Typing Context-Dependent Behavioural Variations

Pierpaolo Degano degano@di.unipi.it
Gian-Luigi Ferrari giangi@di.unipi.it
Letterio Galletta galletta@di.unipi.it
Gianluca Mezzetti mezzetti@di.unipi.it
Dipartimento di Informatica, Università di Pisa, Pisa, Italy

Context Oriented Programming (COP) is a hot topic nowadays. A number of programming languages endowed with COP features has been developed. However, some foundational issues remain unclear. This paper proposes adopting static analysis techniques to predict how programs may react in different execution environments. We introduce a core functional language, ContextML, equipped with COP primitives for manipulating contexts and for programming behavioural variations. In particular, we specify the dispatching mechanism, used to select the program fragments to be executed in the current active context. Besides the dynamic semantics we present an annotated type system. It guarantees that the well-typed programs adapt to any context, i.e. the dispatching mechanism always succeeds at runtime.

1 Introduction

Computers increasingly pervade our everyday life, pushed by the big steps forward of hardware systems and by the increase of digital information.

On the one side, computer devices surround people in different shapes and sizes in a highly distributed manner. Devices are often interconnected, can interact and share resources. Programs are resources themselves as invokable remote services or downloadable code.

On the other side, processing the great quantity of information generated and consumed by devices brings to an approach, where most of the computation and of the storage is demanded to specific, powerful remote entity. This is the case of Cloud or Grid systems, which often are made by heterogeneous nodes, possibly distributed in a wide-area.

This new setting puts software system into a highly-dynamic environment where services, resources and hardware components appear, mutate and disappear. This calls for a a great shift of programming paradigm. In particular, there is a growing interest in the design and development of applications that are aware of their working environment and are *adaptive*, i.e. capable to adapt to different situations.

Context-Oriented Programming (COP)[7] is a programming paradigm recently proposed to tackle such an issue of building adaptive systems. COP makes available language primitives to express context-dependent behaviour (namely *behavioural variation*) in a modular fashion.

Behavioural variations are chunks of behaviour that modify the execution of a computational system depending on the current working environment. Layers are the linguistic mechanism that enable the programmer to group variations. Layers can be activated in arbitrary places of the code with appropriate primitives. Layer activation has its own scope, activated layers are piled up into a stack that is called *context*. Therefore, the actual behaviour of a COP program is carried out by a *dispatching* procedure that selects the program fragments to be executed depending on the context contents.

Most of the COP research efforts are focused on the concrete implementation details and only few papers [6, 4] investigated basic foundational issues. We briefly discuss them in Section 3.

Our work aims at contributing to the foundations of COP programming languages. We introduce a core calculus (ContextML) with a precise semantic description of the adaptivity constructs.

Submitted to:
PLACES 2012

To illustrate the novel features of ContextML we resort to a running example. We consider a program embedded in a mobile device. Its behaviour depends on the active profile of its battery. We assume that the battery level has two profiles, the power saving mode and the performance one. These profiles are represented at code level as two different layers: `PowerSavingMode` and `PerformanceMode`. The function `getBatteryProfile`, described below, queries the sensor (`batSensor`) and returns the layer describing the current active profile depending on a `threshold` value:

$$\lambda_{\texttt{getBatteryProfile}}() \Rightarrow \textbf{if } (\texttt{batSensor}() > \texttt{threshold}) \textbf{ then } \texttt{PerformanceMode} \textbf{ else } \texttt{PowerSavingMode}$$

Layers are expressible values, hence they can be produced as results of function calls. The construct $\textbf{with}(e_1)$ **in** e_2 activates the layer obtained evaluating e_1 and delimits its activation scope to the inner expression e_2. For instance, in the code below, the layer obtained as result of the call `getBatteryProfile`() is active throughout the execution of the inner expression.

$$\begin{array}{l}\textbf{with}(\texttt{getBatteryProfile}())\ \textbf{in}\\ \quad \texttt{PowerSavingMode.doSomeThing}(),\\ \quad \texttt{PerformanceMode.doSomeThingElse}()\end{array}$$

In the inner expression, the *layered expression* is an expression defined by cases that specifies the context-dependent behavioural variation considered.

Note that if the programmer neglects a case for the profile of the battery, e.g. `OnDemandMode`, then the program throws a runtime error being unable to adapt to the context. We propose to tackle these undesired behaviour by adopting static analysis techniques. To this aim we extend the ML type system in order to guarantee that well-typed programs are always capable to react to their changing environment, i.e. the dispatching procedure always succeeds at runtime.

2 ContextML: a context-oriented ML core

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 variation of expressions, depending on the context in which the program is evaluated. The structural operational semantics and the type system of ContextML follow.

Dynamic Semantics Let $\mathbb{N}$ be the set of naturals, Ide a set of identifiers, LayerNames a set of layer names, then the syntax of ContextML is defined by the following grammar:

$$\begin{array}{rl} & n \in \mathbb{N} \qquad x, f \in \mathsf{Ide} \qquad L \in \mathsf{LayerNames}\\ v, v_1, v' ::= & n \mid L \mid \lambda_f\, x \Rightarrow e\\ e, e_1, e' ::= & v \mid x \mid e_1 e_2 \mid \textbf{let}\, x = e_1\, \textbf{in}\, e_2 \mid e_1\, \textbf{op}\, e_2 \mid \textbf{if}\, e_0\, \textbf{then}\, e_1\, \textbf{else}\, e_2 \mid \textbf{with}(e_1)\, \textbf{in}\, e_2 \mid lexp\\ lexp ::= & L.e \mid L.e, lexp \end{array}$$

The novelties with respect to ML are layers as expressible values; the **with** construct for activating 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.

$$\text{with}_1\ \frac{L :: C \vdash e \to e'}{C \vdash \textbf{with}(L)\ \textbf{in}\ e \to \textbf{with}(L)\ \textbf{in}\ e'} \qquad \text{with}_2\ \frac{}{C \vdash \textbf{with}(L)\ \textbf{in}\ v \to v}$$

$$\text{with}_3\ \frac{C \vdash e \to e'}{C \vdash \textbf{with}(e)\ \textbf{in}\ e_1 \to \textbf{with}(e')\ \textbf{in}\ e_1} \qquad \text{lexp}\ \frac{\exists k.k = \min\{j \mid \exists v.L'_j = L_v\} \wedge L'_k = L_i}{[L'_1,\ldots,L'_m] \vdash L_1.e_1,\ldots,L_n.e_n \to e_i}$$

Figure 1: ContextML semantics.

The semantics is only defined for closed expression and is characterised by judgements having the form $C \vdash e \to e'$ meaning that in context C the closed expression e reduces to e' in one evaluation step. In Figure 1 we only show the semantic rules (with_1),(with_2),(with_3),(lexp) that deal with the new constructs, the others are inherited from the standard ML. We briefly comment on the new ones only. Rules for $\textbf{with}(e_1)\ \textbf{in}\ e_2$ evaluate e_2 in the context extended by the layer obtained evaluating e_1. When a layered expression $L_1.e_1,\ldots,L_n.e_n$ has to be evaluated (rule *lexp*), the context of evaluation is inspected top-down (dispatch mechanism). When a layer in the context matches one of the L_i, the corresponding expression e_i is evaluated. If no layer matches then the computation gets stuck.

Type system We introduce a monomorphic type system for ContextML which ensures that the dispatch mechanism always succeeds at run-time for well-typed expressions.

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

Types are integers, layers and functions.

$$\tau, \tau_1, \tau' ::= \mathit{int} \mid \mathit{ly}_\phi \mid \tau_1 \xrightarrow{\psi} \tau_2 \qquad \phi, \psi \in \wp(\mathsf{LayerNames})$$

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

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

$$\texttt{unit} \xrightarrow{\emptyset} \mathit{ly}_{\{\texttt{PowerSavingMode},\texttt{PerformanceMode}\}}.$$

The intuition is that the function returns 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. Since types are annotated, the type system contains rules dealing with the subtyping. The rules (Sint), (Sly), (Sfun) have judgements of the form $\tau_1 \leq \tau_2$ (τ_1 is a subtype of τ_2). Furthermore, we assume that annotations are ordered by set-inclusion and that $|C|$ is the set of active layers in a context C.

By rule (Sly) a layer type ly_ϕ is a subtype $\mathit{ly}_{\phi'}$ if and only if the annotation ϕ is a subset of ϕ'. Rule (Sfun) is 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.

Rule (Tly) asserts that the type of a layer L is ly annotated with the singleton set $\{L\}$. In rule (Tfun) we guess a type for the bound variable, for the function f and 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 that of the resulting function. Additionally we require that the resulting type is annotated with a precondition that includes the layers in C'.

$$\text{(Sint)}\ \frac{}{int \leq int} \qquad \text{(Sly)}\ \frac{\phi \subseteq \phi'}{ly_\phi \leq ly_{\phi'}}$$

$$\text{(Sfun)}\ \frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2' \qquad \psi \subseteq \psi'}{\tau_1 \xrightarrow{\psi} \tau_2 \leq \tau_1' \xrightarrow{\psi'} \tau_2'}$$

$$\text{(Tint)}\ \frac{}{\langle \Gamma; C\rangle \vdash n : int} \qquad \text{(Tly)}\ \frac{}{\langle \Gamma; C\rangle \vdash L : ly\{L\}}$$

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

$$\text{(Tfun)}\ \frac{\langle \Gamma, x : \tau_1, f : \tau_1 \xrightarrow{|C'|} \tau_2; C'\rangle \vdash e : \tau_2}{\langle \Gamma; C\rangle \vdash \lambda_f\, x \Rightarrow e : \tau_1 \xrightarrow{|C'|} \tau_2}$$

$$\text{(Top)}\ \frac{\langle \Gamma; C\rangle \vdash e_1 : int \qquad \langle \Gamma; C\rangle \vdash e_2 : int}{\langle \Gamma; C\rangle \vdash e_1\ \mathbf{op}\ e_2 : int}$$

$$\text{(Tlet)}\ \frac{\langle \Gamma; C\rangle \vdash e_1 : \tau_1 \qquad \langle \Gamma, x : \tau_1, C\rangle \vdash e_2 : \tau_2}{\langle \Gamma; C\rangle \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

$$\text{(Tif)}\ \frac{\langle \Gamma; C\rangle \vdash e_0 : int \qquad \langle \Gamma; C\rangle \vdash e_1 : \tau \qquad \langle \Gamma; C\rangle \vdash e_2 : \tau}{\langle \Gamma; C\rangle \vdash \mathbf{if}\ \ e_0\ \ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau}$$

$$\text{(Twith)}\ \frac{\langle \Gamma; C\rangle \vdash : e_1 : ly_\phi \qquad \forall L' \in \phi. \langle \Gamma; L' :: C\rangle \vdash e_2 : \tau}{\langle \Gamma; C\rangle \vdash \mathbf{with}(e_1)\ \mathbf{in}\ e_2 : \tau}$$

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

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

Figure 2: ContextML type system

Rule (Twith) establishes that an expression **with** has type τ, provided the type for e_1 is ly_ϕ (recall that ϕ is a set of layers) and e_2 has type τ in the context C extended by the layers in ϕ. By (Tlexp) 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 run-time. Notably, when evaluating a layered expression one of the mentioned layers will be active in the current context.

Back to our example, the expression provided is well-typed as witnessed in Figure 3. The type of `getBatteryProfile` ensures that one of the two layers `PowerSavingMode` or `PerformanceMode` is returned. One among them is required to be active so to evaluate the layered expression. Hence, the (Twith) rule can guarantee that the whole expression is never stuck at runtime.

Rule (Tapp) is almost standard and reveals the mechanism of function precondition. The application gets a type if only if the layers in the precondition ϕ are active in the current context C. To better explain how preconditions work, consider the example in Figure 4. There the function $\lambda_f\, x \Rightarrow L_1.0$ is shown having type $int \xrightarrow{\{L_1\}} int$. This means that L_1 must be active in the context of activation of the function.

The remaining rules are standard and we do not comment on them for brevity.

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:

$$\dfrac{\langle\Gamma;C\rangle \vdash : \texttt{getBatteryProfile}() : ly_{\phi} \quad \langle\Gamma;C'\rangle \vdash \begin{array}{c}\texttt{PowerSavingMode.doSomeThing}(),\\ \texttt{PerformanceMode.doSomeThingElse}()\end{array} : \tau}{\langle\Gamma;C\rangle \vdash \textbf{with}(\texttt{getBatteryProfile}())\ \textbf{in}\ \begin{array}{c}\texttt{PowerSavingMode.doSomeThing}(),\\ \texttt{PerformanceMode.doSomeThingElse}()\end{array} : \tau}\ (\text{Tlexp})$$

$$\dfrac{\dfrac{\langle\Gamma;C'\rangle \vdash \texttt{doSomeThing}() : \tau \quad \langle\Gamma;C'\rangle \vdash \texttt{doSomeThingElse}() : \tau}{\texttt{PowerSavingMode} \in |C'| \vee \texttt{PerformanceMode} \in |C'|}\ (\text{Tlexp}) \qquad \dfrac{\ldots}{\langle\Gamma;C''\rangle \vdash \ldots}}{\langle\Gamma;C\rangle \vdash : \texttt{getBatteryProfile}() : ly_{\phi} \quad \langle\Gamma;C'\rangle \vdash \begin{array}{c}\texttt{PowerSavingMode.doSomeThing}(),\\ \texttt{PerformanceMode.doSomeThingElse}()\end{array} : \tau}\ (\text{Twith})$$

Figure 3: The typing derivation of the running example. We assume $\texttt{doSomeThing}, \texttt{doSomeThingElse} : \texttt{unit} \to \tau$. We denote $\phi = \{\texttt{PowerSavingMode}, \texttt{PerformanceMode}\}$;$C' = \texttt{PowerSavingMode} :: C$;$C'' = \texttt{PerformanceMode} :: C$. The last rule used is (Twith).

$$\dfrac{\dfrac{\dfrac{\langle\Gamma, x:\tau, f:\tau\to\tau; C'\rangle \vdash 0 : \tau \quad L_1 \in C'}{\langle\Gamma, x:\tau, f:\tau \xrightarrow{|C'|} \tau; C'\rangle \vdash L_1.0 : \tau}}{\langle\Gamma;C\rangle \vdash \lambda_f\, x \Rightarrow L_1.0 : \tau \xrightarrow{|C'|} \tau} \qquad \dfrac{\langle\Gamma, g:\tau \xrightarrow{|C'|} \tau; C\rangle \vdash g : \tau\to\tau \quad \dfrac{\langle\Gamma, g:\tau \xrightarrow{|C'|} \tau; C\rangle \vdash 3 : \tau \quad |C'| \subseteq |C|}{\langle\Gamma, g:\tau \xrightarrow{|C'|} \tau; C\rangle \vdash g\,3 : \tau}}{\langle\Gamma, g:\tau \xrightarrow{|C'|} \tau; C\rangle \vdash g\,3 : \tau}}{\langle\Gamma;C\rangle \vdash \textbf{let}\ g = \lambda_f\, x \Rightarrow L_1.0\ \textbf{in}\ g\,3 : \tau}$$

Figure 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 $\tau = int$.

Lemma 2.1 (Progress)**.** *Let e be a closed expression such that for some C $\langle\Gamma;C\rangle \vdash e : \tau$. Then either e is a value or $C \vdash e \to e'$.*

Lemma 2.2 (Subject reduction)**.** *Let e be a closed expression, if $\langle\Gamma;C\rangle \vdash e : \tau$ and $C \vdash e \to e'$ then $\langle\Gamma, C\rangle \vdash e' : \tau$*

3 Discussion

Related work Several COP programming languages have been proposed (see e.g.*ContextL* [5] and *ContextJ* [1]). Usually COP features are introduced within the object oriented paradigm so providing behavioural variations at object level.

Most of the research efforts have mainly tackled implementation issues. To the best of our knowledge only few papers provide a precise semantic description.

In [6] an extension of *Featherweight Java* [9] has been proposed. This calculus includes *layers (de)activation*, but layers are not expressible values. Furthermore, a static type system ensures that there exists a binding for each dispatched method call. This fact is based on the strong assumption that layers do not introduce new methods but only refine existent ones. Our type system relaxes this assumption.

Our approach is much similar to the one of Clarke et al. [4] and the main difference is that we consider a functional language while [4] considers Featherweight Java object oriented language.

Conclusions and further work We started investigating the foundational issues of the COP paradigm with a calculus endowing COP primitives. We have defined a dynamic semantics that formalises the operational mechanisms behind these constructs. A distinguished element of our semantics is the dispatching procedure, that selects the behavioural variation depending on the active context. We have also specified a type system guaranteeing that the dispatching mechanism always succeeds at runtime for well-typed expressions.

In our current proposals, activation of layers is driven according to the flow of program execution. A more general approach would instead consider the asynchronous evolution of the environment. We plan

to investigate this issue and to formalise event-driven changes of context.

We also intend to refine the type system by introducing effects to represent an over-approximation of the evolution pattern of context shape. In doing that, techniques similar to session-types [8] might suggest us useful mechanisms. Types and effects will enhance our static analysis of programs, following the lines of [2, 3]. In particular we would like to accept or reject programs at compile time, also depending on non-functional requirements on the context evolution, e.g. when security policies are to be enforced upon context usages.

Acknowledgement The authors would like to thank the anonymous referees for their comments that pointed us an inaccuracy, and guided us to improve the quality of our paper.

This work has been partially supported by IST-FP7-FET open-IP project ASCENS and Regione Autonoma Sardegna, L.R. 7/2007, project TESLA.

References

[1] M. Appeltauer, R. Hirschfeld, M. Haupt & H. Masuhara (2011): *ContextJ: Context-oriented Programming with Java*. *Computer Software* 28(1).

[2] Massimo Bartoletti, Pierpaolo Degano & Gian-Luigi Ferrari (2009): *Planning and verifying service composition*. *Journal of Computer Security* 17(5), pp. 799–837. Available at `http://dx.doi.org/10.3233/JCS-2009-0357`.

[3] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari & Roberto Zunino (2009): *Local policies for resource usage analysis*. *ACM Trans. Program. Lang. Syst.* 31(6). Available at `http://doi.acm.org/10.1145/1552309.1552313`.

[4] Dave Clarke & Ilya Sergey (2009): *A semantics for context-oriented programming with layers*. In: *International Workshop on Context-Oriented Programming*, COP '09, ACM, New York, NY, USA, pp. 10:1–10:6, doi:http://doi.acm.org/10.1145/1562112.1562122. Available at `http://doi.acm.org/10.1145/1562112.1562122`.

[5] Pascal Costanza (2005): *Language constructs for context-oriented programming*. In: *In Proceedings of the Dynamic Languages Symposium*, ACM Press, pp. 1–10, doi:http://doi.acm.org/10.1145/1146841.1146842.

[6] R. Hirschfeld, A. Igarashi & H. Masuhara (2011): *ContextFJ: a minimal core calculus for context-oriented programming*. In: *Proceedings of the 10th international workshop on Foundations of aspect-oriented languages*, ACM, pp. 19–23, doi:http://doi.acm.org/10.1145/1960510.1960515.

[7] Robert Hirschfeld, Pascal Costanza & Oscar Nierstrasz (2008): *Context-Oriented Programming*. *Journal of Object Technology, March-April 2008, ETH Zurich* 7(3), pp. 125–151.

[8] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, Springer-Verlag, London, UK, pp. 122–138. Available at `http://dl.acm.org/citation.cfm?id=645392.651876`.

[9] Atsushi Igarashi, Benjamin C. Pierce & Philip Wadler (2001): *Featherweight Java: a minimal core calculus for Java and GJ*. *ACM Trans. Program. Lang. Syst.* 23, pp. 396–450, doi:http://doi.acm.org/10.1145/503502.503505. Available at `http://doi.acm.org/10.1145/503502.503505`.