Unifying Multiple Layer Activation Mechanisms Using One Event Sequence

Tomoyuki Aotani  
Tokyo Institute of Technology  
aotani@is.titech.ac.jp

Tetsuo Kamina  
Ritsumeikan University  
kamina@acm.org

Hidehiko Masuhara  
Tokyo Institute of Technology  
masuhara@acm.org

ABSTRACT

Different context-oriented programming languages try to capture contexts with respect to different things, including a computation, an object, and a device that executes a program, by providing different layer activation mechanisms. When we want to exploit all of those different kinds of contexts at the same time, it is not clear how the effects of those contexts should be combined.

We develop LamFJ, a calculus for expressing various layer activation mechanisms. It replaces the with and without expressions in ContextFJ with four expressions that fire context change events, which models changes of each context. LamFJ is not only powerful enough to express multiple layer activation mechanisms but also clearly defines combined effects of those mechanisms. In addition to the supported layer activation mechanisms in the paper, namely imperative activation, per-object activation and dynamic scoping, we aim at supporting other mechanisms like reactive and structural activation with small extensions.

Categories and Subject Descriptors
D.3.1 [Programming Languages]: Formal Definitions and Theory; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms
Languages

Keywords
Context-oriented programming, layer activation mechanisms

1. INTRODUCTION

Context-oriented programming (COP) [5] is an approach to modularize behavioral variations of a program from the viewpoint of the context that changes during execution of a program.

Contexts are defined with respect to a variety of things such as a computation, an object and a device that executes the program. To capture contexts with respect to various kinds of things such as a computation, an object and a device that executes the program, several language mechanisms called layer activation mechanisms have been proposed. Dynamic scoping available in ContextJ [11] and similar languages [1,10,14] try to capture contexts with respect to a computation, i.e., from which method the computation is executed. Per-object activation available in EventCJ [7] and per-agent activation available in ContextErlang [13] try to capture contexts with respect to each object and agent, respectively, e.g., the tab represented by an object is selected or not. Implicit activation available in PyContext [14] and reactive activation available in Flute [2] try to capture contexts with respect to the device, e.g., whether the device is indoors or outdoors. Imperative activation available in Subjective-C [4] tries to capture contexts with respect to the program itself, e.g., in which phase the program execution enters.

It is difficult to capture all kinds of contexts using only one layer activation mechanism [9,12]. Therefore, to capture contexts with respect to multiple things, it is necessary to use multiple layer activation mechanisms. This situation is not rare. For example, in Android applications, we often need to capture contexts with respect to the status of the device and a computation. Using imperative activation is preferable for capturing contexts with respect to the status of the device because it affects the overall computations of the program. On the other hand, using dynamic scoping is preferable for capturing contexts with respect to a computation because we need to delimit the effects of the contexts.

The aim of the study is to provide a simple guideline and foundation for developing clear semantics of the COP languages that allow programmers to use multiple layer activation mechanisms at the same time in a program. As a first step of the work, we propose LamFJ, a calculus for expressing various layer activation mechanisms in the paper. It replaces the with and without expressions in ContextFJ [6] with four expressions that fire context change events, which model changes of each context. LamFJ is not only powerful enough to express multiple layer activation mechanisms but also clearly defines effects of those mechanisms.

The rest of the paper is organized as follows. Section 2 shows an example in which it is necessary to use multiple layer activation mechanisms at the same time and provides our claim on how the effects of the mechanisms should be combined. Section 3 formalizes our claim. Later on, we
review related work in Section 4 and conclude the paper in Section 5 along with showing directions of our future work.

2. MOTIVATION

2.1 Example: an Android application

Initialization and battery-aware computing, which changes behavior depending on the status of the battery, are examples of context-dependent behavior in Android applications. An activity is one of the important components in android applications. It provides a screen that shows the user interface and allow users to interact and is presented by an object of class Activity. Figure 1 shows a simplified life cycle of activities. The boxed nodes show the method name that are called by the framework. The oval nodes explain what happens at each phase. Suppose an application is launched. The activity that provides the main screen is then launched, and the application framework sends onStart to the activity to initialize data for the screen. Later on, suppose the device goes to sleep. Then the application framework sends onStop to the activity. When the device wakes up and the application gets focused, onRestart is called to the activity to initialize and restore data.

Initialization and restoring are often similar tasks, and it is good practice to use COP to modularize them. We show in Figure 2 the code skeleton that uses the with-block within onRestart to capture the context with respect to the initialization task.

To achieve battery-aware computing, we need to monitor changes on the status of the battery. In Android applications, BatteryManager broadcasts all battery and charging details when they are changed. To capture the information, we develop and register event handlers for particular events on the battery. Figure 3 shows how we can capture the context with respect to power connection. In lines 1–6, we show a configuration file that registers PowerConnRecver as the event handler for the events ACTION_POWER_CONNECTED and ACTION_POWER_DISCONNECTED, which are fired when the device is plugged in and unplugged, respectively. PowerConnRecver (lines 8–14) declares onReceive, which receives the battery details as an object of class Intent and activates and deactivates Plugged layer using imperative activation (i.e., activate and deactivate operations) if power is connected and disconnected, respectively.

2.2 Issue and our claim

It is not clear how the effects of multiple layer activation mechanisms should be combined. Figure 4 is an example that uses imperative activation (the deactivate and activate operations) and dynamic scoping (the with and without blocks). Unlike with the Android application example shown in the previous section, one layer is activated and deactivated using the two layer activation mechanisms. Does
layer \( L \) affect invocation \( m_1() \) at line 3? What about invocations \( m_2() \) at line 6, \( m_3() \) at line 8 and \( m_4() \) at line 10?

To answer the question, we claim that the effect of context change event \( \epsilon \), which is either activation of some layer, deactivation of some layer or cancel of some context change event, precedes the effect of context change event \( \epsilon' \) if (1) \( \epsilon' \) is cancel or is canceled, or (2) \( \epsilon \) is not cancel, is not canceled and is fired more recently than \( \epsilon' \). A context change event \( \epsilon \) is canceled if cancel of \( \epsilon \) is fired. An activation event of layer \( L \) is fired when \( L \) is activated using imperative activation (i.e., \text{activate}(L)) or the execution enters the \text{with} block that specifies \( L \). Similarly, a deactivation event of layer \( L \) is fired when \( L \) is deactivated using imperative activation (i.e., \text{deactivate}(L)) or the execution enters the \text{without} block that specifies \( L \). Cancel of \( \epsilon \) is fired when the execution escapes from the \text{with} or \text{without} block that fires \( \epsilon \).

The claim should not be strange. If we ignore cancel, it is actually analogous to the principle in COP languages that layer \( L \) precedes layer \( L' \) if \( L \) is activated more recently than \( L' \).

If we assume that methods \( m_1, m_2, m_3 \) and \( m_4 \) invoked in Figure 4 do not fire any context change events, context change events are fired and methods are invoked in the following order if we execute the program:

\[ \epsilon_1 < \epsilon_2 < m_1() < \epsilon_3 < m_2() < \epsilon_4^{-1} < m_3() < \epsilon_5^{-1} < m_4() \]

\( m_i() \) denotes invoking method \( m_i() \). \( \epsilon_i \) denotes the following context change events:

- \( \epsilon_1 \): entering the \text{with} block at line 1
- \( \epsilon_2 \): deactivating layer \( L \) at line 2
- \( \epsilon_3 \): activating layer \( L \) at line 4
- \( \epsilon_4 \): entering the \text{without} block at line 5

Now we can answer the questions. Layer \( L \) does not affect invocation \( m_1() \) because \( \epsilon_2 \) is the most preceding context change event, which deactivates \( L \). Similarly, it does not affect invocation \( m_2() \) because \( \epsilon_4 \) is the most preceding context change event, which deactivates \( L \). It is a little bit tricky to answer the remaining two questions about invoking \( m_3() \) and \( m_4() \). The most recent event for \( m_3() \) is \( \epsilon_4^{-1} \), but it cannot be the most preceding context change event because it is cancel. \( \epsilon_4 \), which is the next most recent event for \( m_3() \), also cannot be the most preceding event because it is canceled. Thinking in this way, we find that the most preceding context change event for \( m_3() \) is \( \epsilon_3 \). Because \( \epsilon_3 \) activates \( L \), \( L \) affects \( m_3() \). \( \epsilon_3 \) is also the most preceding context change event for \( m_4() \). Therefore, \( L \) also affects \( m_4() \).

3. LamFJ

As a formalization of our claim shown in Section 2, we develop LamFJ, a calculus for expressing various layer activation mechanisms and combining their effects. It removes \text{with} and \text{without} blocks from and adds context change events and \text{event firing expressions} to ContextFJ. Context change events model activation and deactivation of layers. An event firing expression specifies a context change event to be fired.

LamFJ supports three layer activation mechanisms, namely imperative activation, per-object activation and dynamic scoping, and should be considered as a common intermediate language for COP languages that support those mechanisms.

Operators and blocks to activate and deactivate layers are supposed to be expressed using event firing expressions.

LamFJ does not allow layers to add new methods which are not declared in classes as ContextFJ. This does not only make the type system straightforward but also make it trivial to prove type soundness. Therefore, we omit its type system and proofs of type soundness.

3.1 Syntax

Let metavariables \( C, D \) and \( E \) range over class names; \( f \) ranges over field names; \( L \) over layer names; \( n \) ranges over method names; and \( v, w \) and \( x \) range over addresses, which include the null address denoted as \( :: x \) and \( y \) range over variables, which include a special variable \( \text{this} \). The abstract syntax of LamFJ is as follows:

\[
\begin{align*}
\text{CL} &::= \text{class } C \subset C_1 \ F_1 F_2 K \ R \\
K &::= C(x, y, \gamma)\{\text{super}(\xi); \text{this.} \gamma;\} \\
M &::= C(m(x))\{\text{return } e;\} \\
e &::= v | x | \text{new } C(\xi) | e.f | e.m(\xi) | \text{proceed}(\xi) | e;e | \text{super}(\xi) | v < C, L, D, \xi = m(\xi) \ |
\epsilon \ |
\epsilon L \uparrow e | \epsilon L \downarrow e | \epsilon | e L \downarrow e \\
\epsilon &::= \alpha \ |
\delta \ |
\sigma \ |
\pi \ |
\sigma^{-1} \ |
\pi^{-1} 
\end{align*}
\]

Overlines denote sequences, e.g., \( \overline{\tau} \) stands for a possibly empty sequence \( f_1, \ldots, f_n \) and similarly for \( \overline{C}, \overline{x}, \overline{\alpha}, \) and so on. The empty sequence is denoted by \( \epsilon \). We also abbreviate a sequence of pairs, writing \( \text{CL} \overline{F} \) for “\( C_1 \overline{f}_1, \ldots, C_n \overline{f}_n \),” where \( n \) is the length of \( C \) and \( \overline{F} \), and similarly “\( C \overline{F} \)” as shorthand for the sequence of declarations “\( C_1 f_1, \ldots, C_n f_n \)” and “\( \text{this.} \overline{F} \)” for “\( \text{this.} f_1 = f_1, \ldots, \text{this.} f_n = f_n \).” We use commas and semicolons for concatenations. Sequences of field declarations, parameter names, layer names, and method declarations are assumed to contain no duplicate names.

A class definition \( \text{CL} \) consists of its name, its superclass name, field declarations \( \overline{C} \overline{F} \), a constructor \( K \), and method definitions \( R \). A constructor \( K \) is a trivial one that takes initial values of all fields and sets them to the corresponding fields. Unlike the examples in the last section, we do not provide syntax for layers; partial methods are registered in a partial method table, which is explained below. A method \( M \) takes \( x \) as arguments and returns the value of expression \( e \). The method body consists of a single return statement and all constructs return values. An expression \( e \) can be an address, variable, object creation, field access, method call, sequential composition of two expressions, \text{super} call, \text{proceed} call, a special form of method call \( v < C, L, D, \xi = m(\xi) \), or one of the four forms of event firing expressions that fires a context change event \( \epsilon \) on layer \( L \), denoted as \( \epsilon L \). A value is an address \( v \). A context change event is either an activation denoted as \( \alpha \), deactivation denoted as \( \delta \), activation counter increment denoted as \( \sigma \), activation counter decrement denoted as \( \sigma^{-1} \), deactivation counter increment denoted as \( \pi \), or deactivation counter decrement denoted as \( \pi^{-1} \).

Instead of using cancel events to handle the semantics of \text{with} and \text{without} blocks as shown in Section 2, LamFJ uses activation counters \( \sigma \) and \( \pi \) are supposed to be fired when entering \text{with} and \text{without} blocks, respectively. \( \sigma^{-1} \) and \( \pi^{-1} \) are supposed to be fired when escaping from \text{with} and \text{without} blocks, respectively.

We have four forms of event firing expressions. \( \epsilon L \uparrow e \) is supposed to be used to realize the two operators acti-
vate and deactivate in imperative activation. It fires globally
the context change event \( \epsilon \) on layer \( L \). For example,
activate\( (L) \) and deactivate\( (L) \) followed by an expression
\( e \) is expressed as \( \alpha_L \uparrow \varepsilon \) and \( \delta_L \uparrow \varepsilon \), respectively, in LamFJ.
\( \epsilon \uparrow e \uparrow e' \) is similar. It runs \( e' \) after firing \( \epsilon \) to an object
\( e \), which realizes per-object/agent activation available in
EventCJ and ContextErlang. \( e' \downarrow \epsilon_L \uparrow e \) is also similar to
\( \epsilon_L \uparrow e \), but the subexpression \( e \) before the event is
fired. We can express the subexpression \( e_{\downarrow \epsilon_L} \uparrow e \) and
\( \epsilon_L \uparrow e \) in LamFJ when either a method or
processes or the main expression. \( \uparrow e_{\downarrow \epsilon_L} \) is assumed to be a pre-fix of
\( L \), are special run-time expressions and not supposed to appear in either classes, partial methods
or the main expression. \( \uparrow e_{\downarrow \epsilon_L, L}.m(\mathcal{V}) \) that means that \( m \)
is going to be invoked on \( v \). The annotation \( \uparrow e_{\downarrow \epsilon_L, L} \cdot m(\mathcal{V}) \)
denotes a cursor that indicates where method lookup should
start when either a method or proceed are invoked. More
concretely, \( \uparrow e_{\downarrow \epsilon_L, L}(L_i; \cdots; L_j) \# (i \leq n) \) means that the search
for the method definition will start from class \( D \) in
layer \( L_i \). For example, the usual method invocation \( v \cdot e(\mathcal{V}) \), whose receiver and arguments are already reduced to values,
is semantically equivalent to \( v(\mathcal{V})_{\downarrow \epsilon_L, L} \cdot m(\mathcal{V}) \), where an object
new \( C(\mathcal{V}) \) for some values \( \mathcal{V} \) is at the address \( v \). The
triple plays the role of a cursor in the method lookup
procedure and the behavior of super and proceed calls as in
ContextFJ.

An LamFJ program \((C, PT, e)\) consists of a class table
\( CT \) that maps a class name to a class definition, a partial
method table \( PT \) that maps a triple \( C, L, m \) of class, layer,
and method names to a method definition, and an expression
\( e \) that corresponds to the body of the main method. In the
paper, we assume \( CT \) and \( PT \) to be fixed and satisfy the
following sanity conditions:

1. \( CT(C) = \text{class } C \ldots \) for any \( C \in \text{dom}(CT) \).
2. Object \( \notin \text{dom}(CT) \).
3. For every class name \( C \) (except Object) appearing anywhere
in \( CT \), we have \( C \in \text{dom}(CT) \);
4. There are no cycles in the transitive closure of the
extends clauses.
5. \( PT(m, C, L) \) is a method definition for any \((m, C, L) \in \)
\( \text{dom}(PT) \).

### 3.2 Operational semantics

The operational semantics of LamFJ is given by a reduction
relation of the form \( e = e' \) or \( e = e' \) for \( e, e' \). “expression e with store s and
history h at time t is reduced to e' with s' and h' at time t'”, read
“expression e with store s and history h at time t is
reduced to e' with s' and h' at time t'”. A store \( s \) is a partial
function from addresses \( v \) to objects \( \text{new } C(\mathcal{V}) \). We write
\( s _0 \to \text{new } C(\mathcal{V}) \) to denote a partial function that maps
\( s_0 \) to \( \text{new } C(\mathcal{V}) \) and other addresses \( w \to s(w) \). We also
write \( \text{dom}(s) \) for the set \( \{v \mid s(v) \} \) is defined.
A history \( h \) is \( \{\mathcal{V}, F, \mathcal{E}, \mathcal{C} \} \), a set of triples of addresses, times and context
change events on layers. Time stamp t is an integer.

The rules are shown in Figure 6. Note that LamFJ
employs call-by-value and the evaluation order of subexpressions
is deterministic. The property is important because we need
to know in what order layers are activated.

```
fields(C) = \mathcal{C} \mathcal{T} 
fields(\text{Object}) = \bullet 
```

```
class C \triangleq D \{ \mathcal{C} \; \mathcal{T} \ldots \} 
fields(D) = \mathcal{B} \mathcal{G} 
fields(C) = \mathcal{B} \mathcal{G}, \mathcal{C} \mathcal{T} 
```

```
mbody(m, C, L, L) = x.e in D, L' 
```

```
class C \triangleq D \{ \ldots C_0 = m(C) \{ \text{return } e; \} \ldots \} 
mbody(m, C, C', L_0) = x.e in C(C', L_0) 
```

```
class C \triangleq D \{ \ldots \} 
\text{mbody}(m, C, D, L, L) = x.e in E, L' 
```

```
\text{mbody}(m, C, (L'; L_0), L) = x.e in D, L' 
```

```
PT(m, C, L_0) \text{ undefined} 
\text{mbody}(m, C, C', L_0, L) = x.e in D, L' 
```

```
\text{mbody}(m, C, (C', L_0), L) = x.e in D, L' 
```

```
figure 5: Auxiliary functions
```

It represents an expression with a hole (denoted as \( [] \) somewhere inside it. We write \( E[e] \) for either object creation,
field access, method invocation, sequential composition and
event firing obtained by replacing the hole in \( E \) with \( e \).

The second rule reduces an object creation \( \text{new } C(\mathcal{V}) \) to
a new address \( v_0 \) that is not in the domain of \( s \) and is not
equal to the null address, and extends the store so that the
extended store maps \( v_0 \) to \( \text{new } C(\mathcal{V}) \).

The third rule is for field access. Auxiliary function \( \text{fields} \)
takes a class name \( C \) and returns the fields in class \( C \) and its
ancestors (Figure 5).

The forth rule reduces a sequential composition of value
\( v \) and expression \( e \) to \( e \).

The next four rules are for firing events. The first rule of
the four fires an context change event \( \epsilon \) on layer \( L \) globally.
It adds a triple \( (v, t, \epsilon) \) to history \( h \) and increments the time
stamp using function \( \text{succ} \). The second rule of the four fires
a context change event \( \epsilon \) on layer \( L \) to object \( a(v) \). Therefore
it adds \( (v, t, \epsilon) \) to \( h \) and increment the time stamp. The
other two rules are defined similarly.

The last three rules are for method invocation. The only
interesting rule is the first one of the three. It initializes the
cursor of the method lookup procedure. Auxiliary function
\( \text{actives} \) takes a history \( h \) and address \( v \), and returns a se-
quence of layers that are active on the object addressed by
\( v \). It is defined as follows:

\[
\text{actives}(h, v) = \text{seq}(\text{pick}(L, \gamma(p), \theta(p), \theta(p)), L \in \Lambda, \rho(v, L) = h = p) 
\]

\( A \) is the set of all of the layer names. \( \rho(v, L) = h \) is a sub-history
of history h whose elements are related to object v and layer L. More concretely, \( \triangledown (v, L, h) \) contains every triple \((v', t, \epsilon_L')\) in h if \( v' \) is the null address or is equal to \( v \) and \( L' \) is equal to \( L \), i.e., \( \triangledown (v, L, h) = \{ (t, \epsilon | \langle v', t, \epsilon_L' \rangle, v = v', L = L' \} \). \( \gamma(p) \) finds the most recent activation \( \alpha \) or deactivation \( \delta \) in history p, i.e., \( \gamma(p) = \text{maximum} \{ (t, \epsilon) | (t, \epsilon) \in p, \epsilon \in \{ \alpha, \delta \} \} \) where \( \text{maximum} \{ (t_1, \epsilon_1), \ldots, (t_n, \epsilon_n) \} = (t_i, \epsilon) \) and \( t_i \) is the largest among \( \{ t_k | 1 \leq k \leq n \} \). \( \theta_1(p) \) and \( \theta_2(p) \) are the states of the two activation contexts computed from history p, i.e., \( \theta_1(p) = \bigotimes \{ (t, \epsilon) | (t, \epsilon) \in p, \epsilon \in \{ \sigma, \pi^{-1} \} \} \) and \( \theta_2(p) = \bigotimes \{ (t, \epsilon) | (t, \epsilon) \in p, \epsilon \in \{ \pi, \pi^{-1} \} \} \). \( \bigotimes \{ (t_1, t') \ldots \} \) counts how many times with and without blocks are opened and closed, i.e., \( \bigotimes \{ (t_1, t'), \ldots \} = (t_1, t') \otimes (t_2, t') \otimes \ldots \) where \( (t_1, t') \otimes (t_2, t') \otimes \ldots = (\max(t_1, t_2), t'\ldots) \) and \( \epsilon \in \{ \sigma, \pi \} \).

5. CONCLUSIONS AND FUTURE WORK

This paper discusses an issue on language semantics that allows programmers to use multiple layer activation mechanisms to capture contexts with respect to various kinds of things. We claimed that more recently fired context change events should precede others and formalized it by developing a calculus called LamFJ. LamFJ is not only powerful enough to express multiple layer activation mechanisms but also clearly defines combined effects of those mechanisms.

There are several directions for future work. One direction is to support more layer activation mechanisms including implicit/reactive layer activation and structural activation [10]. Another direction is developing an extensible compiler that supports the idea in LamFJ for the future researches and developments of COP languages.
6. REFERENCES


