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.

COP '14, July 28 – August 01 2014, Uppsala, Sweden

http://dx.doi.org/10.1145/2637066.2637068

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] tries 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

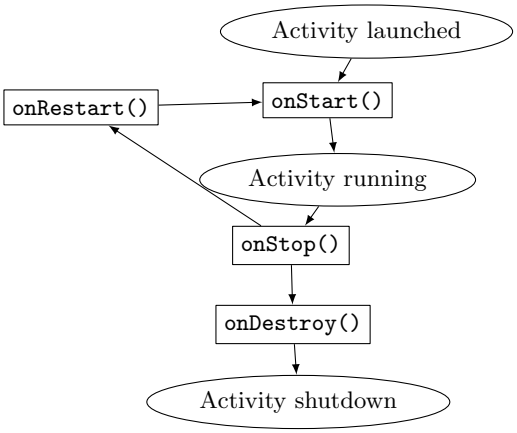

Figure 1: Life cycle of an activity

```
1  class MyActivity extends Activity{
2    void anInitializer(){...}
3    void onStart(){ anInitializer(); ... }
4    void onRestart(){
5      with(Restart){ onStart(); }
6    }
7    layer Restart{
8      void anInitializer(){...}
9    }
10 }
```

Figure 2: A code skeleton of Activity using COP for initialization

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.[1] 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

[1]See `http://developer.android.com/reference/android/app/Activity.html` for details.

```
1  <receiver android:name=".PowerConnRecver">
2  <intent-filter>
3  <action android:name="..ACTION_POWER_CONNECTED"/>
4  <action android:name="..ACTION_POWER_DISCONNECTED"/>
5  </intent-filter>
6  </receiver>
7  ..................................................
8  class PowerConnRecver extends BroadcastReceiver{
9    void onReceive(Context c,Intent intent){
10     if(intent.getIntExtra("plugged",0)==0)
11       deactivate(Plugged);
12     else
13       activate(Plugged);
14   }}
```

Figure 3: Switching the context with respect to power

```
1  with(L){
2    deactivate(L);
3    m1();
4    activate(L);
5    without(L){
6      m2();
7    }
8    m3();
9  }
10 m4();
```

Figure 4: Two layer mechanisms activates and deactivates the same layer

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 m1() at line 3? What about invocations m2() at line 6, m3() at line 8 and m4() at line 10?

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

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 m1, m2, m3 and m4 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:

$$\epsilon_1 < \epsilon_2 < \texttt{m1()} < \epsilon_3 < \epsilon_4 < \texttt{m2()} < \epsilon_4^{-1} < \texttt{m3()} < \epsilon_1^{-1} < \texttt{m4()}$$

mi() denotes invoking method mi(). ϵ_i^{-1} denotes cancel of ϵ_i. ϵ_i denotes the following context change events:

- ϵ_1: entering the with block at line 1
- ϵ_2: deactivating layer L at line 2
- ϵ_3: activating layer L at line 4
- ϵ_4: entering the without block at line 5

Now we can answer the questions. Layer L does not affect invocation m1() because ϵ_2 is the most preceding context change event, which deactivates L. Similarly, it does not affect invocation m2() because ϵ_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 m3() and m4(). The most recent event for m3() is ϵ_4^{-1}, but it cannot be the most preceding context change event because it is cancel. ϵ_4, which is the next most recent event for m3(), 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 m3() is ϵ_3. Because ϵ_3 activates L, L affects m3(). ϵ_3 is also the most preceding context change event for m4(). Therefore, L also affects m4().

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 with and without blocks from and adds *context change events* and *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; m ranges over method names; and v,w and z range over addresses, which include the null address denoted as $\cdot$; x and y range over variables, which include a special variable this. The abstract syntax of LamFJ is as follows:

$$\begin{array}{lcl}
\texttt{CL} & ::= & \texttt{class C} \lhd \texttt{C \{}\overline{\texttt{T f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \\
\texttt{K} & ::= & \texttt{C(}\overline{\texttt{C x}}\texttt{, }\overline{\texttt{C y}}\texttt{)\{super(}\overline{\texttt{x}}\texttt{); this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{y}}\texttt{;\}} \\
\texttt{M} & ::= & \texttt{C m(}\overline{\texttt{C x}}\texttt{)\{return e;\}} \\
\texttt{e} & ::= & \texttt{v} \mid \texttt{x} \mid \texttt{new C(}\overline{\texttt{e}}\texttt{)} \mid \texttt{e.f} \mid \texttt{e.m(}\overline{\texttt{e}}\texttt{)} \mid \texttt{proceed(}\overline{\texttt{e}}\texttt{)} \\
 & & \mid \texttt{e;e} \mid \texttt{super.m(}\overline{\texttt{e}}\texttt{)} \mid \texttt{v<C,}\overline{\texttt{L}}\texttt{,}\overline{\texttt{L}}\texttt{>.m(}\overline{\texttt{e}}\texttt{)} \\
 & & \mid \epsilon_{\texttt{L}} \uparrow \texttt{e} \mid \epsilon_{\texttt{L}}\texttt{@e} \uparrow \texttt{e} \mid \texttt{e} \downarrow \epsilon_{\texttt{L}} \mid \texttt{e} \downarrow \epsilon_{\texttt{L}}\texttt{@e} \\
\epsilon & ::= & \alpha \mid \delta \mid \sigma \mid \sigma^{-1} \mid \pi \mid \pi^{-1}
\end{array}$$

Overlines denote sequences, e.g., $\overline{\texttt{f}}$ stands for a possibly empty sequence $\texttt{f}_1, \cdots, \texttt{f}_n$ and similarly for $\overline{\texttt{C}}$, $\overline{\texttt{x}}$, $\overline{\texttt{e}}$, and so on. The empty sequence is denoted by $\bullet$. We also abbreviate a sequence of pairs, writing "$\overline{\texttt{C f}}$" for "$\texttt{C}_1\ \texttt{f}_1, \cdots, \texttt{C}_n\ \texttt{f}_n$", where n is the length of $\overline{\texttt{C}}$ and $\overline{\texttt{f}}$, and similarly "$\overline{\texttt{C f}}$;" as shorthand for the sequence of declarations "$\texttt{C}_1\ \texttt{f}_1\texttt{;} \ldots \texttt{C}_n\ \texttt{f}_n\texttt{;}$" and "$\texttt{this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;}$" for "$\texttt{this.f}_1\texttt{=f}_1\texttt{;} \ldots \texttt{;this.f}_n\texttt{=f}_n\texttt{;}$". 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 CL consists of its name, its superclass name, field declarations $\overline{\texttt{C}}$ $\overline{\texttt{f}}$, a constructor K, and method definitions $\overline{\texttt{M}}$. 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 $\overline{\texttt{x}}$ 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, super call, proceed call, a special form of method call $\texttt{v<C,}\overline{\texttt{L}}\texttt{,}\overline{\texttt{L}}\texttt{>.m(}\overline{\texttt{v}}\texttt{)}$, or one of the four forms of event firing expressions that fires a context change event ϵ on layer L, denoted as $\epsilon_{\texttt{L}}$. A value is an address v. A context change event is either an activation denoted as α, deactivation denoted as δ, activation counter increment denoted as σ, activation counter decrement denoted as σ^{-1}, deactivation counter increment denoted as π, or deactivation counter decrement denoted as π^{-1}.

Instead of using cancel events to handle the semantics of with and without blocks as shown in Section 2, LamFJ uses activation conters [3]. σ and π are supposed to be fired when entering with and without blocks, respectively. σ^{-1} and π^{-1} are supposed to be fired when escaping from with and without blocks, respectively.

We have four forms of event firing expressions. $\epsilon_{\texttt{L}} \uparrow \texttt{e}$ is supposed to be used to realize the two operators acti-

vate and deactivate in imperative activation. It fires globally the context change event ϵ on layer L. For example, activate(L) and deactivate(L) followed by an expression e is expressed as $\alpha_{\mathtt{L}} \uparrow \mathtt{e}$ and $\delta_{\mathtt{L}} \uparrow \mathtt{e}$, respectively, in LamFJ. $\epsilon_{\mathtt{L}}@\mathtt{e} \uparrow \mathtt{e}'$ is similar. It runs $\mathtt{e}'$ after firing $\epsilon_{\mathtt{L}}$ to an object e, which realizes per-object/agent activation available in EventCJ and ContextErlang. $\mathtt{e} \downarrow \epsilon_{\mathtt{L}}$ is also similar to $\epsilon_{\mathtt{L}} \uparrow \mathtt{e}$, but the subexpression e is evaluated before the event is fired. We can express dynamic scoping using $\epsilon_{\mathtt{L}} \uparrow \mathtt{e}$ and $\epsilon_{\mathtt{L}} \uparrow \mathtt{e}$. For example, with L e and without L e in ContextFJ are expressed as $\sigma_{\mathtt{L}} \uparrow (\mathtt{e} \downarrow \sigma_{\mathtt{L}}^{-1})$ and $\pi_{\mathtt{L}} \uparrow (\mathtt{e} \downarrow \pi_{\mathtt{L}}^{-1})$, respectively.

The expressions v and $\mathtt{v<D,}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m(}\overline{\mathtt{w}}\mathtt{)}$, where $\overline{\mathtt{L}}'$ is assumed to be a prefix of $\overline{\mathtt{L}}$, are special run-time expressions and not supposed to appear in either classes, partial methods or the main expression. $\mathtt{v<D,}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m(}\overline{\mathtt{w}}\mathtt{)}$ means that m is going to be invoked on v. The annotation $\mathtt{<D,}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}\mathtt{>}$ denotes a *cursor* that indicates where method lookup should start when either a method or proceed are invoked. More concretely, $\mathtt{<D,}(\mathtt{L}_1;\cdots;\mathtt{L}_i)\mathtt{,}(\mathtt{L}_1;\cdots;\mathtt{L}_n)\mathtt{>}$ $(i \leq n)$ means that the search for the method definition will start from class D in layer $\mathtt{L}_i$. For example, the usual method invocation $\mathtt{v.m(}\overline{\mathtt{w}}\mathtt{)}$, whose receiver and arguments are already reduced to values, is semantically equivalent to $\mathtt{v}_0\mathtt{<C,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m(}\overline{\mathtt{v}}\mathtt{)}$, where an object $\mathtt{new\ C(}\overline{\mathtt{w}}\mathtt{)}$ for some values $\overline{\mathtt{w}}$ 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 $(CT, PT, \mathtt{e})$ 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, and 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. $CT(\mathtt{C}) = \mathtt{class\ C\ \ldots}$ for any $\mathtt{C} \in \mathit{dom}(CT)$.
2. $\mathtt{Object} \notin \mathit{dom}(CT)$.
3. For every class name C (except Object) appearing anywhere in CT, we have $\mathtt{C} \in \mathit{dom}(CT)$;
4. There are no cycles in the transitive closure of the extends clauses.
5. $PT(\mathtt{m}, \mathtt{C}, \mathtt{L})$ is a method definition for any $(\mathtt{m}, \mathtt{C}, \mathtt{L}) \in \mathit{dom}(PT)$.

3.2 Operational semantics

The operational semantics of LamFJ is given by a reduction relation of the form $\mathtt{e} \mid \mathtt{s}, \mathtt{h}, \mathtt{t} \longrightarrow \mathtt{e}' \mid \mathtt{s}', \mathtt{h}', \mathtt{t}'$, read "expression e with store s and history h at time t is reduced to $\mathtt{e}'$ with $\mathtt{s}'$ and $\mathtt{h}'$ at time $\mathtt{t}'$". A store s is a partial function from addresses v to objects $\mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)}$. We write $[\mathtt{v}_0 \mapsto \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)}]s$ to denote a partial function that maps $\mathtt{v}_0$ to $\mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)}$ and other addresses w to $\mathtt{s(w)}$. We also write $\mathit{dom}(\mathtt{s})$ for the set $\{\mathtt{v} \mid \mathtt{s(v)}\ \text{is defined}\}$. A history h is $\{\langle \overline{\mathtt{v}}, \overline{\mathtt{t}}, \overline{\epsilon_{\mathtt{L}}} \rangle\}$, 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.

$\boxed{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}$

$$\mathit{fields}(\mathtt{Object}) = \bullet$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}\ \ldots\} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}\mathtt{,}\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}}$$

$\boxed{\mathit{mbody}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}', \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \text{in}\ \mathtt{D}, \overline{\mathtt{L}}''}$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D\{\ldots\ \ C_0\ m(}\overline{\mathtt{C}}\ \overline{\mathtt{x}}\mathtt{)\{return\ e;\}\ \ldots\}}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}, \bullet, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \text{in}\ \mathtt{C}, \bullet}$$

$$\frac{PT(\mathtt{m}, \mathtt{C}, \mathtt{L}_0) = \mathtt{C\ m(}\overline{\mathtt{C}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}'\mathtt{;L}_0, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \text{in}\ \mathtt{C}, (\overline{\mathtt{L}}'\mathtt{;L}_0)}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D\{\ldots}\overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{m} \notin \mathtt{M} \qquad \mathit{mbody}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{L}}, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \text{in}\ \mathtt{E}, \overline{\mathtt{L}}'}{\mathit{mbody}(\mathtt{m}, \mathtt{C}, \bullet, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \text{in}\ \mathtt{E}, \overline{\mathtt{L}}'}$$

$$\frac{PT(\mathtt{m}, \mathtt{C}, \mathtt{L}_0)\ \text{undefined} \qquad \mathit{mbody}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}', \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \text{in}\ \mathtt{D}, \overline{\mathtt{L}}''}{\mathit{mbody}(\mathtt{m}, \mathtt{C}, (\overline{\mathtt{L}}'\mathtt{;L}_0), \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \text{in}\ \mathtt{D}, \overline{\mathtt{L}}''}$$

Figure 5: Auxiliary functions

The first rule reduces a subexpression. We define an evaluation context E as follows:

$$\begin{array}{rcl} E & ::= & [] \mid \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{,}E\mathtt{,}\overline{\mathtt{e}}\mathtt{)} \mid E\mathtt{.f} \mid \mathtt{v.m(}\overline{\mathtt{v}}\mathtt{,}E\mathtt{,}\overline{\mathtt{e}}\mathtt{)} \\ & & \mid \mathtt{v<C,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m(}\overline{\mathtt{v}}\mathtt{,}E\mathtt{,}\overline{\mathtt{e}}\mathtt{)} \mid E\mathtt{;e} \mid \epsilon_{\mathtt{L}}@E \uparrow \mathtt{e} \\ & & \mid \mathtt{v} \downarrow \epsilon_{\mathtt{L}}@E \mid E \downarrow \epsilon_{\mathtt{L}}@\mathtt{e} \end{array}$$

It represents an expression with a hole (denoted as []) somewhere inside it. We write $E[\mathtt{e}]$ 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 $\mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)}$ to a new address $\mathtt{v}_0$ 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 $\mathtt{v}_0$ to $\mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)}$.

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

The forth rule reduces a sequencial 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 ϵ on layer L globally. It adds a triple $\langle \cdot, \mathtt{t}, \epsilon_{\mathtt{L}} \rangle$ to history h and increments the time stamp using function *succ*. The second rule of the four fires a context change event ϵ on layer L to object $\mathtt{s(v)}$. Therefore it adds $\langle \mathtt{v}, \mathtt{t}, \epsilon_{\mathtt{L}} \rangle$ 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 *actives* takes a history h and address v, and returns a sequence of layers that are active on the object addressed by v. It is defined as follows:

$$\mathit{actives}(\mathtt{h}, \mathtt{v}) = \mathit{seq}(\{\mathit{pick}(\mathtt{L}, \gamma(p), \theta_1(p), \theta_2(p)) \mid \mathtt{L} \in \Lambda, \triangleright_{(\mathtt{v},\mathtt{L})}\mathtt{h} = p\})$$

Λ is the set of all of the layer names. $\triangleright_{(\mathtt{v},\mathtt{L})}\mathtt{h}$ is a sub-history

$$\frac{\mathtt{e} \mid \mathtt{s},\mathtt{h},\mathtt{t} \longrightarrow \mathtt{e}' \mid \mathtt{s}',\mathtt{h}',\mathtt{t}'}{E[\mathtt{e}] \mid \mathtt{s},\mathtt{h},\mathtt{t} \longrightarrow E[\mathtt{e}'] \mid \mathtt{s}',\mathtt{h}',\mathtt{t}'}$$

$$\frac{\mathtt{v}_0 \notin \mathit{dom}(\mathtt{s}) \cup \{\cdot\}}{\mathtt{new\ C}(\overline{\mathtt{v}}) \mid \mathtt{s},\mathtt{h},\mathtt{t} \longrightarrow \mathtt{v}_0 \mid [\mathtt{v}_0 \mapsto \mathtt{new\ C}(\overline{\mathtt{v}})]\mathtt{s},\mathtt{h},\mathtt{t}}$$

$$\frac{\mathtt{s}(\mathtt{v}) = \mathtt{new\ C}(\overline{\mathtt{w}}) \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\mathtt{v.f}_i \mid \mathtt{s},\mathtt{h},\mathtt{t} \longrightarrow \mathtt{w}_i \mid \mathtt{s},\mathtt{h},\mathtt{t}}$$

$$\mathtt{v;e} \mid \mathtt{s},\mathtt{h},\mathtt{t} \longrightarrow \mathtt{e} \mid \mathtt{s},\mathtt{h},\mathtt{t}$$

$$\epsilon_{\mathtt{L}} \uparrow \mathtt{e} \mid \mathtt{s},\mathtt{h},\mathtt{t} \longrightarrow \mathtt{e} \mid \mathtt{s},\mathtt{h} \cup \{\langle \cdot,\mathtt{t},\epsilon_{\mathtt{L}}\rangle\}, \mathit{succ}(\mathtt{t})$$

$$\epsilon_{\mathtt{L}}@\mathtt{v} \uparrow \mathtt{e} \mid \mathtt{s},\mathtt{h},\mathtt{t} \longrightarrow \mathtt{e} \mid \mathtt{s},\mathtt{h} \cup \{\langle \mathtt{v},\mathtt{t},\epsilon_{\mathtt{L}}\rangle\}, \mathit{succ}(\mathtt{t})$$

$$\mathtt{v} \downarrow \epsilon_{\mathtt{L}} \mid \mathtt{s},\mathtt{h},\mathtt{t} \longrightarrow \mathtt{v} \mid \mathtt{s},\mathtt{h} \cup \{\langle \cdot,\mathtt{t},\epsilon_{\mathtt{L}}\rangle\}, \mathit{succ}(\mathtt{t})$$

$$\mathtt{v}' \downarrow \epsilon_{\mathtt{L}}@\mathtt{v} \mid \mathtt{s},\mathtt{h},\mathtt{t} \longrightarrow \mathtt{v}' \mid \mathtt{s},\mathtt{h} \cup \{\langle \mathtt{v},\mathtt{t},\epsilon_{\mathtt{L}}\rangle\}, \mathit{succ}(\mathtt{t})$$

$$\frac{\mathtt{s}(\mathtt{v}) = \mathtt{new\ C}(\overline{\mathtt{z}}) \qquad \mathit{actives}(\mathtt{h},\mathtt{v}) = \overline{\mathtt{L}}}{\mathtt{v<C,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m}(\overline{\mathtt{w}}) \mid \mathtt{s},\mathtt{h},\mathtt{t} \longrightarrow \mathtt{e} \mid \mathtt{s}',\mathtt{h}',\mathtt{t}'}{\mathtt{v.m}(\overline{\mathtt{w}}) \mid \mathtt{s},\mathtt{h},\mathtt{t} \longrightarrow \mathtt{e} \mid \mathtt{s}',\mathtt{h}',\mathtt{t}'}$$

$$\frac{\begin{array}{c}\mathtt{class\ C}'' \lhd \mathtt{D\{...\}} \\ \mathit{mbody}(\mathtt{m},\mathtt{C}',\overline{\mathtt{L}}',\overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e\ in\ C}'',\bullet\end{array}}{\begin{array}{c}\mathtt{v<C}',\overline{\mathtt{L}}',\overline{\mathtt{L}}\mathtt{>.m}(\overline{\mathtt{w}}) \mid \mathtt{s},\mathtt{h},\mathtt{t} \longrightarrow \\ \left[\begin{array}{ll}\mathtt{v} & /\mathtt{this} \\ \overline{\mathtt{w}} & /\overline{\mathtt{x}} \\ \mathtt{v<D,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>} & /\mathtt{super}\end{array}\right] \mathtt{e} \mid \mathtt{s},\mathtt{h},\mathtt{t}\end{array}}$$

$$\frac{\begin{array}{c}\mathit{mbody}(\mathtt{m},\mathtt{C}',\overline{\mathtt{L}}',\overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e\ in\ C}'',(\overline{\mathtt{L}}'';\mathtt{L}_0) \\ \mathtt{class\ C}'' \lhd \mathtt{D\{...\}}\end{array}}{\begin{array}{c}\mathtt{v<C}',\overline{\mathtt{L}}',\overline{\mathtt{L}}\mathtt{>.m}(\overline{\mathtt{w}}) \mid \mathtt{s},\mathtt{h},\mathtt{t} \longrightarrow \\ \left[\begin{array}{ll}\mathtt{v} & /\mathtt{this} \\ \overline{\mathtt{w}} & /\overline{\mathtt{x}} \\ \mathtt{v<C}'',\overline{\mathtt{L}}'',\overline{\mathtt{L}}\mathtt{>.m} & /\mathtt{proceed} \\ \mathtt{v<D,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>} & /\mathtt{super}\end{array}\right] \mathtt{e} \mid \mathtt{s},\mathtt{h},\mathtt{t}\end{array}}$$

Figure 6: Reduction rules

of history h whose elements are related to object v and layer L. More concretely, $\triangleright_{(\mathtt{v},\mathtt{L})}\mathtt{h}$ contains every triple $\langle \mathtt{v}',\mathtt{t},\epsilon_{\mathtt{L}'}\rangle$ in h if v′ is the null address or is equal to v and L′ is equal to L, i.e., $\triangleright_{(\mathtt{v},\mathtt{L})}\mathtt{h} = \{\langle \mathtt{t},\epsilon\rangle \mid \langle \mathtt{v}',\mathtt{t},\epsilon_{\mathtt{L}'}\rangle, \mathtt{v} = \mathtt{v}', \mathtt{L} = \mathtt{L}'\}$. $\gamma(p)$ finds the most recent activation α or deactivation δ in history p, i.e., $\gamma(p) = \mathit{maximum}\{\langle \mathtt{t},\epsilon\rangle \mid \langle \mathtt{t},\epsilon\rangle \in p, \epsilon \in \{\alpha,\delta\}\}$ where $\mathit{maximum}\{\langle \mathtt{t}_1,\epsilon_1\rangle,\cdots,\langle \mathtt{t}_n,\epsilon_n\rangle\} = \langle \mathtt{t}_i,\epsilon_i\rangle$ and $\mathtt{t}_i$ is the largest among $\{\mathtt{t}_k \mid 1 \le k \le n\}$. $\theta_1(p)$ and $\theta_2(p)$ are the states of the two activation conters computed from history p, i.e., $\theta_1(p) = \bigotimes\{\langle \mathtt{t},\epsilon\rangle \mid \langle \mathtt{t},\epsilon\rangle \in p, \epsilon \in \{\sigma,\sigma^{-1}\}\}$ and $\theta_2(p) = \bigotimes\{\langle \mathtt{t},\epsilon\rangle \mid \langle \mathtt{t},\epsilon\rangle \in p, \epsilon \in \{\pi,\pi^{-1}\}\}$. $\bigotimes\{\langle \mathtt{t}_1,\iota^i\rangle,\cdots\}$ counts how many times with and without blocks are opened and closed, i.e., $\bigotimes\{\langle \mathtt{t}_1,\iota^i\rangle,\cdots\} = \langle \mathtt{t}_1,\iota^i\rangle \otimes \langle \mathtt{t}_2,\iota^j\rangle \otimes \cdots$ where $\langle \mathtt{t}_1,\iota^i\rangle \otimes \langle \mathtt{t}_2,\iota^j\rangle = \langle \mathit{max}(\mathtt{t}_1,\mathtt{t}_2),\iota^{i+j}\rangle$ and $\iota \in \{\sigma,\pi\}$. Here we regard σ and π as σ^1 and π^1, respectively. *pick* takes a layer name, a pair of a time stamp and context change event which is either α or δ, and two pairs of a time stamp and context change event which is either σ^i or π^i for some integer i, and returns a pair of a time stamp and the layer name. The time stamp is -1 if L is not active and intuitively the maximum value among the three time stamp given as the arguments otherwise. It is defined as follows:

$\mathit{pick}(\mathtt{L},\langle \mathtt{t},\epsilon\rangle,\langle \mathtt{t}',\sigma^i\rangle,\langle \mathtt{t}'',\pi^j\rangle) = \langle \mathtt{t}''',\mathtt{L}\rangle$, where

$$\mathtt{t}''' = \begin{cases} -1 & \text{if } \mathtt{t} < \mathtt{t}'' \wedge (i = 0 \vee \mathtt{t}' < \mathtt{t}'') \wedge j \neq 0 \\ \delta(\mathtt{t},\epsilon) & \text{if } (\mathtt{t} > \mathtt{t}'' \wedge i = 0) \vee (\mathtt{t} > \mathtt{t}' \wedge j = 0) \\ \mathtt{t}' & \text{if } (\mathtt{t}' > \mathtt{t}'' \vee j = 0) \wedge \mathtt{t} < \mathtt{t}' \wedge i \neq 0 \end{cases}$$

and $\delta(\mathtt{t},\epsilon) = \mathtt{t}$ if $\epsilon = \alpha$ and -1 otherwise. The last function *seq* takes a set of pairs of a time stamp and a layer name, and creates a sequence of layer names by simply sorting the elements in the input in the ascendant order removes the elements whose time stamp is a negative number and drops time stamps.

The others are similar to ContextFJ. Auxiliary function *mbody* takes a method name m, class name C and two sequences of layer names $\overline{\mathtt{L}}'$ and $\overline{\mathtt{L}}$ and returns the body expression to be evaluated if m is invoked to an object of type C in the environment where the layers $\overline{\mathtt{L}}$ are active (Figure 5).

4. RELATED WORK

ContextJS [10] is a COP language based on JavaScript. It provides an open implementation platform in which programmers can implement their own layer activation mechanisms thanks to the powerful meta programming facilities of JavaScript. The study shows how to realize dynamic scoping, imperative activation and structural activation that captures contexts with respect to data structures on the platform. The big difference between our work and ContextJS is whether or not use meta programming. Meta programming is powerful but is easily breaks language semantics. Our approach tries to find a minimal language mechanism that is powerful but still permits reasoning about about the program.

ServalCJ [8, 9] is a Java-based COP language that allows programmer to use imperative activation, implicit activation, dynamic scoping and per-object activation freely in a program. It unifies the four mechanisms by introducing three language constructs, namely context declaration, activate declaration and context group declaration. Using ServalCJ as a frontend language and LamFJ as a foundation of intermediate language would be one option to develop a practical COP language that supports multiple layer activation mechanisms.

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

[1] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *SC '10*, pages 50–65, 2010.

[2] Engineer Bainomugisha, Jorge Vallejos, Coen De Roover, Andoni Lombide Carreton, and Wolfgang De Meuter. Interruptible context-dependent executions: A fresh look at programming context-aware applications. In *Onward! '12*, pages 67–84, 2012.

[3] Nicolás Cardozo, Sebastián González, Kim Mens, and Theo D'Hondt. Safer context (de)activation: Through the prompt-loyal strategy. In *COP'11*, pages 2:1–2:6, 2011.

[4] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-C: bringing context to mobile platform programming. In *SLE'10*, pages 246–265, 2011.

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

[6] Robert Hirschfeld, Atsushi Igarashi, and Hidehiko Masuhara. ContextFJ: a minimal core calculus for context-oriented programming. In *FOAL '11*, pages 19–23, 2011.

[7] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: a context-oriented programming language with declarative event-based context transition. In *AOSD '11*, pages 253–264, 2011.

[8] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. A unified context activation mechanism. In *COP'13*, pages 2:1–2:6, 2013.

[9] Tetsuo Kamina, Tomoyuki Aotani, Hidehiko Masuhara, and Tetsuo Tamai. Context-oriented software engineering: A modularity vision. In *MODULARITY '14*, pages 85–98, 2014.

[10] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in ContextJS. *Science of Computer Programming*, 76(12):1194–1209, December.

[11] Michael Haupt Malte Appeltauer, Robert Hirschfeld and Hidehiko Masuhara. ContextJ: Context-oriented programming with java. *Computer Software*, 28(1):272–292, 2011.

[12] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801–1817, 2012.

[13] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. ContextErlang: Introducing context-oriented programming in the actor model. In *AOSD'12*, pages 191–202, 2012.

[14] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: Beyond layers. In *ICDL '07*, pages 143–156, 2007.