A Context-Oriented Extension of F# *

Andrea Canciani Pierpaolo Degano Gian-Luigi Ferrari Letterio Galletta
Dipartimento di Informatica, Università di Pisa, Italy
{canciani,degano,giangi,galletta}@di.unipi.it

Context-Oriented programming languages provide us with primitive constructs to adapt program behaviour depending on the evolution of their operational environment, namely the context. In previous work we proposed ML_{CoDa}, a context-oriented language with two-components: a declarative constituent for programming the context and a functional one for computing. This paper describes the implementation of ML_{CoDa} as an extension of F#.

1 Introduction

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

Using standard programming languages, context-awareness is usually implemented by modelling the context through a special data structure, which can answer to a fixed number of queries, whose results can be tested through `if` statements. This approach however charges the programmer with the responsibility of implementing this data structure and the corresponding operations. In addition, she is responsible of achieving a good modularisation, as well as a good separation of cross-cutting concerns, and of the interactions between the code using the context and legacy code. Successful examples are offered by special design patterns [21] and Aspect-Oriented Programming [17] that encapsulate context-dependent behaviour into separate modules (objects and aspects, respectively), but leave most of the burden of handling the context and its changes (and their correctness) to the programmers. Therefore, new dynamic development models and programming language constructs are required to effectively support the *evolution* and *adaptation* of applications to changes of the context [5].

Recently, Context Oriented Programming (COP) [8] was proposed as a viable paradigm to develop context-aware software. It advocates languages with suitable constructs for adaptation, so to express context-dependent behaviour in a modular manner. In this way adaptivity is built-in, and the provided linguistic abstractions impose a good practice to programmers, with a positive effect on correctness and modularity of code, mainly because low level details about context management are masked by the compiler.

In previous papers [10, 9] we followed this line of research from a foundational perspective and we proposed a programming language core, called ML_{CoDa}, specifically designed for adaptation and equipped with a clear formal semantics. It has two components: a logic constituent for programming the context and a functional one for computing. The logical component provides high level primitives for describing and interacting with complex working environments. The functional component supports

*Work partially supported by the MIUR-PRIN project Security Horizon.

José Proença and Massimo Tivoli (Eds.):
14th International Workshop on Foundations of
Coordination Languages and Self-Adaptive Systems (FOCLASAŠ15)
EPTCS 201, 2015, pp. 18–32, doi:10.4204/EPTCS.201.2

programming of a variety of adaptation patterns. Its higher-order facilities are essential to exchange the bundle of functionalities required to manage adaptivity changes. Moreover, ML_{CoDa} offers a further support through a static analysis that guarantees programs to always be able to adapt in every context [9].

In addition to the formal aspects of ML_{CoDa} studied in [9], a main feature of our approach is that a single and fairly small set of constructs is sufficient enough for becoming a *practical* programming language, as shown in the present paper. Indeed, ML_{CoDa} can easily be embedded in a real programming eco-system, in our case .NET, so preserving compatibility with its future extensions and with legacy code. Being part of a well supported programming framework, our proposal minimises the learning cost and lowers the complexity of deploying and maintaining applications.

This paper illustrates a prototypical implementation of ML_{CoDa} as an extension of the (ML family) functional language F# through a simple case study. Indeed, no modifications at all was needed to the available compiler and to its runtime. We exploited F# metaprogramming facilities, such as code introspection, quotation and reflection, as well as all the features provided by .NET, including a vast collection of libraries and modules, and in particular a Just-In-Time (JIT) mechanism for compiling to native code. All in all, ML_{CoDa} is implemented as a standard .NET library.[1] In the path towards the implementation a main role has been played by the formal description of the language. In particular, the ML_{CoDa} formal semantics highlights and explains how the interactions between the two components occur. Indeed, the crucial parts of the implementation toolchain, compilation, generated code and run-time structures, together with their interactions, are *formally* identified and described. As a matter of fact, the formal semantics gave us a great support, because it has been used to drive the definition of all the fine-grained control mechanisms governing ML_{CoDa} implementation.

Plan of the paper In Section 2, we briefly review the design of the language and of its two components. Section 3 is a gentle introduction to our COP language, through which we describe a simple case study, simulating an e-Healthcare system. In Section 4, we present our approach to the implementation of ML_{CoDa} that is then discussed in Section 5; the same section also briefly compares our work with the relevant literature. Finally, we draw some conclusions and present future work in Section 6.

2 Design of ML_{CoDa}

We survey ML_{CoDa}, a core functional programming language, equipped with linguistic primitives for context-awareness and coupled with a logical language, for context definition and management. We first present how the context is described, and then the high level primitives for adaptation. We omit discussing the two-phase static analysis designed for ML_{CoDa} [9]. Briefly, it detects if an application will be able to adapt to its execution contexts: at compile time, the first phase safely approximates the actions performed on the context; at loading time the second phase exploits the approximation above to check that the program will adapt to *all* the contexts that may occur at runtime.

The context The notion of *context*, i.e. the environment where applications run, is fundamental for adaptive software. Intuitively, it is a heterogeneous collection of data coming from different sources and having different representations. Some of these data are application independent, like those about the hardware capabilities, e.g. screen resolution, and about the physical environment, e.g. the location of the user; other data are application-dependent like user's preferences, e.g. the language of the user. It is

[1] Available at `https://github.com/vslab/fscoda`

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

In ML_{CoDa}, the context is split in two coarse parts: the *system* and the *application context*, following a well-established practice that separates data coming from inside the application and those coming from outside [22]. Both parts are represented and manipulated in a uniform way, so guaranteeing compatibility and modularity. Also, a programmer needs tools to access and manipulate all kind of contextual data in a easy and uniform way. Indeed, the context is developed by requirements engineers [26], that have tools and skills different from those needed for building applications [20, 18]. This methodological issue, as well as separation of concerns motivated us to define ML_{CoDa} as a two-component language: a declarative constituent for programming the context and a functional one for computing.

The declarative approach allows programmers to express *what* information the context has to include, leaving to the virtual machine *how* this information is actually collected and managed. For us, a context is a knowledge base and we implement it as a Datalog program, following a well-studied approach [20, 18], within established methodologies [15, 11]. In other words, a context in ML_{CoDa} is a set of facts that predicate over a possibly rich data domain, and a set of logical rules that permit to deduce further implicit properties of the context itself. With this representation, adaptive programs can query the context and retrieve relevant information, by simply verifying whether a given property holds in it, i.e. by checking a Datalog goal. Note that deduction in Datalog is fully decidable in polynomial time [7].

Adaptation As for programming adaptations, the functional part of the language provides two mechanisms. The first is *context-dependent binding* 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 an additional Datalog goal therein: `dlet x = e1 when goal in e2`. The variable `x` (called *parameter* hereafter) may denote different objects, with different behaviour depending on the different properties of the current context, as required by `goal`. This is a major aspect of adaptivity.

The second mechanism is based on *behavioural variations*: a chunk of behaviour that can be activated depending on information picked up from the context, so to dynamically adapt the running application. This construct implements the fundamental concept of the COP paradigm, and in ML_{CoDa} it is a list of pairs `goal.expression`, similar to pattern-matching, that alters the control flow of applications depending on the context. When a behavioural variation is executed, parts of the deployed code are suitably selected to meet the new requirements. Behavioural variations have parameters and they are values, so they can be referred to by identifiers; passed and returned by functions; supplied by the context; and composed with existing ones. This facilitates programming dynamic adaptation patterns, as well as reusable and modular code.

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

3 A simulator of an e-Healthcare system

In this section we illustrate and discuss the main features of ML_{CoDa} and how these affect the development of an application. In the following example, we consider a fragment of an e-Healthcare system with a few aspects typical of the Internet of Things. In particular, we discuss how physicians can access the medical data through their devices and how this system can help them to plan some medical activities.

An e-Healthcare scenario In our scenario each physician is provided with a device, e.g. a smartphone or a tablet, which tracks her location and enables her to retrieve a patient's clinical record. On the basis of the obtained data, the doctor can decide which exams the patient needs and the system helps scheduling them. Additionally, the system checks whether the doctor has the competence and the permission to actually perform the required exam, otherwise it may suggest another physician who can, possibly coming from another department. When a doctor moves from a ward to another, her operating context changes, in particular she can access the complete clinical records of the patients therein. The application must adapt to the new context and it may additionally provide different features, e.g. by disabling rights to use some equipment and by acquiring access to new ones.

The e-Healthcare context We consider a small portion of our e-Healthcare system, and we show how to declaratively describe the relevant contextual information, through Datalog. In particular, we take into account the part of the context that stores and makes available some data about the doctors' location, information about their devices, the patients' medical records and the ward medical equipment. Some basic data are asserted by Datalog facts, and one can retrieve further information through the inference machinery of Datalog, that uses logical rules, also stored in the context.

For example, the fact that Dr. Turk is in cardiology is rendered by the fact

```
physician_location('Dr. Turk', 'Cardiology').
```

The following inference rule permits to deduce that a doctor can access the clinical data of patients in the same department where she is:

```
physician_can_view_patient(Physician, Patient) :-
        physician_location(Physician, Location),
        patient_location(Patient, Location).
```

This rule states that the predicate on the left hand-side of the implication operator `:-` holds when the conjunction of the predicates (`physician_location` and `patient_location`) in the right hand-side yields true, i.e. when the physician and patient's location are the same. The $\mathrm{ML}_{\mathrm{CoDa}}$ context is quite expressive and allows us to model fairly complex situations. For example, sometimes the patients could be prescribed an exam which can only be performed after some other screenings. Therefore, to compute the list of exams a patient needs, we have to take into account all the dependencies among them. This could be encoded in the context through the following recursive rules:

```
patient_needs_result(Patient, Exam) :-
        patient_has_been_prescribed(Patient, Exam).

patient_needs_result(Patient, Exam) :-
        exam_requirement(TargetExam, Exam),
        patient_needs_result(Patient, TargetExam).
```

The first rule states that the prescription of an exam implies that the involved patient needs the results of the test. The second rule says that whenever a patient needs an exam, she also needs all the screenings the exam depends on. Datalog provides a convenient way to model recursive relations like the dependency among exams, that may require involved queries with standard relational databases.

The next rule dictates that a patient has to do an exam if the two clauses in the right hand-side are true. The first has been already discussed above, while the second clause says that a patient should *not* do an exam if its results are already known (in the rule below the operator `\+` denotes the logical *not*). [2]

[2] Our version of Datalog only admits safe and stratified programs, so to effectively cope with negation [7].

```
patient_should_do(Patient, Exam) :-
        patient_needs_result(Patient, Exam),
        \+ patient_has_result(Patient, Exam).
```

In addition, physical objects can be declaratively described in quite a similar, homogeneous manner. The following (simplified) rule specifies when a device can display a certain exam, by checking whether it has the needed capabilities:

```
device_can_display_exam(Device, Exam) :-
        device_has_caps(Device, Capability),
        exam_view_caps(Exam, Capability).
```

Asserting the capabilities of a device is straightforward, by listing a set of facts, e.g.

```
device_has_caps('iPhone 5', '3D acceleration').
device_has_caps('iPhone 5', 'Video codec').
device_has_caps('iPhone 5', 'Text display').
device_has_caps('Apple Watch', 'Text display').
```

Adaptation constructs Now, we show how context-dependent bindings and behavioural variations allow a programmer to express program behaviour which depends on the context in our e-Healthcare system. When a doctor enters a department and visits some patients, she can display the patients' medical records on her personal device. Moreover, the e-Healthcare system computes the list of the clinical exams a patient should do and that the doctor can perform. The following code implements these functionalities, and shows all the adaptation constructs of ML_{CoDa}. The `display` function below takes a doctor `phy` and a patient `pat` as arguments and prints on the screen the information about the patient's exams.

```
1  let display phy pat =
2    match ctx with
3    | _ when !- physician_can_view_patient(phy, pat) ->
4      match ctx with
5      | _ when !- patient_has_result(pat, ctx?e) ->
6        printfn "%s sees that %s has done:" phy pat
7        for _ in !-- patient_has_result(pat, ctx?exam) do
8          display_exam phy ctx?exam
9      | _ ->
10       printfn "%s sees that %s has done no exam" phy pat
11
12     let next_exam = "no exam" |- True
13     let next_exam = ctx?exam |-
14                       (physician_exam(phy, ctx?exam),
15                        patient_active_exam(pat, ctx?exam))
16     printfn "%s can submit %s to %s" phy pat next_exam
17   | _ ->
18     printfn "%s cannot view details on %s" phy pat
```

Actually, the code above is F#, a dialect of ML. Since we wanted to keep the F# parser unmodified, the syntax is slightly different from the one used in [9] and recalled in Section 2. This is only an implementation detail, because a simple macro-expansion suffices to translate the original syntax in the intermediate notation used in this section.

A behavioural variation has the form `match ctx with | _ when !- Goal -> expression`. The sub-expression `match ctx with` explicitly refers to the context; the part `| _ when !- Goal` introduces the goal to solve; and `-> expression` is the sub-expression to evaluate when the goal is true.

The outermost behavioural variation (starting at line 2) checks whether the doctor `phy` is allowed to inspect the data of the patient `pat`, as granted when the goal `physician_can_view_patient(phy, pat)` at line 3 holds.

The nested behavioural variation (line 4) checks if the patient has got the results of some exams, through the predicate `patient_has_result`. If this goal holds, the `for` construct extracts the list of exams and results from the context (line 7). The statement `for _ in !-- Goal do expression` iterates the evaluation of `expression` over all the solutions of the `Goal`. In other words, it is an iterator on-the-fly, driven by the solvability of the goal in the context. Note that the predicate `patient_has_result` at line 7 contains the *goal variable* `ctx?exam`: if the query succeeds, at each iteration `ctx?exam` is bound to the current value satisfying `Goal`. A goal variable is introduced in a goal, defining its scope, through the syntax `ctx?var_name`.

Finally, the function `display` prints an exam that can be performed next on the patient `pat` by the physician `phy`, by using the construct for the context-dependent binding. Here we write it in the following form `let x = expression1 |- Goal [in] expression2`, where the keyword `let` and the operator `|-` replace `dlet` and `when`, used in Section 2. At lines 12-13 we declare by cases the parameter `next_exam`, that is referred to in line 16. Which case applies and which value will be bound to `next_exam` can only be determined at runtime when the parameter is used, because they depend on the actual context. If the goal in lines 14-15 hold, then `next_exam` assumes the value retrieved from the context, otherwise it gets the default value `"no exam"`. (The predicate `True` always holds independently of the context.)

Actually, it may happen that no goal is satisfied in a context during the execution of a behavioural variation or during the resolution of a parameter. This reflects the inability of the application to adapt, either because the programmer assumed at design time the presence of functionalities that the current context lacks, or because of programming errors. This is a new class of runtime errors that we call *adaptation failures*. For example, the following function assumes that given the identifier of a physician, it is always possible to retrieve her location from the context through the `physician_location` predicate:

```
let find_physician phy =
  let loc = ctx?location |-
              physician_location(phy, ctx?location) in
  loc
```

If the function `find_physician` is invoked on a physician whose location is not stored in the context, e.g. due to a programming error, the context-dependent binding will fail to find a solution for the goal. In such a case the current implementation throws a runtime exception. This is a common pattern in languages like F#, which makes behaviour both easy to manage and to integrate with existing code. In the current implementation we use the standard F# construct to handle an adaptation failure as shown in the following snippet of code:

```
let find_physician phy =
  try
    let loc = ctx?location |-
                physician_location(phy, ctx?location) in
    loc
  with e ->
    printfn "WARNING: cannot locate %s:\n%A" phy e
    "unknown location"
```

A more sophisticated approach involves statically determining whether the adaptation might fail and reporting it before running the application, as described in [9].

The interaction with the Datalog context is not limited to queries: it is possible to program the modifications to the knowledge base on which it performs deduction. Adding or removing facts is done

by the `tell` and `retract` operations, as in:

```
tell <| patient_has_result("Alice", "CT scan")
```

Some execution examples We now show how the functions defined above give different results when invoked in different contexts, some parts of which will only be described intuitively.[3]

For example, in a context where Dr. Turk is not in the same ward as Bob

```
display "Dr. Turk" "Bob"
```

outputs

```
Dr. Turk cannot view details on Bob
```

because physicians are only allowed to see data about the patients in the department where they are. In particular, the behavioural variation on `physician_can_view_patient` at line 3 finds out that the operation is not allowed. If instead Dr. Cox is in the same department where Bob is, the call

```
display "Dr. Cox" "Bob"
```

correctly prints the details about Bob (actually these are stored in the Datalog knowledge base):

```
Dr. Cox sees that Bob has done no exam
Dr. Cox can submit Bob to Blood test
```

In this case the outermost behavioural variation (starting at line 2) confirms that Dr. Cox can view the data. The nested one (starting at line 4), driven by `patient_has_result`, finds no exam for Bob, hence the function displays the no-exam message (line 10). Moreover, the program finds out that Dr. Cox could do a blood test on Bob, as he is enabled to, and additionally Bob needs no pre-screening and so that exam can be done immediately, because the predicate at line 15 holds.

We now consider a slightly more complex situation, in which the context itself is modified. Patient Alice has already performed an EEG test, and doctors prescribed her a CT and nothing else. Dr. Kelso is in Alice's room, is enabled to do only CT tests and carries a device on which he can visualise the results. In this context

```
display "Dr. Kelso" "Alice"
```

outputs

```
Dr. Kelso sees that Alice has done:
  - EEG
Dr. Kelso can submit Alice to CT scan
```

The main difference from the case above is that Alice has already performed an exam, which is hence listed by the iteration construct. Now Dr. Kelso performs a CT scan on Alice and thus the context has to be accordingly changed, by asserting the fact

```
tell <| patient_has_result("Alice", "CT scan")
```

The same fragment of code above

```
display "Dr. Kelso" "Alice"
```

has a different output in the modified context

[3] For a full definition of the code see `https://github.com/vslab/fscoda`.

```
Dr. Kelso sees that Alice has done:
 - EEG
 - CT scan
Dr. Kelso can submit Alice to no exam
```

Besides displaying a longer list of exam results, the application shows Dr. Kelso that Alice needs him to perform no other exam.

Assume Dr. Cox moves to Alice's room and checks her medical report, but he has a device that cannot show CT images

```
display "Dr. Cox" "Alice"
```

prints the following

```
Dr. Cox sees that Alice has done:
 - EEG
 - CT scan (current device cannot display the exam data)
Dr. Cox can submit Alice to no exam
```

Since the CT scan cannot be displayed by the device, the `display_exam` function warns the doctor and it might present the results in a more limited form, e.g a static thumbnail.

The e-Healthcare system might also help Bob in finding out the physicians that can visit him:

```
for _ in !-- (patient_active_exam("Bob", ctx?exam),
              physician_exam(ctx?physician, ctx?exam)) do
  printfn "%s (currently in %s) can submit Bob to %s"
    ctx?physician (find_physician ctx?physician) ctx?exam
```

This fragment prints the name and the position of all of the physicians who can perform an exam which Bob needs. This code snippet relies on the second version of `find_physician`, which handles adaptation failures by logging the error and returning `"unknown location"` as a result. Assuming that Dr. Turk has left the hospital and Dr. Cox is in the cardiology department, the output would be:

```
Dr. Cox (currently in Cardiology) can submit Bob to Blood test
WARNING: cannot locate Dr. Turk:
CoDa.InconsistentContext: Context inconsistency detected
Dr. Turk (currently in unknown location) can submit Bob to Blood test
```

Since it is not possible to deduce the location of Dr. Turk from the context, the context-dependent binding in `find_location` fails. Nonetheless the program can continue the execution after handling the exception.

4 Internals of a prototype compiler

In order to implement ML_{CoDa}, we found it convenient to build upon a functional language and to integrate it with a Datalog engine, as easily as possible. As said, our choice has been F# which is commercially supported, and fully integrated in the .NET eco-system. We exploited the metaprogramming facilities available in F# for the ML_{CoDa} adaptation constructs to avoid re-implementing well-known primitives, while the deduction engine is that of YieldProlog, available as a .NET library. The bipartite nature of ML_{CoDa} fosters the independence between context and application development. This design choice is well supported by .NET through the notion of *assemblies*. They work just as modules and offer us a natural way to separate the code of the context from that of the application.

Implementing the context A requirements engineer writes several Datalog sources describing the context in hand. These files are ahead-of-time translated to a .NET code by our compiler `ypc`, which is based on a *customised* version of the library YieldProlog.[4] YieldProlog works as our Datalog engine, handling the context state and solving goals. Our translation compiles each predicate into a method, whose code enumerates one by one the solutions, i.e. the assignments of values to variables which satisfy the predicate. In this way, the interaction and the data exchange between the application and the context is fully transparent to the programmer because the .NET type system is uniformly used everywhere. Indeed, data inside the context are instances of the class `object`, hence the programmer can insert any object in the context as long as the method `Equals` is appropriately overridden. This is required because the Datalog engine during the deduction process needs to check if two objects (considered as atoms by the engine) are equal.

For example, the predicate

```
patient_needs_result(Patient, Exam) :-
        patient_has_been_prescribed(Patient, Exam).

patient_needs_result(Patient, Exam) :-
        exam_requirement(TargetExam, Exam),
        patient_needs_result(Patient, TargetExam).
```

is translated into

```
1  public static IEnumerable<bool> patient_needs_result(object Patient, object Exam)
2  {
3          foreach (bool l2 in patient_has_been_prescribed(Patient, Exam))
4              yield return false;
5
6          Variable TargetExam = new Variable();
7          foreach (bool l2 in exam_requirement(TargetExam, Exam))
8              foreach (bool l3 in patient_needs_result(Patient, TargetExam))
9                  yield return false;
10 }
```

The enumeration of goal solutions works through side-effects, by modifying the values of the input parameters, i.e. the return value (which is always `false`) is never used. The loop at lines 3-4 returns to the caller for each solution of `patient_has_been_prescribed`, because the `Patient` and `Exam` pair is a solution according to the first rule. At line 6, the compilation of the second rule introduces the variable `TargetExam`, because it appears free in the body. The conjunction of subgoals is obtained by two nested loops, so that the statement at line 9 is reached only if both loops enumerate consistent solutions. The unification algorithm, implemented by the class `Variable`, ensures the consistency of the solutions computed by the different subgoals. The recursive definition of `patient_needs_result` requires no special handling as it is implemented as a recursive method in a straightforward way.

Functional part The application programmer writes F# code, annotating the functions which use ML_{CoDa} extensions with *custom attributes* (see code below) and starting the ML_{CoDa} runtime. Since the operations needed to adapt the application to contexts are transparently handled by our runtime support, the compiler `fsharpc` works as it is. Indeed, the ML_{CoDa}-specific constructs are just-in-time replaced in a single step by their F# implementation when they are about to be run. A simple example follows.

[4] Available at `https://github.com/vslab/YieldProlog`

```
1  [<CoDa.Code>]
2  module Physicians.Test
3
4  open CoDa.Runtime       // the runtime
5  open Physicians.Facts   // functional to logic typed interface
6  open Physicians.Types   // logic to functional typed interface
7  open ...                // other libraries used by the application
8
9
10 [<CoDa.Context("pysician-ctx")>]
11 [<CoDa.Context("patients-ctx")>]
12 [<CoDa.Context("devices-ctx")>]
13 [<CoDa.EntryPoint>]
14 let main () =
15   display "Dr. Turk" "Bob"
16   display "Dr. Cox" "Bob"
17   printfn ""
18   display "Dr. Cox" "Alice"
19   display "Dr. Kelso" "Alice"
20   printfn ""
21
22   // Other code
23 do
24   run ()
```

The attribute `CoDa.Code` marks the ML_{CoDa}-specific constructs which need to be transformed, e.g. the above module `Physicians.Test`. Actually, the attribute `CoDa.Code` is an alias for the standard `ReflectedDefinitionAttribute`, that marks modules and members whose abstract syntax trees (AST) are used at runtime through reflection. Note that ML_{CoDa}-specific operations are only allowed in methods marked with this attribute; otherwise an exception is raised when they are invoked.

The attribute `CoDa.EntryPoint` marks the principal function of the application (`main` above). When the ML_{CoDa} runtime is initialised and started through the function `run`, it looks for the function `f` marked by `CoDa.EntryPoint`; then it transforms the code of the function replacing the ML_{CoDa}-specific constructs with their F# object code; and finally runs the obtained object code. The translation is performed on the AST represented in the form of quotations. Hereafter, for readability, we will show the object code in F# syntax, rather than the quotation emitted by the JIT compiler.

The lines 10-12 in the code show that the context is conveniently split in distinct modules. The attribute `CoDa.Context` describes which of its parts are needed for the application. The runtime initialises the context and links it with the application, before running it. The code that initialises the context of the e-Healthcare system behaves as the following one:

```
[<CoDa.Code>]
module Physicians.Context

// code to import Runtime, Types, Facts (see lines 4-7 in the code above)

[<CoDa.ContextInit>]
let initFacts () =
  tell <| physician_exam("Dr. Cox", "ECG")
  tell <| physician_exam("Dr. Cox", "Blood test")
  // other code
```

The ML_{CoDa} construct `tell` adds facts in the context (there is also a `retract` to remove facts); it is implemented as a method of the `Runtime.context` object, only accessible by the ML_{CoDa} runtime.

The modules `Facts` and `Types` provide the interface between the functional code and the context. In particular, they contain utility functions for typing facts and predicates in F#, respectively. The above function `initFacts` is compiled as

```
let initFacts () =
  Runtime.context.Tell <| physician_exam("Dr. Cox", "ECG")
  Runtime.context.Tell <| physician_exam("Dr. Cox", "Blood test")
  // other code
```

We now discuss how the two main constructs for expressing adaptation, *behavioural variations* and *context-dependent binding*, are implemented in our e-Healthcare system. Consider the function `display` defined in Section 3. The behavioural variations of lines 2-5 are compiled as

```
if Runtime.context.Solve([|physician_can_view_patient(phy, pat)|],
                         null) then
  let solution1 = new Dictionary<string, obj>()
  solution1.["e"] <- new Variable()
  if Runtime.context.Solve([|patient_has_done(pat, solution1.["e"])|],
                           solution1) then
    ...
  else
    printfn "%s sees that %s has done no exam" phy pat
else
  printfn "%s cannot view details on %s" phy pat
```

The `match` expression becomes an `if`, whose guard queries the context by the method `Solve`. The above translation also illustrates the mechanism that implements the goal variables. The dictionary `solution1` is initialised with the variable introduced in the goal, namely `ctx?e` of line 5. The dictionary is then passed to the Datalog solver as second argument. If a solution is found, `solution1` contains an assignment to the value that satisfies the goal, and allows us to access the solution, but only within the scope of the corresponding `if` expression. Note that the outermost behavioural variation uses no goal variables, hence no dictionary is needed (the second argument of `Solve` is `null`).

The `for` construct follows the same schema; the iteration of line 7 is translated as

```
let solution2 = new Dictionary<string, obj>()
solution2.["exam"] <- new Variable()
let variables0 = new Dictionary<string, obj>(solution2)
for _ in Runtime.context.Enumerate([|patient_has_done(pat, solution2.["exam"])|],
                                   variables0, solution2) do
  Runtime.callTramp display_exam [| phy; solution2.["exam"] |]
```

The method `Enumerate` returns a stream that iterates over the goal solutions by storing them in the dictionary `solution2`. The original body of the `for` loop consisted of a call to the function `display_exam` (line 8), which is instrumented by interposing the method `callTramp`. This is the hook used by the JIT compiler to overtake control when an ML_{CoDa} function needs to be translated. For merely technical reasons, the dictionary `variables0`, which initially is a clone of `solution2`, is passed to the method `Enumerate`.

The binding of the parameter `next_exam` (lines 12-16) is implemented by the following snippet

```
let next_exam () =
  if Runtime.context.Solve([| Runtime.True |], null) then
    "no exam"
  else
    raise <| new InconsistentContext()
let next_exam () =
  let solution2 = new Dictionary<string, obj>()
  solution2.["exam"] <- new Variable()
  if Runtime.context.Solve([| physician_exam(phy, solution2.["exam"]);
                              patient_active_exam(pat, solution2.["exam"]) |],
                              solution2) then
    solution2.["exam"]
  else
    next_exam ()
printfn "%s can perform %s on %s" phy (next_exam ()) pat
```

Recall that parameters are evaluated in the context where they are referred to (line 16 in the original code), in a lazy way. This is rendered by the application `next_exam ()` in the `printfn` statement above. This resolves to the innermost definition of the function `next_exam`, which checks the goal at lines 14-15. If the runtime finds a solution, the `then` branch evaluates to the goal variable `ctx?exam`; otherwise the task of determining the binding is delegated to the outermost `let`. If even the outermost binding fails, an exception is raised to signal that the application is unable to adapt to the current context. However, here this will never be the case, because the predicate `Runtime.True` always holds.

5 Discussion

Here, we briefly discuss the relevant literature on COP languages, and we refer the reader to the survey by Salvaneschi et al. [24] on the design of languages, and to that by Appeltauer et al. [2] on some implementations. Most of the proposals of COP languages (to cite just a few: *JCop* [3], *ContextL* [8], *Javanese* [14], *Subjective-C* [12], *PyContext* [19]) describe the properties of the context as a stack of layers. A layer can roughly be seen as an elementary proposition that drives adaptation and that can be activated or deactivated at runtime. Our context is instead a knowledge base, that offers primitives for easily storing and retrieving contextual data through Datalog queries. Consequently, adaptation is driven on the basis of possibly complex deductions on the knowledge base. We note that a (functional) language linked with a database system does not suffice for implementing adaptation to the context easily and directly, unless equipped with a deductive engine. In the existing implementations, behavioural variations are often implemented as partially defined methods, and are not first-class (except for, e.g., *ContextL* [8]), while ours are and it is well known that this feature improves code modularisation. As a final remark, many COP languages include a `proceed` construct, a sort of `super` invocation in object oriented languages [13], typically used for composing active behavioural variations. This construct is strictly related to the idea of representing the context as a stack of layers, and it is unclear whether it makes sense to introduce a similar construct also in a full-fledged declarative context as ours. Nevertheless, one could consider to implement a construct similar to *call-next-method* [25], in order to run the next case with a satisfied goal, within the active behavioural variation.

Our implementation of ML_{CoDa} took advantage of F#, both for the adaptation component and for the knowledge base. The F# compiler is stable and generates optimised bytecode; it is officially supported by Microsoft and fully integrated inside the .NET environment (and Mono, its open-source counterpart). F# applications can readily run on all platforms (computers or mobile devices) supported by the many libraries and modules of CLR (or Mono).

Our strategy has been to identify in the ML_{CoDa} code the constructs not native in F# through suitable metaprogramming annotations. These annotations drive a JIT compilation of the adaptation constructs by reflecting over the code, while the rest of the code is compiled directly. Therefore the whole compiler of ML_{CoDa} integrates the original F# with the JIT compilation steps. As a result, ML_{CoDa} becomes an ordinary .NET library, usable by any other (F# compatible) application. This prototypical implementation of ML_{CoDa} shows that the .NET type system allows us to solve the impedance mismatch between the functional and the logical components of ML_{CoDa} with a minimal effort.

Our compilation strategy is general and could be followed regardless of the host implementation language, although some implementation details, such as the generation of new identifiers, may require more involved realisations in host languages other than F#. In addition, our implementation of ML_{CoDa}-specific constructs would work independently from the ability to perform JIT compilation in the hosting framework. Consequently, an Ahead-Of-Time (AOT) compilation approach would work as well, so possibly giving feedback to the programmer. Typically, this would require to implement suitable static analyses, e.g. the two-step analysis of [9] that guarantees reliable adaptation of the application to contexts it will be hosted at runtime. Both in JIT and AOT compilation, our approach allows for other different extensions, e.g. to enable code to easily interact with a model checker or other verifiers.

As a final remark, we recall that COP has been proposed by [23] as a basis for implementing the software architecture of autonomic element proposed by [16]. During an execution run, our JIT implementation only compiles those code fragments that the autonomic component needs to adapt, and skips those functions that are not invoked.

6 Conclusions

We presented an extension of F# implementing the COP language ML_{CoDa}, that has a functional component for computing and a logical one for representing and querying the context. Our implementation exploits metaprogramming mechanisms, such as reflection and quotation, to build a JIT compiler. Functions containing ML_{CoDa} adaptation code are marked by the programmer, so driving an automatic translation to pure F# code. Our approach guarantees us a natural interaction between the code using the context and legacy code. This is particularly valuable since ML_{CoDa} code can run on all platforms supported by .NET and can access to all its libraries. Besides the implementation of ML_{CoDa} itself, our JIT compilation schema scales to other host languages, and also to other different extensions.

We plan to extend our work along different lines. First, we will equip our implementation with the two-step static analysis of [9]. Besides the case studies available [5] we will assess our approach on other larger case studies. Also benchmarking and comparing our implementation with others in the literature [6] is of interest.

More on the linguistic aspects, in the current setting the context is only updated by the applications, either by `tell` or `retract`. As experienced on the case study of the e-Healthcare system however, the context can evolve independently of the applications, emitting events to signal the changes — implicitly representing the presence of many different applications sharing the same context. The literature has a great deal of work on this topic, among which [1, 4]. The major extensions to ML_{CoDa} for supporting these aspects include at least the ability of handling the concurrency between the context evolution and the running application, as well as primitives for react and adapt to events.

[5] See `https://github.com/vslab/fscoda`.

References

[1] Tomoyuki Aotani, Tetsuo Kamina & Hidehiko Masuhara (2011): *Featherweight EventCJ: a core calculus for a context-oriented language with event-based per-instance layer transition*. COP '11, ACM, New York, NY, USA, pp. 1:1–1:7, doi:10.1145/2068736.2068737.

[2] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke & Michael Perscheid (2009): *A comparison of context-oriented programming languages*. In: *International Workshop on Context-Oriented Programming*, COP '09, ACM, New York, NY, USA, pp. 6:1–6:6, doi:10.1145/1562112.1562118.

[3] Malte Appeltauer, Robert Hirschfeld & Jens Lincke (2013): *Declarative Layer Composition with The JCop Programming Language*. *Journal of Object Technology* 12(2), pp. 4:1–37, doi:10.5381/jot.2013.12.2.a4.

[4] Engineer Bainomugisha (2012): *Reactive method dispatch for Context-Oriented Programming*. Ph.D. thesis, Comp. Sci. Dept., Vrije Universiteit Brussel.

[5] Luciano Baresi, Elisabetta Di Nitto & Carlo Ghezzi (2006): *Toward Open-World Software: Issue and Challenges*. *Computer* 39(10), pp. 36–43, doi:10.1109/MC.2006.362.

[6] Radu Calinescu, Carlo Ghezzi, Marta Z. Kwiatkowska & Raffaela Mirandola (2012): *Self-adaptive software needs quantitative verification at runtime*. *Commun. ACM* 55(9), pp. 69–77, doi:10.1145/2330667.2330686.

[7] S. Ceri, G. Gottlob & L. Tanca (1989): *What You Always Wanted to Know About Datalog (And Never Dared to Ask)*. *IEEE Trans. on Knowl. & Data Eng.* 1(1), doi:10.1109/69.43410.

[8] Pascal Costanza & Robert Hirschfeld (2005): *Language Constructs for Context-oriented Programming: An Overview of ContextL*. In: *Proceedings of the 2005 Symposium on Dynamic Languages*, DLS '05, ACM, New York, NY, USA, pp. 1–10, doi:10.1145/1146841.1146842.

[9] Pierpaolo Degano, Gian-Luigi Ferrari & Letterio Galletta (2014): *A Two-Phase Static Analysis for Reliable Adaptation*. In Dimitra Giannakopoulou & Grenoble Gwen Salaün, editors: *12th International Conference on Software Engineering and Formal Methods, SEFM 2014*, *Lecture Notes in Computer Science* 8702, Springer, pp. 347–362, doi:10.1007/978-3-319-10431-7_28.

[10] Pierpaolo Degano, Gian-Luigi Ferrari & Letterio Galletta (2014): *A Two-Component Language for COP*. In: *Proceedings of 6th International Workshop on Context-Oriented Programming*, COP'14, ACM, New York, NY, USA, pp. 6:1–6:7, doi:10.1145/2637066.2637072.

[11] Brecht Desmet, Jorge Vallejos, Pascal Costanza, Wolfgang De Meuter & Theo D'Hondt (2007): *Context-Oriented Domain Analysis*. In Boicho N. Kokinov, Daniel C. Richardson, Thomas Roth-Berghofer & Laure Vieu, editors: *Modeling and Using Context*, *Lecture Notes in Computer Science* 4635, Springer Berlin Heidelberg, pp. 178–191, doi:10.1007/978-3-540-74255-5_14.

[12] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht & Julien Goffaux (2011): *Subjective-C*. In Brian Malloy, Steffen Staab & Mark van den Brand, editors: *Software Language Engineering*, *Lecture Notes in Computer Science* 6563, Springer Berlin Heidelberg, pp. 246–265, doi:10.1007/978-3-642-19440-5_15.

[13] Robert Hirschfeld, Pascal Costanza & Oscar Nierstrasz (2008): *Context-oriented Programming*. *Journal of Object Technology* 7(3), pp. 125–151, doi:10.5381/jot.2008.7.3.a4.

[14] Tetsuo Kamina, Tomoyuki Aotani & Hidehiko Masuhara (2013): *A Unified Context Activation Mechanism*. In: *Proceedings of the 5th International Workshop on Context-Oriented Programming*, COP'13, ACM, New York, NY, USA, pp. 2:1–2:6, doi:10.1145/2489793.2489795.

[15] Tetsuo Kamina, Tomoyuki Aotani, Hidehiko Masuhara & Tetsuo Tamai (2014): *Context-oriented Software Engineering: A Modularity Vision*. MODULARITY '14, ACM, New York, NY, USA, pp. 85–98, doi:10.1145/2577080.2579816.

[16] Jeffrey O. Kephart & David M. Chess (2003): *The Vision of Autonomic Computing*. *IEEE Computer* 36(1), pp. 41–50, doi:10.1109/MC.2003.1160055.

[17] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm & William G. Griswold (2001): *An Overview of AspectJ*. In Jørgen Lindskov Knudsen, editor: *ECOOP 2001 — Object-Oriented Programming*, *Lecture Notes in Computer Science* 2072, Springer Berlin, pp. 327–354, doi:10.1007/3-540-45337-7_18.

[18] Seng W. Loke (2004): *Representing and Reasoning with Situations for Context-aware Pervasive Computing: a Logic Programming Perspective*. *Knowl. Eng. Rev.* 19(3), pp. 213–233, doi:10.1017/S0269888905000263.

[19] Martin von Löwis, Marcus Denker & Oscar Nierstrasz (2007): *Context-oriented Programming: Beyond Layers*. In: *Proceedings of the 2007 International Conference on Dynamic Languages: In Conjunction with the 15th International Smalltalk Joint Conference 2007*, ICDL '07, ACM, New York, NY, USA, pp. 143–156, doi:10.1145/1352678.1352688.

[20] Giorgio Orsi & Letizia Tanca (2011): *Context Modelling and Context-Aware Querying*. In O. Moor, G. Gottlob, T. Furche & A. Sellers, editors: *Datalog Reloaded*, *LNCS* 6702, Springer, pp. 225–244, doi:10.1007/978-3-642-24206-9_13.

[21] Andres J. Ramirez & Betty H. C. Cheng (2010): *Design Patterns for Developing Dynamically Adaptive Systems*. In: *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, ACM, New York, NY, USA, pp. 49–58, doi:10.1145/1808984.1808990.

[22] Mazeiar Salehie & Ladan Tahvildari (2009): *Self-adaptive software: Landscape and research challenges*. *ACM Trans. Auton. Adapt. Syst.* 4(2), pp. 14:1–14:42, doi:10.1145/1516533.1516538.

[23] Guido Salvaneschi, Carlo Ghezzi & Matteo Pradella (2011): *Context-Oriented Programming: A Programming Paradigm for Autonomic Systems*. *CoRR* abs/1105.0069. Available at `http://arxiv.org/abs/1105.0069`.

[24] Guido Salvaneschi, Carlo Ghezzi & Matteo Pradella (2013): *An Analysis of Language-Level Support for Self-Adaptive Software*. *ACM Trans. Auton. Adapt. Syst.* 8(2), pp. 7:1–7:29, doi:10.1145/2491465.2491466.

[25] Jorge Vallejos, Sebastián González, Pascal Costanza, Wolfgang De Meuter, Theo D'Hondt & Kim Mens (2010): *Predicated Generic Functions*. In Benoît Baudry & Eric Wohlstadter, editors: *Software Composition*, *Lecture Notes in Computer Science* 6144, Springer Berlin Heidelberg, pp. 66–81, doi:10.1007/978-3-642-14046-4_5.

[26] Pamela Zave & Michael Jackson (1997): *Four Dark Corners of Requirements Engineering*. *ACM Trans. Softw. Eng. Methodol.* 6(1), pp. 1–30, doi:10.1145/237432.237434.