions). *Given H^l and an initial context C, the set $\{(\Sigma_\circ, \Sigma_\bullet) \mid (\Sigma_\circ, \Sigma_\bullet) \models H^l\}$ of the acceptable estimates of the analysis for H^l and C is a Moore family; hence, there exists a minimal valid estimate.*

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

$$\textsc{Anil}\;\frac{}{(\Sigma_\circ,\Sigma_\bullet) \models \ni}\qquad \textsc{Aeps}\;\frac{\Sigma_\circ(l) \subseteq \Sigma_\bullet(l)}{(\Sigma_\circ,\Sigma_\bullet) \models \epsilon^l}\qquad \textsc{Atell}\;\frac{\forall C \in \Sigma_\circ(l) \qquad C \cup \{F\} \in \Sigma_\bullet(l)}{(\Sigma_\circ,\Sigma_\bullet) \models \mathit{tell}\, F^l}$$

$$\textsc{Aretract}\;\frac{\forall C \in \Sigma_\circ(l) \qquad C \backslash \{F\} \in \Sigma_\bullet(l)}{(\Sigma_\circ,\Sigma_\bullet) \models \mathit{retract}\, F^l}$$

$$\textsc{Aseq1}\;\frac{\begin{array}{c}(\Sigma_\circ,\Sigma_\bullet) \models H_1^{l_1}\\ (\Sigma_\circ,\Sigma_\bullet) \models H_2^{l_2} \qquad \Sigma_\circ(l) \subseteq \Sigma_\bullet(l_1) \qquad \Sigma_\bullet(l_1) \subseteq \Sigma_\circ(l_2) \qquad \Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l)\end{array}}{(\Sigma_\circ,\Sigma_\bullet) \models (H_1^{l_1} \cdot H_2^{l_2})^l}$$

$$\textsc{Aseq2}\;\frac{(\Sigma_\circ,\Sigma_\bullet) \models H_2^{l_2} \qquad \Sigma_\circ(l) \subseteq \Sigma_\bullet(l_2) \qquad \Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l)}{(\Sigma_\circ,\Sigma_\bullet) \models (\ni \cdot H_2^{l_2})^l}$$

$$\textsc{Asum}\;\frac{\begin{array}{ccc}(\Sigma_\circ,\Sigma_\bullet) \models H_1^{l_1} & \Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) & \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)\\ (\Sigma_\circ,\Sigma_\bullet) \models H_2^{l_2} & \Sigma_\circ(l) \subseteq \Sigma_\circ(l_2) & \Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l)\end{array}}{(\Sigma_\circ,\Sigma_\bullet) \models (H_1^{l_1} + H_2^{l_2})^l}$$

$$\textsc{Aask1}\;\frac{\begin{array}{lllll}\forall C \in \Sigma_\circ(l) & (C \models G \implies (\Sigma_\circ,\Sigma_\bullet) \models H^{l_1} & \Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) & \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l))\\ & (C \not\models G \implies (\Sigma_\circ,\Sigma_\bullet) \models \Delta^{l_2} & \Sigma_\circ(l) \subseteq \Sigma_\circ(l_2) & \Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l))\end{array}}{(\Sigma_\circ,\Sigma_\bullet) \models (\mathit{ask}\, G.H^{l_1} \otimes \Delta^{l_2})^l}$$

$$\textsc{Aask2}\;\frac{\bullet \in \Sigma_\bullet(l)}{(\Sigma_\circ,\Sigma_\bullet) \models \mathit{fail}^l}\qquad \textsc{Arec}\;\frac{(\Sigma_\circ,\Sigma_\bullet) \models H^{l_1} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \qquad \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)}{(\Sigma_\circ,\Sigma_\bullet) \models (\mu h.H^{l_1})^l}$$

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

Figure 4.9: Specification of the analysis for History Expressions

	$\Sigma^1_\circ$	$\Sigma^1_\bullet$
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	$\emptyset$	$\{\bullet\}$
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	$\emptyset$	$\{\bullet\}$
11	$\emptyset$	$\emptyset$
12	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_2, F_5\}\}$
13	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_1, F_5\},\{F_2, F_5\}\}$

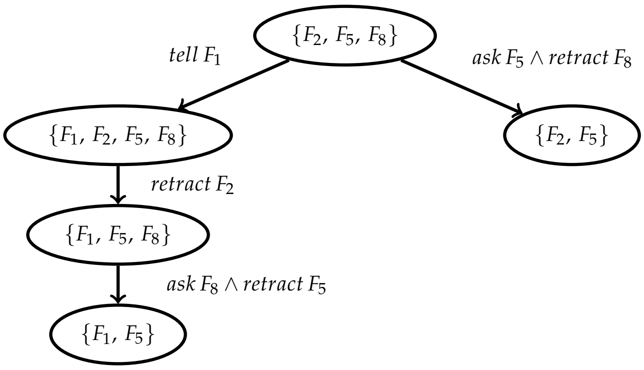


Figure 4.10: The analysis result (on top) and the evolution graph (on bottom) for the context $C = \{F_2, F_5, F_8\}$ and the history expression H_a.

Theorem 4.6.2 (Subject Reduction). *Let H^l be a closed history expression such that $(\Sigma_\circ, \Sigma_\bullet) \models H^l$. If for all $C \in \Sigma_\circ(l)$ it is $C, H^l \to C', H'^{l'}$ then $(\Sigma_\circ, \Sigma_\bullet) \models H'^{l'}$ 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 4.6.2 (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 $\forall l \in dom(\Sigma_\bullet) \setminus lfail(H_p)$ it is $\bullet \notin \Sigma_\bullet(l)$.

Example 4.6.1. We illustrate how viability is checked through a couple of examples. Consider the history expression

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

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

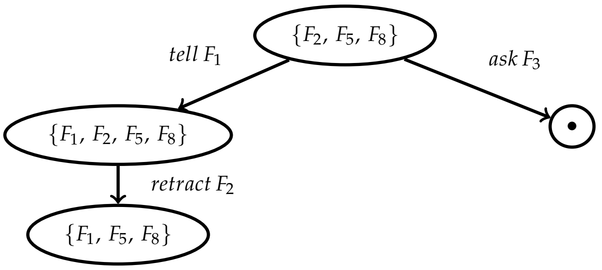


Figure 4.11: The analysis result (on top) and the evolution graph (on bottom) for the context $C = \{F_2, F_5, F_8\}$ and the history expression H'_a

$$(ask\, F_5.retract\, F_8^8 \otimes (ask\, F_3.retract\, F_4^9 \otimes fail^{10})^{11})^{12})^{13}$$

and the initial context $C = \{F_2, F_5, F_8\}$, consisting of facts only. For each label l occurring in H_a, Figure 4.10 shows the corresponding values of $\Sigma^1_\circ(l)$ and $\Sigma^1_\bullet(l)$, respectively. The column describing $\Sigma_\bullet$ contains $\bullet$ only for $l \in \{4, 7\}$ which are the labels of *fail*, so H_a is viable for C.

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

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

Indeed H'_a is not viable because the goal F_3 does not hold in C, and this is reflected by the occurrences of $\bullet$ in $\Sigma^2_\bullet(4)$ and $\Sigma^2_\bullet(7)$ as shown in Figure 4.11.

Now we exploit the result of the above analysis to build up the evolution graph $\mathcal{G}$ describing how the initial context C evolves at runtime. The abstract machine can use $\mathcal{G}$ to study how the application interacts with and affects the context. Reachability of specific contexts is easily checked on this graph. It can help verifying, besides viability, various other properties of the application behaviour, both functional and non-functional. For example, we could equip the language with security policies, and exploit the evolution graph to *statically* detect which context changes may lead to security violations (see **??**).

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

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

Example 4.6.2. As examples of evolution graphs consider the context C and the history expressions H_p and H'_p introduced above. The Figure 4.10 depicts the evolution graph of C for H_p. For the sake of clarity we put a label on the arc to specify which action changes the context. Since the node $\bullet$ is not reachable, H_p is viable for C. Instead, in the evolution graph of C for H'_p, displayed in Figure 4.11, the node $\bullet$ is reachable, showing H'_p not viable for C. Also in this case, we labelled for readability.

4.6.2 Analysis Algorithm

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

Given a history expression H_p 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 denoted by X.

To formalize the analysis as a constraint satisfaction problem, we first define the syntax of set-expressions and their semantics (Definition 4.6.4); the function $\mathcal{C}\,[\_] : \mathbb{H} \to \mathcal{SC}$ which generates constraints from a history expression (Definition 4.6.5); the constraints satisfaction relation $\models_{sc} \subseteq \mathcal{AE} \times \mathcal{SC}$ saying when a set of constraints is satisfied by an analysis estimate (Definition 4.6.6); finally, we prove that the valid estimates of the analysis (Definition 4.6.1) coincide with the solutions of the constraint system (Theorem 4.6.3).

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

$$X \in SetVar \qquad C \in Context^* \qquad G \in Goals^* \qquad F \in Facts^*$$
$$E ::= X \mid \{\bullet\} \mid \{C\} \mid E \sqcup F \mid E \setminus F \mid E \models G \mid E \not\models G \mid E_1 \Rightarrow E_2$$

where X is a variable; $\{\bullet\}$ is the singleton containing the bullet; the expression $E \sqcup F$ denotes the set of contexts of E where we have added the fact F to each element; the expression $E \setminus F$ denotes the set of contexts of E where we have removed the fact F from each element; the expression $E \models G$ denotes the subset of E containing only the contexts satisfying the goal G; the expression $E \not\models G$ denotes the subset of E containing only the contexts not satisfying the goal G; the expression $E_1 \Rightarrow E_2$ is a conditional expression: intuitively, the result is 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 defined on the syntax. 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 $\Sigma\colon Lab^* \to \wp(Context^* \cup \{\bullet\})$ 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 subscript variables with $\circ$ and $\bullet$ to indicate to which element of analysis estimate the variable refers.

The meaning of a set-expression is formalized as follow

Definition 4.6.4 (Set-expression semantics)**.** Given an analysis estimate $(\Sigma_\circ, \Sigma_\bullet)$ the semantics of set expressions is specified by the function $[\![\_]\!]\colon E \to \mathcal{AE} \to \wp(Context^* \cup \{\bullet\})$ defined as

$$
\begin{aligned}
&[\![\widehat{\Sigma}_\circ(l)]\!](\Sigma_\circ,\Sigma_\bullet) = \Sigma_\circ(l)\\
&[\![\widehat{\Sigma}_\bullet(l)]\!](\Sigma_\circ,\Sigma_\bullet) = \Sigma_\bullet(l)\\
&[\![\{\bullet\}]\!](\Sigma_\circ,\Sigma_\bullet) = \{\bullet\}\\
&[\![\{C\}]\!](\Sigma_\circ,\Sigma_\bullet) = \{C\}\\
&[\![E \sqcup F]\!](\Sigma_\circ,\Sigma_\bullet) = \{C \cup \{F\} \mid C \in [\![E]\!](\Sigma_\circ,\Sigma_\bullet)\}\\
&[\![E \setminus F]\!](\Sigma_\circ,\Sigma_\bullet) = \{C \setminus \{F\} \mid C \in [\![E]\!](\Sigma_\circ,\Sigma_\bullet)\}\\
&[\![E \models G]\!](\Sigma_\circ,\Sigma_\bullet) = \{C \in [\![E]\!](\Sigma_\circ,\Sigma_\bullet) \mid C \models G\}\\
&[\![E \not\models G]\!](\Sigma_\circ,\Sigma_\bullet) = \{C \in [\![E]\!](\Sigma_\circ,\Sigma_\bullet) \mid C \not\models G\}\\
&[\![E_1 \Rightarrow E_2]\!](\Sigma_\circ,\Sigma_\bullet) = \begin{cases} [\![E_2]\!](\Sigma_\circ,\Sigma_\bullet) & \text{if } [\![E_1]\!](\Sigma_\circ,\Sigma_\bullet) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}
$$

Given a history expression H, the function $\mathcal{C}\,[\_] : \mathbb{H} \to \mathcal{SC}$ generates the wanted set of constraints, mimicking the specification rules in Figure 4.9.

Definition 4.6.5 (Generation function)**.** Given a history expression $H_p^{l_p}$ and an initial context C, the set of constraints for $H_p^{l_p}$ and C is $\mathcal{S} = \{\{C\} \subseteq \widehat{\Sigma}_\circ(l_p)\} \cup \mathcal{C}\left[H_p^{l_p}\right]$, where the function $\mathcal{C}\,[\_] : \mathbb{H} \to \mathcal{SC}$ is inductively defined as follow

$$\mathcal{C}\,[\backepsilon] = \emptyset$$

$$\mathcal{C}\left[\epsilon^l\right] = \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\bullet(l)\}$$

$$
\begin{aligned}
&\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\\
&\qquad\qquad\qquad\qquad \{\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)\} \qquad \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),\\
&\qquad\qquad\qquad\qquad \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] = \{\bullet\} \subseteq \widehat{\Sigma}_\bullet(l)
\end{aligned}
$$

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

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 4.6.6 (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 two formulations of the analysis are equivalent

Theorem 4.6.3 (Equivalence)**.** *Let H be a history expression and let $(\Sigma_\circ, \Sigma_\bullet)$ be an analysis estimate, then*

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

The above theorem guarantees that the solutions of the set constraints $\mathcal{C}\left[H\right]$ are valid analysis estimates and vice versa.

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 4.12 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).

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 4.12: 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 $C = \{F_2, F_3, F_5\}$.

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 4.6.1).

The valid analysis estimate displayed at the bottom of Figure 4.12 is a solution to the constraints for H_p. Below we verify the satisfaction for the constraints of H_p by using Definition 4.6.4:

$$
\begin{aligned}
(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \widehat{\Sigma}_\circ(1) \sqcup F_1 \subseteq \widehat{\Sigma}_\bullet(1) \quad & \Leftrightarrow\ \llbracket \widehat{\Sigma}_\circ(1) \sqcup F_1 \rrbracket\ (\Sigma_\circ, \Sigma_\bullet) \subseteq \llbracket \widehat{\Sigma}_\bullet(1) \rrbracket\ (\Sigma_\circ, \Sigma_\bullet) \\
& \Leftrightarrow\ \{\{F_1, F_2, F_3, F_5\}\} \subseteq \{\{F_1, F_2, F_3, F_5\}\} \\
(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \widehat{\Sigma}_\circ(2) \setminus F_2 \subseteq \widehat{\Sigma}_\bullet(2) \quad & \Leftrightarrow\ \llbracket \widehat{\Sigma}_\circ(2) \setminus F_2 \rrbracket\ (\Sigma_\circ, \Sigma_\bullet) \subseteq \llbracket \widehat{\Sigma}_\bullet(2) \rrbracket\ (\Sigma_\circ, \Sigma_\bullet) \\
& \Leftrightarrow\ \{\{F_1, F_3, F_5\}\} \subseteq \{\{F_1, F_3, F_5\}\} \\
(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \widehat{\Sigma}_\circ(3) \subseteq \widehat{\Sigma}_\circ(1) \quad & \Leftrightarrow\ \llbracket \widehat{\Sigma}_\circ(3) \rrbracket\ (\Sigma_\circ, \Sigma_\bullet) \subseteq \llbracket \widehat{\Sigma}_\circ(1) \rrbracket\ (\Sigma_\circ, \Sigma_\bullet) \\
& \Leftrightarrow\ \{\{F_2, F_3, F_5\}\} \subseteq \{\{F_2, F_3, F_5\}\} \\
(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \widehat{\Sigma}_\bullet(1) \subseteq \widehat{\Sigma}_\circ(2) \quad & \Leftrightarrow\ \llbracket \widehat{\Sigma}_\bullet(1) \rrbracket\ (\Sigma_\circ, \Sigma_\bullet) \subseteq \llbracket \widehat{\Sigma}_\circ(2) \rrbracket\ (\Sigma_\circ, \Sigma_\bullet) \\
& \Leftrightarrow\ \{\{F_1, F_2, F_3, F_5\}\} \subseteq \{\{F_1, F_2, F_3, F_5\}\} \\
(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \widehat{\Sigma}_\bullet(2) \subseteq \widehat{\Sigma}_\bullet(3) \quad & \Leftrightarrow\ \llbracket \widehat{\Sigma}_\bullet(2) \rrbracket\ (\Sigma_\circ, \Sigma_\bullet) \subseteq \llbracket \widehat{\Sigma}_\bullet(3) \rrbracket\ (\Sigma_\circ, \Sigma_\bullet) \\
& \Leftrightarrow\ \{\{F_1, F_3, F_5\}\} \subseteq \{\{F_1, F_3, F_5\}\} \\
(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \{F_2, F_3, F_5\} \subseteq \widehat{\Sigma}_\circ(3) \quad & \Leftrightarrow\ \llbracket \{F_2, F_3, F_5\} \rrbracket\ (\Sigma_\circ, \Sigma_\bullet) \subseteq \llbracket \widehat{\Sigma}_\circ(3) \rrbracket\ (\Sigma_\circ, \Sigma_\bullet) \\
& \Leftrightarrow\ \{\{F_2, F_3, F_5\}\} \subseteq \{\{F_2, F_3, F_5\}\}
\end{aligned}
$$

To solve a system of constraints we define a worklist algorithm by instantiating the general schema in [NNH05] (Chapter 6). Given a set of constraints $\mathcal{S}$, it produces as solution an assignment $\mathcal{E}$ for the variables (represented as an array) occurring in the constraints. The algorithm is displayed in Figure 4.13. It 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 constraints its value influences.

In the first step we initialize 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 refinements of $\mathcal{E}$; 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 (`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 (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 $\mathcal{A}[\mathtt{X}]$. The algorithm terminates when $\mathcal{W}$ is empty, i.e. when all the constraints are satisfied.

Figure 4.14 shows the iterations of the algorithm to solve the constraints in Figure 4.12. After the initialization (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 initialized as shown at the bottom of Figure 4.14. 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. In particular, the following theorem holds:

Theorem 4.6.4 (Termination and Correctness of the Analysis Algorithm)**.** *Let H be a history expression of size n and let h be the height of the complete lattice $\wp(Context^* \cup \{\bullet\})$. The algorithm in Figure 4.13 always terminates and computes the minimal solution of the constraints $\mathcal{C}\,[H]$ in time $O(h \cdot n)$.*

The complexity of our algorithm depends on the value of height h of $\wp(Context^* \cup \{\bullet\})$. We conjecture that this value depends on the size n of the history expression to be analysed. If this value is large, the algorithm will perform many iterations before converging to the solution. Several approaches have been proposed to keep efficient the execution of the algorithm in practice, e.g. [Bou93, AS13]. These approaches consists of introducing a widening operator to speed up the fixed point calculation, but in this thesis we omit that optimisation.

4.7 Properties of the Analysis

In this we prove the correctness and some properties about our loading time analysis, in particular Theorems 4.6.1 and 4.6.2.

4.7.1 Analysis specification

First of all, 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 [DP02].

```
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 4.13: The worklist algorithm to solve constraints over set-expression

Definition 4.7.1 (Analysis Estimates Order)**.** Given two analysis estimates $(\Sigma_\circ^1, \Sigma_\bullet^1)$ and $(\Sigma_\circ^2, \Sigma_\bullet^2)$ we say $(\Sigma_\circ^1, \Sigma_\bullet^1) \sqsubseteq (\Sigma_\circ^2, \Sigma_\bullet^2)$ iff $\Sigma_\circ^1(l) \subseteq \Sigma_\circ^2(l)$ and $\Sigma_\bullet^1(l) \subseteq \Sigma_\bullet^2(l)$ for all $l \in$ *Lab*, where $\subseteq$ is the order of $\wp(Context)$. Furthermore, we define $(\Sigma_\circ^1, \Sigma_\bullet^1) \sqcap (\Sigma_\circ^2, \Sigma_\bullet^2) = (\Sigma_\circ^1 \sqcap \Sigma_\circ^2, \Sigma_\bullet^1 \sqcap \Sigma_\bullet^2) = (\lambda l.\Sigma_\circ^1(l) \cap \Sigma_\circ^2(l), \lambda l.\Sigma_\bullet^1(l) \cap \Sigma_\bullet^2(l))$.

By exploiting standard lattice theory results it is straightforward prove that analysis estimates are a complete lattice [DP02]. In the following we denote the top of this lattice with $(\Sigma^\top, \Sigma^\top)$.

Lemma 4.7.1.

1. *Let be $(\Sigma^\top, \Sigma^\top)$ the top of lattice of the analysis estimates, then $(\Sigma^\top, \Sigma^\top) \models H^l$ for all H^l*

2. *Let be $(\Sigma_\circ^1, \Sigma_\bullet^1)$ and $(\Sigma_\circ^2, \Sigma_\bullet^2)$, if $(\Sigma_\circ^1, \Sigma_\bullet^1) \models H^l$ and $(\Sigma_\circ^2, \Sigma_\bullet^2) \models H^l$ then $(\Sigma_\circ^1 \sqcap \Sigma_\circ^2, \Sigma_\bullet^1 \sqcap \Sigma_\bullet^2) \models H^l$*

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	$\{\mathbf{C}\} \subseteq \boldsymbol{\widehat{\Sigma}_\circ(3)}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\cancel{\{C\} \subseteq \widehat{\Sigma}_\circ(3)}\ \boldsymbol{\widehat{\Sigma}_\circ(3) \subseteq \widehat{\Sigma}_\circ(1)}$	$\emptyset$	$\emptyset$	$\mathbf{C}$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\cancel{\widehat{\Sigma}_\circ(3) \subseteq \widehat{\Sigma}_\circ(1)}\ \boldsymbol{\widehat{\Sigma}_\circ(1) \sqcup F_1 \subseteq \widehat{\Sigma}_\bullet(1)}$	$\mathbf{C}$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$
3	$\cancel{\widehat{\Sigma}_\circ(1) \sqcup F_1 \subseteq \widehat{\Sigma}_\bullet(1)}\ \boldsymbol{\widehat{\Sigma}_\bullet(1) \subseteq \widehat{\Sigma}_\circ(2)}$	C	$\emptyset$	C	$\mathbf{C_1}$	$\emptyset$	$\emptyset$
4	$\cancel{\widehat{\Sigma}_\bullet(1) \subseteq \widehat{\Sigma}_\circ(2)}\ \boldsymbol{\widehat{\Sigma}_\circ(2) \setminus F_2 \subseteq \widehat{\Sigma}_\bullet(2)}$	C	$\mathbf{C_1}$	C	C_1	$\emptyset$	$\emptyset$
5	$\cancel{\widehat{\Sigma}_\circ(2) \setminus F_2 \subseteq \widehat{\Sigma}_\bullet(2)}\ \boldsymbol{\widehat{\Sigma}_\bullet(2) \subseteq \widehat{\Sigma}_\bullet(3)}$	C	C_1	C	C_1	$\mathbf{C_2}$	$\emptyset$
6	$\cancel{\widehat{\Sigma}_\bullet(2) \subseteq \widehat{\Sigma}_\bullet(3)}\ \emptyset$	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 4.14: The iterations of the worklist algorithm to solve constraints in Figure 4.12 (top) and the content of the corresponding array $\mathcal{A}$ (bottom).

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 $H^l = (H_1^{l_1} \cdot H_2^{l_2})^l$.

1. By induction hypothesis we know that $(\Sigma^\top, \Sigma^\top) \vDash H_i^{l_i}$ 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 $(\Sigma_\circ^1, \Sigma_\bullet^1) \vDash H^l$ and the premise of rule Aseq1, we know that $\Sigma_\circ^1(l) \subseteq \Sigma_\circ^1(l_1)$, $\Sigma_\bullet^1(l_1) \subseteq \Sigma_\circ^1(l_2)$, $\Sigma_\bullet^1(l_2) \subseteq \Sigma_\bullet^1(l)$ and that $\Sigma_\circ^2(l) \subseteq \Sigma_\circ^2(l_1)$, $\Sigma_\bullet^2(l_1) \subseteq \Sigma_\circ^2(l_2)$, $\Sigma_\bullet^2(l_2) \subseteq \Sigma_\bullet^2(l)$. Since $\sqcap$ is monotonic with respect $\subseteq$ it holds $\Sigma_\circ^1(l) \sqcap \Sigma_\circ^2(l) \subseteq \Sigma_\circ^1(l_1) \sqcap \Sigma_\circ^2(l_1)$, $\Sigma_\bullet^1(l_1) \sqcap \Sigma_\bullet^2(l_1) \subseteq \Sigma_\circ^1(l_2) \sqcap \Sigma_\circ^2(l_2)$, $\Sigma_\bullet^1(l_2) \sqcap \Sigma_\bullet^2(l_2) \subseteq \Sigma_\bullet^1(l) \sqcap \subseteq \Sigma_\bullet^2(l)$. Then by the induction hypothesis and by the above inclusions we satisfy the premise of rule (Aseq1) and we conclude $(\Sigma_\circ^1 \sqcap \Sigma_\circ^2, \Sigma_\bullet^1 \sqcap \Sigma_\bullet^2) \vDash H^l$.

□

By exploiting the above two lemmata we can prove:

Theorem 4.6.1 (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 $Y = \{(\Sigma_\circ^i, \Sigma_\bullet^i) \mid i \in \{1, \ldots, n\}\} \subseteq \{(\Sigma_\circ, \Sigma_\bullet) \mid (\Sigma_\circ, \Sigma_\bullet) \vDash H^l\}$, $\sqcap Y \in \{(\Sigma_\circ, \Sigma_\bullet) \mid (\Sigma_\circ, \Sigma_\bullet) \vDash H^l\}$. By applying $n+1$ times Lemma 4.7.1 we have that $(\Sigma^\top, \Sigma^\top) \sqcap (\Sigma_\circ^1, \Sigma_\bullet^1) \sqcap \cdots \sqcap (\Sigma_\circ^n, \Sigma_\bullet^n) \vDash H^l$ holds. □

The following definition and lemmata helps us to prove the subject reduction result.

Definition 4.7.2 (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 4.7.2 (Pre-substitution)**.** *Let H^l, $H_1^{l_1}$ and $H_2^{l_2}$ be history expressions such that $H_1^{l_1}$ is an immediate subterm of H^l; let $(\Sigma_\circ, \Sigma_\bullet) \models H^l$, $(\Sigma_\circ, \Sigma_\bullet) \models H_1^{l_1}$ and $(\Sigma_\circ, \Sigma_\bullet) \models H_2^{l_2}$, 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 $(\Sigma_\circ, \Sigma_\bullet) \models H^l[H_2^{l_2}/H_1^{l_1}]$.

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

- case *э*, ϵ^l, *tell* F^l, $fail^l$, *retract* F^l, h^l
  straightforward

- case $H^l = (H_1^{l_1} + H_3^{l_3})^l$
  From the hypothesis and from the 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 $(\Sigma_\circ, \Sigma_\bullet) \models (H_2^{l_2} + H_3^{l_3})^l$

- case $H^l = (H_3^{l_3} + H_1^{l_1})^l$
  Similar to the previous case.

- case $H^l = (H_1^{l_1} \cdot H_3^{l_3})^l$
  From the hypothesis and by the premise of rule Aseq1 we have $\Sigma_\circ(l) \subseteq \Sigma_\bullet(l_1) \subseteq \Sigma_\circ(l_2)$ 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 $(\Sigma_\circ, \Sigma_\bullet) \models (H_2^{l_2} \cdot H_3^{l_3})^l$.

- case $H^l = (H_3^{l_3} \cdot H_1^{l_1})^l$
  From the hypothesis and 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 $(\Sigma_\circ, \Sigma_\bullet) \models (H_3^{l_3} \cdot H_1^{l_1})^l$.

- case $H^l = (\mu h.H_1^{l_1})^l$
  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 $(\Sigma_\circ, \Sigma_\bullet) \models (\mu h.H_2^{l_2})^l$

- case $H^l = (askG.H_1^{l_1} \otimes \Delta^{l_3})^l$
  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 $(\Sigma_\circ, \Sigma_\bullet) \models (askG.H_2^{l_2} \otimes \Delta^{l_3})^l$

□

Lemma 4.7.3 (Substitution)**.** *Le H^l, $H_1^{l_1}$ and $H_2^{l_2}$ be history expressions such that $H_1^{l_1}$ is a subterm of H^l; let $(\Sigma_\circ, \Sigma_\bullet) \models H^l$, $(\Sigma_\circ, \Sigma_\bullet) \models H_1^{l_1}$ and $(\Sigma_\circ, \Sigma_\bullet) \models H_2^{l_2}$, 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 $(\Sigma_\circ, \Sigma_\bullet) \models H^l[H_2^{l_2}/H_1^{l_1}]$.

Proof. Since $H_1^{l_1}$ is subterm of H, there exists then another subterm of H, say $H_3^{l_3}$, such that $H_1^{l_1}$ is an immediate subterm of $H_3^{l_3}$. Since $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l$, there exists then a subderivation with conclusion $(\Sigma_\circ, \Sigma_\bullet) \vDash H_3^{l_3}$. Since $H_1^{l_1}$ is an immediate subterm of $H_3^{l_3}$ there exists another subderivation with conclusion $(\Sigma_\circ, \Sigma_\bullet) \vDash H_1^{l_1}$. So by applying Lemma 4.7.2, 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 4.6.2 (Subject Reduction)**.** *Let H^l be a closed history expression such that $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l$. If for all $C \in \Sigma_\circ(l)$ it is $C, H^l \to C', H'^{l'}$ then $(\Sigma_\circ, \Sigma_\bullet) \vDash H'^{l'}$ and $\Sigma_\circ(l) \subseteq \Sigma_\circ(l')$ and $\Sigma_\bullet(l') \subseteq \Sigma_\bullet(l)$.*

Proof. 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 $C, \epsilon^l \to C, \backepsilon$, then the statement vacuously holds.

- rule ATELL
  We know that in this case $C, tell\, F^l \to C \cup \{F\}, \backepsilon$, 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

  $$\begin{aligned} \Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_3) &\implies \Sigma_\circ(l) \subseteq \Sigma_\circ(l_3) \\ \Sigma_\bullet(l_3) \subseteq \Sigma_\bullet(l_1) \subseteq \Sigma_\circ(l_2) &\implies \Sigma_\bullet(l_3) \subseteq \Sigma_\circ(l_2) \end{aligned}$$

  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 $H^l = (\backepsilon \cdot H_2^{l_2})^l$ 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 $(\Sigma_\circ, \Sigma_\bullet) \models H_1'^{l_1'}$, $\Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_1')$ 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 $C \models G$ and by the premise of Aask1 rule we trivially $(\Sigma_\circ, \Sigma_\bullet) \models H_1^{l_1}$, $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$ and $\Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$.
  2. case $H'^{l'} = \Delta^{l_2}$. Similar to case (1) but we know $C \not\models G$.

- 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 $(\Sigma_\circ, \Sigma_\bullet) \models H_1^{l_1}$, $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$ 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 $(\Sigma_\circ, \Sigma_\bullet) \models H_1^{l_1}$ and since our analysis rules are defined on the syntax of history expressions, there exists a subderivation of $(\Sigma_\circ, \Sigma_\bullet) \models H_1^{l_1}$ proof with conclusion $(\Sigma_\circ, \Sigma_\bullet) \models h^{l^i}$. 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 4.7.3 n-times, we have $(\Sigma_\circ, \Sigma_\bullet) \models H_1^{l_1}[(\mu.H_1^{l_1})^l/h^{l^i}]$ 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.

□

4.7.2 Analysis Algorithm

Here, we present some properties about our constraints and the correctness of our algorithm. The following lemma establishes the height of our property domain $\wp(Context^* \cup \{\bullet\})$. Recall that $Context^*$ is the set of all contexts may be generated from the initial context by a given history expression.

Lemma 4.7.4 (Height of Context Lattice). *The height of the complete lattice $\wp(Context^* \cup \{\bullet\})$ is $\#Context^* + 1$.*

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

Definition 4.7.3 (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

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

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 4.7.5 (Constraints Size)**.** *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 4.6.5 it is easy to see that the left-hand-size of all generated constraints can be a variable X, a a term $X \boxtimes F$ for some fact F and $\boxtimes \in \{\sqcup, \setminus\}$, a term $X \Box G$ and a term $X_1 \Box G \Rightarrow X_2$ with $\Box \in \{\models, \not\models\}$ for some G and $X_1 \neq X_2$. The thesis follows by considering the case $X_1 \Box G \Rightarrow X_2$. □

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 4.7.6 (Number of Constraints)**.** *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 $H = \ni$, $H = \epsilon^l$, $H = h^l$, $H = tell\,F^l$, $H = retract\,F^l$, $H = fail^l$
  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 4.6.5 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 4.6.5 we have
  $$\#\mathcal{C}\,[H] \;=\; \#\mathcal{C}\,[H_1] + 2 \;\leq 4n_1 + 2 \;\leq\; 4(n_1 + 1) \;=\; 4n$$

□

Theorem 4.6.3 (Equivalence). *Let H be a history expression and let $(\Sigma_\circ, \Sigma_\bullet)$ be an analysis estimate, then*
$$(\Sigma_\circ, \Sigma_\bullet) \vDash H \iff (\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \mathcal{C}\,[H]$$

Proof. The proof is in two parts. In the first part, we prove
$$(\Sigma_\circ, \Sigma_\bullet) \vDash H \implies (\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \mathcal{C}\,[H]$$

- case $H = \backepsilon$
  Trivial since $\mathcal{C}\,[\backepsilon] = \emptyset$ and $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \emptyset$.

- case $H = \epsilon^l$
  We need to prove $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\bullet(l)\}$, i.e. by Definition 4.6.6 $[\![\widehat{\Sigma}_\circ(l)]\!](\Sigma_\circ, \Sigma_\bullet) \subseteq [\![\widehat{\Sigma}_\bullet(l)]\!](\Sigma_\circ, \Sigma_\bullet)$. The thesis follows from the premise of the rule Aeps.

- case $H = tell\,F^l$
  We have that $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \{\widehat{\Sigma}_\circ(l) \sqcup F \subseteq \widehat{\Sigma}_\bullet(l)\}$ is equivalent to $[\![\widehat{\Sigma}_\circ(l) \sqcup F]\!](\Sigma_\circ, \Sigma_\bullet) \subseteq [\![\widehat{\Sigma}_\bullet(l)]\!](\Sigma_\circ, \Sigma_\bullet)$ by Definition 4.6.6, $\{C \cup \{F\} \mid C \in \Sigma_\circ(l)\} \subseteq \Sigma_\bullet(l)$. The thesis follows from applying the premise of the rule Atell.

- case $H = retract\,F^l$
  Similar to the case $H = tell\,F$, replace $\widehat{\Sigma}_\circ(l) \setminus F$ with $\widehat{\Sigma}_\circ(l) \sqcup F$ and $C \cup \{F\}$ with $C \setminus \{F\}$.

- case $H = (H_1^{l_1} \cdot H_2^{l_2})^l$
  We need to prove $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \mathcal{C}\,[H_1] \cup \mathcal{C}\,[H_2] \cup \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1),\, \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$. By the premise of rule Aseq1 we know that it holds $(\Sigma_\circ, \Sigma_\bullet) \vDash H_i$ for $i = 1, 2$; so by applying the induction hypothesis we have $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \mathcal{C}\,[H_i]$ for $i = 1, 2$. It remains to prove $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\bullet(l_1),\, \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$, i.e. by Definition 4.6.6 and Definition 4.6.4 $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$, $\Sigma_\bullet(l_1) \subseteq \Sigma_\circ(l_2)$ 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 $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \mathcal{C}\left[H_1^{l_1}\right] \;\cup\; \mathcal{C}\left[H_2^{l_2}\right] \cup \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l), \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$. By the premise of the rule Asum we know $(\Sigma_\circ, \Sigma_\bullet) \vDash H_i$ for $i = 1, 2$; by applying the induction hypothesis $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \mathcal{C}\left[H_i^{l_i}\right]$ holds for $i = 1, 2$. It remains to prove $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l), \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$. By applying Definition 4.6.6 and Definition 4.6.4, 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 Definition 4.6.6 and Definition 4.6.4 $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1), \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$

- case $H = \mathit{fail}^l$
  By Definition 4.6.6 and Definition 4.6.4 $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \{\bullet\} \subseteq \widehat{\Sigma}_\bullet(l)$ holds iff $\{\bullet\} \subseteq \Sigma_\bullet(l)$. 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 Definition 4.6.6 and Definition 4.6.4 $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1), \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$. These inequalities hold by the premise of the rule Arec.

- case $H = (\mathit{ask}\, G.H_1^{l_1} \otimes \Delta^{l_2})^l$
  We need to prove $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \mathcal{C}\,[H] \cup \mathcal{C}\,[\Delta] \cup \{\widehat{\Sigma}_\circ(l) \models G \Rightarrow \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\circ(l) \models G \Rightarrow \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l), \widehat{\Sigma}_\circ(l) \not\models G \Rightarrow \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\circ(l) \not\models G \Rightarrow \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$. By the premise of the rule Aask1 we know $(\Sigma_\circ, \Sigma_\bullet) \models H$ and $(\Sigma_\circ, \Sigma_\bullet) \models \Delta$ hold, so by applying the induction hypothesis we have $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \mathcal{C}\,[H]$ and $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \mathcal{C}\,[\Delta]$. It remains to prove $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \{\widehat{\Sigma}_\circ(l) \models G \Rightarrow \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\circ(l) \models G \Rightarrow \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l), \widehat{\Sigma}_\circ(l) \not\models G \Rightarrow \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\circ(l) \not\models G \Rightarrow \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$, i.e. by Definition 4.6.6 and Definition 4.6.4 $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$, $\Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$ 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

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

- case $H = \backepsilon$
  trivial since the rule Anil is an axiom.

- case $H = \epsilon^l$
  By our assumption and by Definition 4.6.6 and Definition 4.6.4 we know $\Sigma_\circ(l) \subseteq \Sigma_\bullet(l)$, so by applying the rule Aeps we have $(\Sigma_\circ, \Sigma_\bullet) \models \epsilon^l$.

- case $H = \mathit{tell}\, F^l$
  By our assumption and by Definition 4.6.6 and Definition 4.6.4 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 $(\Sigma_\circ, \Sigma_\bullet) \models \mathit{tell}\, F^l$.

- case $H = \mathit{retract}\, F^l$
  Similar to the case of $H = \mathit{tell}\, F^l$

- case $H = \mathit{fail}^l$
  By our assumption and by Definition 4.6.6 and Definition 4.6.4 we know $\{\bullet\} \subseteq \Sigma_\bullet(l)$, so by applying the rule Aask2 we have $(\Sigma_\circ, \Sigma_\bullet) \models \mathit{fail}^l$.

- case $H = h^l$
  By our assumption and by Definition 4.6.6 and Definition 4.6.4 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 $(\Sigma_\circ, \Sigma_\bullet) \models h^l$.

- case $H = (H_1^{l_1} \cdot H_2^{l_2})^l$
  By our assumption we know $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \mathcal{C}\,[H_i]$ for $i = 1, 2$ and $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\bullet(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$. By induction hypothesis we have $(\Sigma_\circ, \Sigma_\bullet) \models H_i$ for $i = 1, 2$ hold. Also by Definition 4.6.4 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 $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \mathcal{C}\,[H_i]$ for $i = 1, 2$ and $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\bullet(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l), \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\bullet(l_2), \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$. By induction hypothesis we have $(\Sigma_\circ, \Sigma_\bullet) \models H_i$ for $i = 1, 2$ hold. Also by Definition 4.6.4 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 $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \mathcal{C}\,[H_1]$ and $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l)\}$ hold by our assumption. By induction hypothesis we have $(\Sigma_\circ, \Sigma_\bullet) \models H_1$ and by Definition 4.6.4 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 $H = (\mathit{ask}\, G.H_1^{l_1} \otimes \Delta^{l_2})^l$
  By our assumption we know $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \mathcal{C}\,[H_1]$, $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \mathcal{C}\,[\Delta]$, $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \{\widehat{\Sigma}_\circ(l) \models G \Rightarrow \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\circ(l) \models G \Rightarrow \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l), \widehat{\Sigma}_\circ(l) \not\models G \Rightarrow \widehat{\Sigma}_\bullet(l) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\circ(l) \not\models G \Rightarrow \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$. By applying induction hypothesis we have $(\Sigma_\circ, \Sigma_\bullet) \models H_1$ and $(\Sigma_\circ, \Sigma_\bullet) \models \Delta$. Also by Definition 4.6.4 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 4.6.4 (Termination and Correctness of the Analysis Algorithm). *Let H be a history expression of size n and let h be the height of the complete lattice $\wp(Context^* \cup \{\bullet\})$. The algorithm in Figure 4.13 always 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 [NNH05], the termination and the correctness follows from Lemma 6.4 of [NNH05].

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 4.7.5 and $N = O(n)$ by Lemma 4.7.6, thus the complexity is $O(h \cdot n)$. Moreover, by Lemma 4.7.4 we have that $h = O(\#Context^*)$, hence the overall complexity becomes $O(\#Context^* \cdot n)$. As we said in Section 4.6 we conjecture that $\#Context^*$ is a function $f(n)$ of history expression size and we plan either to compute $f(n)$ or give a good upper bound. □

4.8 Security in $\mathrm{ML}_{\mathrm{CoDa}}$

In this section we extend $\mathrm{ML}_{\mathrm{CoDa}}$ for addressing security issues, in particular for enforcing security policies over the context. Our idea consists of expressing policies in Datalog as we did for the context. The version of Datalog used in $\mathrm{ML}_{\mathrm{CoDa}}$, i.e. Stratified Datalog with Negation, is sufficiently expressive for security policies. Indeed, many logical languages for defining control access policies compile in it, e.g. [LM03, DeT02, BDCDVS02]. Furthermore, in this way $\mathrm{ML}_{\mathrm{CoDa}}$ requires few extensions to deal with security because we can reuse our logical machinery.

We can distinguish two classes of policies: those specified by the system to control the user's behaviour, and those expressed by the application. We are only interest in system policies. This is because the application developer has full knowledge of his policies, and so he can directly specify them through behavioural variation constructs. Instead, the application has no *a priori* knowledge about system policies; there is no warranty then that the application was designed according to them. Hence, here we assume a scenario in which the application is not malicious but it can violate a system policy Φ because at development time this policy is *unknown*.

In this new setting, an application can also fail due to a policy violation (*non-functional failure*). One would like to predict as earlier as possible if this case may occur, but because of the open context notion of $\mathrm{ML}_{\mathrm{CoDa}}$ a fully static approach is not feasible.

Our solution consists of introducing a sort of runtime monitor that is switched on and off at need. As we will see, the dispatching mechanism of $\mathrm{ML}_{\mathrm{CoDa}}$ suffices for natively supporting it. However, we extend our static analysis to detect in which parts of the code there may be policy violations, in order to provide the monitor with the information needed to check the possible risky activities. In particular, we modify the type and effect system so that it also computes a *labelling environment*, i.e. a function storing the correspondence between the program code and the history expression. We use this labelling environment at loading time to annotate the edge of the evolution graph $\mathcal{G}$ with references (*labels*) to the **tell** and **retract** operations in the code. Before launching the execution, now we also detect the unsafe operations by checking the system policy Φ on each node of $\mathcal{G}$. Our runtime monitor can guard them, while it will be switched off on the remaining (*safe*) actions. Actually, we collect the labels of the risky operations and associate the value `on` with them, and `off` with all others.

To make the above effective, we introduce a compilation schema where the compiler substitutes a behavioural variation *bv* for each occurrence of a **tell**/**retract**. A runtime,

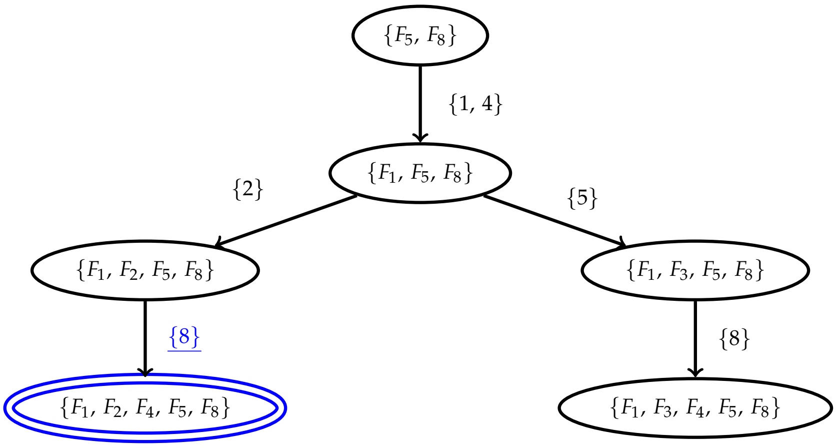


Figure 4.15: The evolution graph for the context $\{F_5, F_8\}$ and for the history expression $H_a = (((\mathit{tell}\ F_1^1 \cdot \mathit{tell}\ F_2^2)^3 + (\mathit{tell}\ F_1^4 \cdot \mathit{tell}\ F_3^5)^6)^7 \cdot \underline{\mathit{tell}\ F_4^8)^9}$

bv checks if the policy Φ holds in the current context, but only when the label of is `on`.

In the following we first explain our approach through a running example, then we extend the semantics of ML_{CoDa} and its static analysis.

4.8.1 Security in the museum

Consider the museum scenario described in Section 4.2; assume that the context also stores information about the room in which the user is through the predicate `current_room`. If the user moves from the room *delicate paintings* to the one *sculptures*, the application updates the context by executing

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

Assume that one can take pictures in every room, but that in the delicate paintings room it is forbidden to use the camera flash, not to damage the paintings. This policy is specified by the museum (the system) and it must be enforced during the user's tour. As said policies predicate on the context, so they are easily expressed as Datalog goals. If the fact `flash_on` holds when the flash is active and if `button_clicked` holds when the user presses the button of the camera. The above policy `phi`, intuitively, corresponds to the logical condition $\mathit{current\_room}(\mathit{delicate\_paintings}) \Rightarrow (\mathit{button\_clicked} \Rightarrow \neg \mathit{flash\_on})$ and it can be expressed in Datalog by the clauses

```
phi ← ¬ current_room(delicate_paintings).
phi ← ¬ flash_on.
phi ← ¬ button_pressed.
```

Of course, the museum can specify other policies, but for simplicity we assume that there is a unique global policy Φ, obtained by suitably combining all the required policies. The enforcement is obtained by querying the validity of Φ in the current context. The risky operations are those which modify the context possibly leading to a violation. Hence, we need to carry checks out only before every **tell**/**retract**. To better understand this point, take the following expression e_a:

```
let x =
  if always_flash then
    let y = tell photocamera_started^1 in
      tell flash_on^2
  else
    let y = tell mode_museum_activated^3 in
      tell photocamera_started^4
  in
    tell button_pressed^5
```

For clarity, here (and in the syntax later), we show the labels of **tell**/**retract** in the code, inserted by the compiler while parsing. Intuitively, the expression simulates the actions done in order to take a picture: if the user has specified he wants the flash to be always used, then the camera is activated and the context is informed that the flash is on; otherwise, the camera is activated and the context is informed that we have chosen a modality which is suitable for museums. Finally, in both cases the context is informed we are about to take the picture.

Consider a context satisfying the policy `phi` and where the fact `current_room(` `delicate_paintings)` holds. If we evaluate the above code in this context, it is easy to observe that carrying out **tell** `button_pressed` may cause a violation, depending on the actual value of `always_flash`. Thus, we have to activate the runtime monitor to check it. However, other **tell**s lead to no violation, hence, we do not need to check them.

To gather the needed information for the runtime monitor at compile time computes a type, an annotated history expression and a labelling environment. The type of e_a is `unit`, as well as that of **tell** $\texttt{F}_4$, and its history expression is

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

The labels of history expressions allow us to link the actions in histories to the corresponding actions of the code, e.g. the first $\texttt{tell}\,\texttt{F}_1^1$ in H_a, corresponds to the first **tell** in e_a, that is also labelled by 1, while the $\texttt{tell}\,\texttt{F}_4^8$ in H_a, is linked to the action with label 5 in e_a. All the correspondences are stored in the labelling environment $\Lambda_a = \{1 \mapsto 1, 2 \mapsto 2, 4 \mapsto 3, 5 \mapsto 4, 8 \mapsto 5\}$ (the abstract labels that do not annotate **tell**/**retract** actions have no counterpart).

Consider now an initial context C that includes the facts F_5 (irrelevant here), and $F_8 \equiv$ `current_room(delicate_paintings)`, but no facts in $\{F_1, F_2, F_3, F_4\}$. Starting from C (and from H_a) our *loading time* analysis builds the graph described in Figure 4.15. Nodes represent contexts, possibly reachable at runtime, while edges represent transitions from one context to another. But now each edge is annotated with the set of actions in H_a that may cause that transition, e.g. from C it is possible to reach the context also including the fact F_1, because of the two *tell* operations labelled by 1 and by 4 in H_a. Therefore, an edge can have more than one label (e.g. the one labelled $\{1, 4\}$). Note also that the same label may occur in more than one edge (e.g. the label 8).

By visiting the graph, we observe that the context $\{F_1, F_2, F_4, F_5, F_8\}$ (the blue and double circled node in Figure 4.15) violates our no-flash policy. At runtime the action

$$\textsc{Tell2}\ \frac{dsp(C \cup \{F\}, \Phi.()) = ((), \emptyset)}{\rho \vdash C,\ \mathbf{tell}(F) \to C \cup \{F\},\ ()} \qquad \textsc{Retract2}\ \frac{dsp(C \backslash \{F\}, \Phi.()) = ((), \emptyset)}{\rho \vdash C,\ \mathbf{retract}(F) \to C \backslash \{F\},\ ()}$$

Figure 4.16: The modified rules for updating the context

labelled with 8 (underlined and in blue), corresponding to **tell** $\mathtt{F}_4$ must be blocked. For preventing this violation, all we have to do is activating the runtime monitor, right before executing this risky operation.

4.8.2 Extending ML$_{\mathbf{CoDa}}$ semantics

The syntax of ML$_{\text{CoDa}}$ is almost unchanged apart from the fact that we associate each **tell**/**retract** with a label $l \in Lab_C$.

As we said, we implement our runtime monitor exploiting the dispatching mechanism. The idea is that given the system policy Φ we build the special variation $\Phi.()$, through which we invoke the dsp function: if Φ does not hold in the current context C the dispatching mechanism fails meaning that we have a violation. To do that we need only modify the rules Tell2 and Retract2 in the semantics.

The new versions of the rules are in Figure 4.16. The new context C', obtained from C by adding/removing F, is checked against Φ in the premise of the rules. If this call produces a result, then the evaluation yields the unit value and the new context C' as done in Section 4.3.

The following example shows the reduction of a **retract** construct violating the policy $\Phi \leftarrow F_4$. Let C be $\{F_3, F_4, F_5\}$ and apply $\mathtt{f} = \mathbf{fun}\ f\mathtt{x} \Rightarrow \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{F}_5\ \mathtt{else}\ \mathtt{F}_4$ to unit, assuming that the evaluation of e_1 reduces to `false` without changing the context:

$$\rho \vdash C,\ \mathtt{retract(f\ ())}^l \to^+ C,\ \mathtt{retract(F_4)}^l \not\to$$

Since Φ requires the fact F_4 to always hold, every attempt to remove it from the context violates Φ. Consequently, the evaluation gets stuck because $dsp(C\backslash\{F_4\}, \Phi.())$ fails. If e_1 reduces to `true`, there is no policy violation and the evaluation reduces to unit:

$$\rho \vdash C,\ \mathtt{retract(f\ ())}^l \to^+ C,\ \mathtt{retract(F_5)}^l \to C\backslash\{\mathtt{F_5}\}, ()$$

4.8.3 Extending the static analysis

Type and Effect system We modify the type and effect system in such a way it computes labelled History Expressions, and also a function called labelling environment. This function stores the correspondence between the labels of History Expressions and labels of the code.

The syntax and the semantics of History Expression are unchanged but we assume them labelled as done in Section 4.6 on a given set of Lab_H. Consider as given the function $h\colon Lab_H \to \mathbb{H}$ that recovers a construct in a given history expression $h \in \mathbb{H}$ from a label l.

$$\textsc{Srefl}\ \frac{}{\tau \le \tau} \qquad \textsc{Sfact}\ \frac{\phi \subseteq \phi'}{fact_{\phi} \le fact_{\phi'}}$$

$$\textsc{Sfun}\ \frac{\begin{array}{cc} \tau_1' \le \tau_1 & \tau_2 \le \tau_2' \\ K \sqsubseteq K' & H \sqsubseteq H' \end{array}}{\tau_1 \xrightarrow{K|H} \tau_2 \le \tau_1' \xrightarrow{K'|H'} \tau_2'} \qquad \textsc{Sva}\ \frac{\begin{array}{cc} \tau_1' \le \tau_1 & \tau_2 \le \tau_2' \\ K \sqsubseteq K' & \Delta \sqsubseteq \Delta' \end{array}}{\tau_1 \xRightarrow{K|\Delta} \tau_2 \le \tau_1' \xRightarrow{K'|\Delta'} \tau_2'}$$

$$(\textsc{Tsub})\ \frac{\begin{array}{c} \Gamma; K \vdash e \,:\, \tau' \triangleright H'; \Lambda' \\ \tau' \le \tau \qquad H' \sqsubseteq H \qquad \Lambda' \sqsubseteq \Lambda \end{array}}{\Gamma; K \vdash e \,:\, \tau \triangleright H; \Lambda} \qquad \textsc{Tconst}\ \frac{}{\Gamma; K \vdash c \,:\, \tau_c \triangleright \epsilon; \bot} \qquad \textsc{Tvar}\ \frac{\Gamma(x) = \tau}{\Gamma; K \vdash x \,:\, \tau \triangleright \epsilon; \bot}$$

$$(\textsc{Tif})\ \frac{\Gamma; K \vdash e_1 : int \triangleright H_1;\ \Lambda \qquad \Gamma; K \vdash e_2 : \tau \triangleright H_2;\ \Lambda \qquad \Gamma; K \vdash e_3 : \tau \triangleright H_3;\ \Lambda}{\Gamma; K \vdash if\ e_1\ then\ e_2\ else\ e_3 : \tau \triangleright H_1 \cdot (H_2 + H_3); \Lambda}$$

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

$$\textsc{Tabs}\ \frac{\Gamma, x : \tau_1, f : \tau_1 \xrightarrow{K'|H} \tau_2; K' \vdash e \,:\, \tau_2 \triangleright H; \Lambda}{\Gamma; K \vdash \mathbf{fun}\ f x \Rightarrow e \,:\, \tau_1 \xrightarrow{K'|H} \tau_2 \triangleright \epsilon; \Lambda}$$

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

Figure 4.17: Extending the typing rules for standard ML construct

Definition 4.8.1 (Labelling environment)**.** A labelling environment is a (partial) function $\Lambda\colon Lab_H \to Lab_C$, defined if $h(l) = tell(F)$ or $h(l) = retract(F)$ for some fact F.

Note that a labelling environment needs not to be injective. Furthermore, as usual we denote with $\Lambda[l_1 \mapsto l_2]$ the environment Λ extended with the binding between l_1 and l_2. We introduce the following ordering $\sqsubseteq_\Lambda$ for labelling environment:

$$\Lambda_1 \sqsubseteq_\Lambda \Lambda_2 \text{ iff } \exists \Lambda_3 \text{ such that } dom(\Lambda_3) \cap dom(\Lambda_1) = \emptyset \wedge \Lambda_2 = \Lambda_1 \uplus \Lambda_3$$

The syntax of type is unchanged respect to Section 4.4 but type judgements now have the form $\Gamma; K \vdash e \,:\, \tau \triangleright H;\ \Lambda$, expressing that in the environments Γ and K the expression e has type τ, effect H and yields a labelling environment Λ.

The typing rules, displayed in Figures 4.17 and 4.18, are only extended to deal with the labelling environment. We comment on the most interesting cases only. The labelling environment generated by the rules (Tfact) is $\bot$ (the function always undefined),

$$\textsc{(Tfact)}\quad \frac{}{\Gamma;\, K \vdash F \;:\; fact_{\{F\}} \triangleright \epsilon;\, \bot} \qquad\qquad \textsc{(Tpar)}\quad \frac{K(\tilde{x}) = (\tau,\, \Delta)}{\Gamma;\, K \vdash \tilde{x} \;:\; \tau \triangleright \Delta;\, \bot}$$

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

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

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

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

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

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

Figure 4.18: Extending the typing rules of the constructs for adaptation

because there is no **tell** or **retract**. Instead both (TTELL) and (TRETRACT) update the current environment Λ by associating all the labels of the facts which e can evaluate to, with the label l of the **tell**(e) (**retract**(e), respectively) being typed. The rule (TLET) produces an environment Λ that contains all the correspondences of Λ_1 and Λ_2 coming from e_1 and e_2; note that uniqueness of the labelling is guaranteed by the condition $dom(\Lambda_1) \cap dom(\Lambda_2) = \emptyset$. The other rules are trivial.

Also this new version of the type and effect system is sound and as usual this result derives from the following lemmata:

Lemma 4.8.1 (Preservation). *Let e_s be a closed expression; and let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e_s and such that $\Gamma \vdash \rho \,:\, K$.*
If $\Gamma;\, K \vdash e_s \,:\, \tau \triangleright H_s; \Lambda_s$ and $\rho \vdash C,\, e_s \rightarrow C',\, e'_s$ then $\Gamma;\, K \vdash e'_s \,:\, \tau \triangleright H'_s; \Lambda'_s$ and
(i) $\exists \overline{H}$, s.t. $\overline{H} \cdot H'_s \preccurlyeq H_s$ and $C, \overline{H} \cdot H'_s \rightarrow^{+} C',\, H'_s$ and (ii) $\Lambda'_s \sqsubseteq \Lambda_s$.

Proof (Sketch). The proof is an extension of the one of Lemma 4.4.1. Here, we report only a sketch showing how to deal with (ii).

- rule TTELL $\quad e_s = \mathbf{tell}(e') \quad \Gamma;\, K \vdash e \,:\, fact_\phi \triangleright H;\, \Lambda \quad H_s = \left(H \cdot \left(\sum_{F_i \in \phi} tell\, F_i^{l_i}\right)\right)^{l'}$
  There are two rules through which $\rho \vdash C, e_s \rightarrow C', e'_s$ can be derived:
  - TELL1
    For the premises we have $\rho \vdash C,\, e' \rightarrow C',\, e''$, then $e'_s = tell(e'')$. By inductive hypothesis we have $\Gamma;\, K \vdash e'' \,:\, fact_\phi \triangleright H'';\, \Lambda''$ with $\Lambda'' \sqsubseteq \Lambda$. By rule TTELL it follows that $\Gamma;\, K \vdash e'_s \,:\, unit \triangleright H'_s;\, \Lambda'_s$ with $H'_s = (H'' \cdot \sum_{F_i \in \phi} tell\, F_i^{l_i})^l$ and $\Lambda'_s = \Lambda'' \uplus_{F_i \in \phi} [l_i \mapsto l]$. Since $\Lambda'' \uplus_{F_i \in \phi} [l_i \mapsto l] \sqsubseteq \Lambda \uplus_{F_i \in \phi} [l_i \mapsto l] = \Lambda_s$ the thesis follows.
  - TELL2
    We have $e_s = tell(F)$ so $e'_s = ()$ and $C' = C \cup \{F\}$. It follows that $\Gamma;\, K \vdash () :$ $unit \triangleright \epsilon;\, \bot$, and since $\bot \sqsubseteq \Lambda_s$ the thesis follows.

The other cases follow the same schema. □

Lemma 4.8.2 (Progress). *Let e_s be a closed expression such that $\Gamma; K \vdash e_s \,:\, \tau \triangleright H_s; \Lambda_s$; and let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e_s, and such that $\Gamma \vdash \rho \,:\, K$. If $\rho \vdash C,\, e_s \not\rightarrow$, H is viable for C (i.e. $C,\, H_s \not\rightarrow^{+} C', fail$) and there is no policy violation then e_s is a value.*

Proof. The proof is the same of Lemma 4.4.2. □

Since as said it is not feasible to verify statically the compliance with the security policy, the Progress Lemma requires that no violation occurs. However, if during the loading-time analysis it turns out that the monitor could be always off, this assumption is always satisfied.

Loading-time analysis The analysis specification and the analysis algorithm are unchanged, we need only to label the evolution graph with set of labels. These labels denote which actions **tell**/**retract** are involved in the changes of the context. Formally, the labelled evolution graph is defined as

Definition 4.8.2 (Labelled Evolution Graph). Let H_p be a history expression, C be a context, and $(\Sigma_\circ, \Sigma_\bullet)$ be a valid analysis estimate.
The evolution graph of C is $\mathcal{G} = (N, E, L)$, where

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

Consider again the history expression H_a and its evolution graph $\mathcal{G}$ of Figure 4.15. In $\mathcal{G}$ from the initial context $C = \{F_5, F_8\}$ there is an arc labelled $\{1, 4\}$ to $C' = \{F_1, F_5, F_8\}$ because of *tell* F_1^1 and *tell* F_1^4.

As done in Section 4.6 we can use this graph to verify that the dispatching mechanism always succeeds, but also to detect which contexts violate the policy Φ and to drive the runtime monitor, as we will see in the following.

4.8.4 Code instrumentation

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

Actually, we will guard all the **tell**/**retract** actions in the code, but our runtime monitor will only be invoked on the risky ones. To do that, the compiler (labels the source code as said in Section 4.8.2 and) generates specific calls to *trampoline-like* procedures. We offer a lightweight form of code instrumentation that does not operate on the object code, differently from standard instrumentations. More in detail, we define a procedure, called `check_violation(l)`, for verifying if the policy Φ is satisfied. It takes a label `l` as parameter and has *unit* as return type. Our compilation schema requires to replace every **tell**$(e)^l$ in the source code with the following:

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

where z is fresh. Similarly for **retract**$(e)^l$:

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

where z is fresh too. At loading time, a global mask risky[l] will be assigned for each label l in the source code, by using the information stored in the set *Risky*, as follows:

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

Intuitively the procedure check_violation looks at risky[l]: if the value is false, then the procedure returns to the caller and the execution goes on normally; otherwise it calls for a check on Φ. Its code in a pseudo ML_{CoDa} is:

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

The call ask phi.() triggers the dispatching mechanism to check the policy Φ: if the call fails then a policy violation is about to occur. Then the computation is aborted or a recovery mechanism is possibly invoked.

An easy optimisation is possible when *Risky* is empty, i.e. when the analysis ensures that all the **tell**/**retract** actions are safe, i.e. when no execution path leads to a policy violation. To do that, we introduce the flag always_ok, whose value will be computed at linking time: if it is true, no check is needed. The previous compilation schema is simply refined by testing always_ok before calling check_violation. Thus, every $\textbf{tell}(e)^l$ becomes:

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

Similarly, every $\textbf{retract}(e)^l$ becomes:

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

In this way, the execution time is likely to be reduced, because some costly, and useless security checks are not performed

4.9 Remarks

First of all, we compare ML_{CoDa} with ContextML and the calculi described in Section 1.1.5. All these approaches differ from ML_{CoDa} in a main aspect, namely the *context*, that for them is a stack of layers carrying no data. Furthermore, their notion of context only captures what we called the application context: all the properties holding in the running context are determined by only considering the code of the application. ML_{CoDa} instead introduces the notion of *open* context, the properties of which not only depend on the application code, but *also* on the actual shape of the system context, where the application is about to run. This difference is reflected in the two stages of the static analysis, and justifies the need for a loading time analysis.

Furthermore, in ML$_{\text{CoDa}}$ behavioural variations not are partially defined methods, but they are first-class citizens and there is the notion of context-dependent binding.

Note that the `proceed` construct is strictly related to the idea of representing the context as a stack of layers and it is still open whether it is possible and it makes sense to introduce a similar construct also when the context is a full-fledged declarative database.

Several proposals have been recently put forward to model the context, in most cases, considered a computational entity with its own data model, and rigidly separated from the programming language. These proposals introduce a *context manager* that essentially acts as an interface between the application and the context hosting it. In [CFJ03, WZGP04, GWPZ04], the context is a collection of ontologies, specified in suitable description logic. The burden of serialising data is often demanded to the application programmer. We avoid the impedance mismatch, because our two-component language has a *single* data model, implemented by the virtual machine. Also other proposals, e.g., [vWPK$^+$10, DL05] aim at mitigating the impedance mismatch by representing the context through objects. However, in these last approaches it may be not easy to make complex queries to the context, involving convoluted deductions. The Datalog machinery is an asset of our proposal.

Chapter 5

Conclusion

Self-adaptive software modifies its own behaviour in response to changes in its operational environment. Our work stems from the idea that developing this kind of software requires a shift in programming technologies and methodologies. Consequently, here we addressed some foundational issues of self-adaptability, by adopting an approach firmly based on programming languages and formal methods. In particular, our main concerns was two:

1. introducing appropriate linguist primitives and abstractions to describe the environment hosting the application and to effectively express adaptation;
2. using static analysis techniques, in order to ensure that software adequately reacts to changes in its execution environment.

To do that, we adopted the Context-oriented paradigm which introduces intro programming languages the notions of context and behavioural variations. We discussed and formally tackled the following open issues of this class of languages:

1. semantics foundations have not been sufficiently studied so far;
2. security and formal verification are not main concerns in the design of COP languages;
3. primitives to describe the context are too low level and not powerful enough to model complex working environments, that may include heterogeneous information;
4. the current implementation of behavioural variations, e.g. done trough partial definition of methods or classes, are not sufficiently expressive.

For (1) and (2) in Chapter 2 and Chapter 3 we introduced ContextML, a core of ML. ContextML includes layers as first class values; behavioural variation at expression level; simple constructs for resource manipulation; mechanisms to declare and enforce security policies; and abstract primitives for communicating with other parties by message exchanging. Furthermore, we provided ContextML with a static technique ensuring that software will be able to:

- run in every context arising at runtime;
- communicate with other parties correctly;
- manipulate resources in accordance with security constraints.

In addition, in Chapter 3 we defined an operational semantics for ContextML and proved the correctness of its static analysis.

For addressing also (3) and (4) in Chapter 4 we introduced $\mathrm{ML_{CoDa}}$, an two component language: a declarative constituent for programming the context and a functional one for computing. $\mathrm{ML_{CoDa}}$ is an evolution is a redesign of ContextML. Its bipartite nature stems from the observation that the context requires special constructs and abstractions for its description. For these reasons, we decided the introduction of a logical constituent for the context. Indeed, in $\mathrm{ML_{CoDa}}$ it is a knowledge base implemented as a Datalog program, where the holding properties not only depend on the software code, but also on the system, where it is about to run (open context). Instead, in the functional part the novel constructs are *context-dependent binding* and *first-class behavioural variations*. The first feature allows program variables to assume different values depending on the context. The second one introduces *behavioural variations* as values, allowing us to easily express dynamic and compositional adaptation patterns as well as reusable and modular code.

Furthermore, $\mathrm{ML_{CoDa}}$ is equipped with a (proved sound) static technique to ensure that software will not fail due to an unexpected context arising at runtime. To effectively manage the notion of open context, we perform the static analysis in two phases: at compile time, where we compute an abstraction over-approximating the capabilities required at runtime through a type and effect system; and at loading time where we verify that no failure will arise, because the actual hosting environment satisfies all the required capabilities.

Additionally, we defined a operational semantics for $\mathrm{ML_{CoDa}}$, we proved the soundness of its analyses and we defined a correct algorithm for the loading time phase.

In this thesis we did not consider context-aware systems in which the occurrence of an event may trigger context changes and vice versa. Event-based systems could be encoded in our calculi, by introducing two threads sharing the context. The first senses the external world and listens for incoming events. The second thread runs the code of the program. In this thesis we were only interested in abstractions for programming and in verification techniques, thus we did not consider any construct for events and their management. Introducing in $\mathrm{ML_{CoDa}}$ mechanisms for tackling event-based adaptivity is an interesting future work.

The ongoing research activities consist of:

- designing an inference algorithm for the type and effect systems of our languages (see below);
- evaluating through a proof-of-concept implementation in F# the analysis algorithm presented in Section 4.6;
- speeding up this algorithm by adopting the approaches described in [NNH05, Bou93, AS13].

5.1 Future Work

Besides the introduction of event-based adaptivity in $\mathrm{ML}_{\mathrm{CoDa}}$ there are other extensions we plan to investigate. In this section we consider those we intend to address in the foreseeable future because we judge them the most interesting.

5.1.1 $\mathbf{ML}_{\mathbf{CoDa}}$ Inference Algorithm

We have already started defining an inference algorithm for $\mathrm{ML}_{\mathrm{CoDa}}$. Its design is deeply affected by the subtyping relation and by the need for managing the guessed precondition K' in the premises of the rules Tabs (Figure 4.6) and Tvariation (Figure 4.7).

The notion of subtyping we adopted (see rules Srefl, Sfact, Sfun and Sva in Figure 4.6) is known in the literature as *shape conformant subtyping* [NNH05]. A peculiarity of this relation is that when $\tau_1 \leq \tau_2$ the annotated types τ_1 and τ_2 have the same shape, i.e. a "pure" type purged by annotations. For example, when $\tau_1 = fact_{\{F_1\}}$ and $\tau_2 = fact_{\{F_1,F_2\}}$ we have $\tau_1 \leq \tau_2$ and their shape is *fact*.

To deal with the precondition in the functional and behavioural variation types we need to introduce special variables which represent parameter environments and constraints over these variables which describe the bindings the parameter environment represented by a given variable have to contain.

We follow the approach proposed by Tang and Jouvelot in [TJ95], 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. In each step we generate constraints and then solve them. The constraint generation algorithms are formalized by a set of syntax-directed inference rules. In the first step the rules are characterized by judgements having the form $A; \xi \vdash_u e : t \,\&\, U$, where A is a type environment, ξ is a variable denoting the current parameter environment, e is the expression, t is the type with no annotations and U are the constraints to be solved (elements after the column : are output of the algorithm, the others are input). These constraints are on types and parameter environments. For example, consider the following rule

$$\text{uTell}\quad \frac{A; \xi_1 \vdash_u e : t \,\&\, U}{A; \xi \vdash_u \mathbf{tell}(e) : unit \,\&\, (U \cup \{t = fact\} \cup \{\xi_1 \sqsubseteq \xi\})}\ \xi_1 \text{ fresh}$$

To compute the constraints for the expression $\mathbf{tell}(e)$, we recursively invoke the algorithm over e (for technical reason we pass a fresh variable ξ_1 to the recursive call) and then return *unit* as type and the constraints U of e extended with the requirements that the type of e must be *fact* and that the environment denoted by ξ_1 must be included in the one denoted by ξ. To solve the generated constraints U we use the unification algorithm [MM82] for types and an instance of the worklist algorithm for parameter environments.

We use the type t computed by this step to annotate an expression e, in symbols e^t. This annotated expression is used in the second step, to generate the constraints about the annotations and the effects. The constraint generation algorithm of the second step is formalized by a set of inference rules having the form $\tilde{\Gamma}; \zeta \vdash_a e^t : \tilde{\tau} \triangleright \& S$, where $\tilde{\Gamma}$ is a type environment, ζ is a variable denoting the current parameter environment, e^t is the annotated expression, $\tilde{\tau}$ is the type with annotations, $\tilde{H}$ is the effect and S are the constraints to be solved (elements after the column : are output of the algorithm, the others are input). The constraints in S are about annotations, effects and parameter environments. For example, consider the following rule

$$\frac{\textsc{aFact}}{\tilde{\Gamma}; \zeta \vdash_a F^{\mathit{fact}} : \mathit{fact}_\beta \triangleright \epsilon \,\&\, \{\{F\} \subseteq \beta\}} \; \beta \text{ fresh}$$

For the annotated expression F^{fact} the algorithm, mimicking the rule Tfact of Figure 4.7, returns the annotated type fact_β where β is an annotation variable, the empty effect ϵ and a constraint requiring that the annotation denoted by the variable β includes the fact F. As a further example, consider the rule for functional abstraction

aAbs

$$\frac{\tilde{\tau}_1 \xrightarrow{\zeta'|h} \tilde{\tau}_2 = \mathit{new}(t_1 \to t_2) \qquad \tilde{\Gamma}, x^{t_1} : \tilde{\tau}_1, f : \tilde{\tau}_1 \xrightarrow{\zeta'|h} \tilde{\tau}_2; \zeta' \vdash_a e^{t_2} : \tilde{\tau}_3 \triangleright \tilde{H} \,\&\, S}{\tilde{\Gamma}; \zeta \vdash_a \mathbf{fun}\; f x^{t_1} \Rightarrow e^{t_2} : \tilde{\tau}_1 \xrightarrow{\zeta'|h} \tilde{\tau}_2 \triangleright \epsilon \,\&\, S \cup \mathcal{E}\mathit{ff}(\tilde{\tau}_3 \leq \tilde{\tau}_2) \cup \{\tilde{H} \sqsubseteq h\}}$$

First, we use the "pure" type t_1 and t_2, labelling the variable x and the expression e respectively, to build an annotated type through the function *new*. This function annotates the pure type passed as parameter with fresh annotation variables to be constrained. Then we invoke recursively the algorithm on the expression e, passing a type environment extended with the bindings for x and f, and the parameter environment variable ζ' (returned by the function *new*). For a functional abstraction the algorithm returns the annotated type constructed by the *new* and the empty effect and the constraints resulting from the recursive call suitably extended with constraints implementing the shape conformant subtyping relation ($\mathcal{E}\mathit{ff}(\tilde{\tau}_3 \leq \tilde{\tau}_2)$) and requiring that the latent effect of the function includes the effect of e ($\tilde{H} \sqsubseteq h$). The call $\mathcal{E}\mathit{ff}(\tilde{\tau}_3 \leq \tilde{\tau}_2)$ generates constraints imposing that the annotations of $\tilde{\tau}_3$ (the type resulting from the recursive call) are included in the one of $\tilde{\tau}_2$. The solution of the constraints generated during this phase allow us to reconstruct the type and effect of an expression.

At the moment, we are completing the proof of the correctness of the algorithm with respect to the logical representation and the "principality" property of the computed solution. Intuitively, this property ensures us that for a given expression e we can obtain every type and effect deduced from the logical system possibly enlarging the result of the algorithm.

5.1.2 Quantitative $\mathbf{ML_{CoDa}}$

As said in Chapter 4, representing the context as a Datalog program allows us to easily describe complex situations. Unfortunately, the real world is characterized by scenarios where an answer true/false is not fully adequate, because either the knowledge is often not completely available or there are some preferences we need to take into account. For example, consider the following snippet of code from Section 4.2, in a context where the facts `device(irda)` and `device(bluetooth)` hold:

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

By using the standard Datalog evaluation (top-down), the goal `direct_comm`() will be satisfied by the first clause. Thus, the application will download the URL from the exhibit through IRDA. If many people are observing the same exhibit and are downloading the URL through IRDA, the chosen communication mechanism could be very slow and thus annoying for the user. Indeed, he needs to position his smartphone near the IRDA sensor of the exhibit, and he must wait for his turn because of high number of visitors. Hence, in this scenario Bluetooth could be a better communication mechanism.

A naive solution consists of changing the order of clauses that specify the predicate `direct_comm`. This approach has at least two drawbacks:

1. the programmer has to decide which is the best choice *a priori*, but often he does not have this knowledge at design/programming time;

2. the top-down Datalog evaluation always selects the first true alternative, that may not be the better one, as we argued previously.

A better solution could be to assign to each fact occurring in the definition of `direct_comm` a numerical weight (e.g. bandwidth, or number of users who are using that communication channel, etc.) which is updated by the system and which indicates how good that alternative is:

```
device(irda) ← s1.
device(bluetooth) ← s2.
device(rfid_reader) ← s3.

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

Assuming that we want to select the alternative with maximum weight and that the weights above satisfy $s_2 > s_1$ and $s_2 > s_3$, our application will use bluetooth.

Also the dispatching mechanism of $\mathrm{ML}_{\mathrm{CoDa}}$ has the same limitation: indeed, it selects the first case of a behavioural variation which holds in current context. For example, consider the function (already defined in Section 4.2)

```
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
```

If the smartphone can directly communicate with the exhibit, the first case is selected and the other one is not taken into account. As argued above, there may be situations where the first alternative is not the best one. To address this, we could modify the dispatching mechanism making it parametric with respect a choice function. Then, the new dispatching mechanism evaluates all goals of a behavioural variation and exploits the choice function to select the best alternative.

To extend $\mathrm{ML}_{\mathrm{CoDa}}$ to a quantitative version, we can take advantage of the notion of semiring, i.e. a domain plus two operations satisfying certain axioms, which has been exploited to model many quantitative scenarios. Indeed, the domain of the semiring provides the level of consistency, which can be interpreted as cost, degrees of preferences, probabilities, and others. The two operations define how combine elements together.

In particular, we plan to adopt ideas from Semiring-based Constraint Logic Programming [BMR01] to implement our context and to provide programmers with means to define their own choice functions.

Bibliography

[ABZ12] Dhaminda B. Abeywickrama, Nicola Bicocchi, and Franco Zambonelli. SOTA: Towards a General Model for Self-Adaptive Systems. *2012 IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 0:48–53, 2012.

[AHH$^+$09] 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 1–6, New York, NY, USA, 2009. ACM.

[AHHM11] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented Programming with Java. *Journal of the Japan Society for Software Science and Technology (JSSST) on Computer Software*, 28(1):272–292, 2011.

[AKM11] Tomoyuki Aotani, Tetsuo Kamina, and Hidehiko Masuhara. Featherweight EventCJ: a core calculus for a context-oriented language with event-based per-instance layer transition. In *Proceedings of the 3rd International Workshop on Context-Oriented Programming*, COP '11, pages 1–7, New York, NY, USA, 2011. ACM.

[AS13] Gianluca Amato and Francesca Scozzari. Localizing widening and narrowing. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*, pages 25–42. Springer Berlin Heidelberg, 2013.

[BBF$^+$11] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement Types for Secure Implementations. *ACM Trans. Program. Lang. Syst.*, 33(2):1–45, February 2011.

[BDCDVS02] Piero Bonatti, Sabrina De Capitani Di Vimercati, and Pierangela Samarati. An Algebra for Composing Access Control Policies. *ACM Transactions on Information and System Security*, 5(1):1–35, February 2002.

[BDF09] Massimo Bartoletti, Pierpaolo Degano, and Gian-Luigi Ferrari. Planning and Verifying Service Composition. *Journal of Computer Security*, 17(5):799–837, October 2009.

[BDF$^+$12] Chiara Bodei, Pierpaolo Degano, Gian Luigi Ferrari, Letterio Galletta, and Gianluca Mezzetti. Formalising security in ubiquitous and cloud scenarios. In Agostino Cortesi, Nabendu Chaki, Khalid Saeed, and Slawomir T. Wierzchon, editors, *Computer Information Systems and Industrial Management, CISIM 2012*, volume 7564 of *Lecture Notes in Computer Science*, pages 1–29. Springer, 2012.

[BDFZ09] Massimo Bartoletti, Pierpaolo Degano, Gian-Luigi Ferrari, and Roberto Zunino. Local Policies for Resource Usage Analysis. *ACM Trans. Program. Lang. Syst.*, 31(6):1–43, August 2009.

[BDNN01] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static Analysis for the *pi*-Calculus with Applications to Security. *Information and Computation*, 168(1):68 – 92, 2001.

[BMR01] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based Constraint Logic Programming: Syntax and Semantics. *ACM Trans. Program. Lang. Syst.*, 23(1):1–29, January 2001.

[Bou93] 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.

[C$^+$09] Betty H. C. Cheng et al. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009.

[Car04] Luca Cardelli. *Type Systems*, volume The Computer Science and Engineering Handbook, chapter 97. CRC Press, 2004.

[CC77] 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.

[CC79] Patrick Cousot and Radhia Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM.

[CCT09] Dave Clarke, Pascal Costanza, and Éric Tanter. How should context-escaping closures proceed? In *International Workshop on Context-Oriented Programming*, COP '09, pages 1–6, New York, NY, USA, 2009. ACM.

[CD08] Pascal Costanza and Theo D'Hondt. Feature descriptions for context-oriented programming. In *SPLC (2)*, pages 9–14, 2008.

[CFJ03] H. Chen, T. Finin, and A. Joshi. An ontology for context-aware pervasive computing environments. *The Knowledge Engineering Review*, 18(03), 2003.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[CGT89] S. Ceri, G. Gottlob, and L. 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.

[CH05] 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.

[CS09] Dave Clarke and Ilya Sergey. A semantics for context-oriented programming with layers. In *International Workshop on Context-Oriented Programming*, COP '09, pages 1–6, New York, NY, USA, 2009. ACM.

[DeT02] John DeTreville. Binder, a Logic-Based Security Language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, SP '02, pages 105–113, Washington, DC, USA, 2002. IEEE Computer Society.

[DFGM12a] 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.

[DFGM12b] Pierpaolo Degano, Gian Luigi Ferrari, Letterio Galletta, and Gianluca Mezzetti. Typing context-dependent behavioural variations. In Simon Gay and Paul Kelly, editors, *Proceedings Fifth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2012*, volume 109 of *EPTCS*, pages 28–33, 2012.

[DL05] Pierre-Charles David and Thomas Ledoux. Wildcat: a generic framework for context-aware applications. In *MPAC '05*. ACM, 2005.

[DP02] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[FKNP11] Vojtěch Forejt, Marta Kwiatkowska, Gethin Norman, and David Parker. Automated Verification Techniques for Probabilistic Systems. In Marco Bernardo and Valérie Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 53–113. Springer Berlin Heidelberg, 2011.

[Gal12] Letterio Galletta. A reconstruction of a types-and-effects analysis by abstract interpretation. In *Proceedings Italian Conference of Theoretical Computer Science, ICTCS 2012*, 2012.

[Gal13] Letterio Galletta. An abstract interpretation framework for type and effect systems, 2013. To appear in Fundamenta Informaticae.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2005.

[GL11] Letterio Galletta and Giorgio Levi. An abstract semantics for inference of types and effects in a multi-tier web language. In Laura Kovács, Rosario Pugliese, and Francesco Tiezzi, editors, *Proceedings 7th International Workshop on Automated Specification and Verification of Web Systems, WWV 2011*, volume 61 of *EPTCS*, pages 81–95, 2011.

[GPS11] Carlo Ghezzi, Matteo Pradella, and Guido Salvaneschi. An Evaluation of the Adaptation Capabilities in Programming Languages. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 50–59, New York, NY, USA, 2011. ACM.

[GWPZ04] 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, 2004.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, March 2008.

[HIM11] 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.

[HM08] Markus C. Huebscher and Julie A. McCann. A Survey of Autonomic Computing&Mdash;Degrees, Models, and Applications. *ACM Comput. Surv.*, 40(3):1–28, August 2008.

[HMS06] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability Classes for Enforcement Mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, January 2006.

[Hop79] *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company, 1979.

[IBMon] IBM. An architectural blueprint for autonomic computing. Technical report, June 2006. Fourth Edition.

[IHM12] Atsushi Igarashi, Robert Hirschfeld, and Hidehiko Masuhara. A Type System for Dynamic Layer Composition. In *FOOL 2012*, page 13, 2012.

[IPW01] 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, May 2001.

[JM09] Ranjit Jhala and Rupak Majumdar. Software Model Checking. *ACM Comput. Surv.*, 41(4):1–54, October 2009.

[KAM11] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: A Context-oriented Programming Language with Declarative Event-based Context Transition. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 253–264, New York, NY, USA, 2011. ACM.

[KBE99] M.M. Kokar, K. Baclawski, and Y.A. Eracar. Control theory-based foundations of self-controlling software. *Intelligent Systems and their Applications, IEEE*, 14(3):37–45, 1999.

[KC03] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, 2003.

[KHH$^+$01] 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.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 - European Conference of Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997.

[Koz99] Dexter Kozen. Language-based security. In *Proceedings of the 24th International Symposium on Mathematical Foundations of Computer Science*, MFCS '99, pages 284–298, London, UK, UK, 1999. Springer-Verlag.

[KSS09] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data Flow Analysis: Theory and Practice.* CRC Press, 2009.

[KW04] J.O. Kephart and W.E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *In Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks. POLICY 2004.*, pages 3–12, 2004.

[Lad97] Robert Laddaga. Self Adaptive Software. Technical report, DARPA BAA, 1997. An excerpt is available at `http://people.csail.mit.edu/rladdaga/BAA98-12excerpt.html` (last access Dec 2013).

[LDB10] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical Model Checking: An Overview. In *Proceedings of the First International Conference on Runtime Verification*, RV'10, pages 122–135, Berlin, Heidelberg, 2010. Springer-Verlag.

[Li05] Sing Li. An Introduction to AOP, 2005. Available at`http://www.ibm.com/developerworks/java/tutorials/j-aopintro/` (last access Oct 2011).

[LM03] Ninghui Li and John C. Mitchell. DATALOG with Constraints: A Foundation for Trust Management Languages. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, PADL '03, pages 58–73, London, UK, UK, 2003. Springer-Verlag.

[Lok04] Seng W. Loke. Representing and reasoning with situations for context-aware pervasive computing: a logic programming perspective. *Knowl. Eng. Rev.*, 19(3):213–233, 2004.

[LWZ06] Jay Ligatti, David Walker, and Steve Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming*, 63(3):240 – 266, 2006.

[LYBB13] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The Java® Virtual Machine Specification, 2013. Available at`http://docs.oracle.com/javase/specs/jvms/se7/html/index.html` (last access Apr 2014).

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[MO84] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for prolog. *Artificial Intelligence*, 23(3):295 – 307, 1984.

[NL98] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In *Mobile Agents and Security*, pages 61–91, London, UK, UK, 1998. Springer-Verlag.

[NN99] Flemming Nielson and HanneRiis Nielson. Type and effect systems. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 114–136. Springer Berlin Heidelberg, 1999.

[NN02] Hanne Riis Nielson and Flemming Nielson. The essence of computation. chapter Flow logic: a multi-paradigmatic approach to static analysis, pages 223–244. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[NNH05] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1st ed. 1999. corr. 2nd printing, 1999 edition, 2005.

[OGT$^+$99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, May 1999.

[OT11] 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.

[PAG03] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time Aspects: Efficient Dynamic Weaving for Java. In *Proceedings of the 2Nd International Conference on Aspect-oriented Software Development*, AOSD '03, pages 100–109, New York, NY, USA, 2003. ACM.

[PCZ13] Mariachiara Puviani, Giacomo Cabri, and Franco Zambonelli. A Taxonomy of Architectural Patterns for Self-adaptive Systems. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, C3S2E '13, pages 77–85, New York, NY, USA, 2013. ACM.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[Ros04] Eva Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, 2004.

[SGP11] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A programming paradigm for autonomic systems. *CoRR*, abs/1105.0069, 2011.

[SGP12] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801–1817, 2012.

[SMH01] Fred B. Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In Reinhard Wilhelm, editor, *Informatics*, volume 2000 of *Lecture Notes in Computer Science*, pages 86–101. Springer Berlin Heidelberg, 2001.

[SSVH08] Christian Skalka, Scott Smith, and David Van Horn. Types and trace effects of higher order programs. *Journal of Functional Programming*, 18:179–249, 3 2008.

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, May 2009.

[SW01] Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

[TJ95] 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.

[VW86] Moshe Y Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*. IEEE Computer Society, 1986.

[VW08] Moshe Y. Vardi and Thomas Wilke. Automata: from logics to algorithms. In Jörge Flum, Erich Grädel, and Thomas Wilke, editors, *Logic and Automata: History and Perspective*, pages 629–736. Amsterdam University Press, 2008.

[vWPK$^+$10] 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.

[WKD04] 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.

[WZGP04] 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.

[WZL03] David Walker, Steve Zdancewic, and Jay Ligatti. A Theory of Aspects. *SIGPLAN Not.*, 38(9):127–139, August 2003.

[XP99] Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 214–227, New York, NY, USA, 1999. ACM.

[ZJ97] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, January 1997.