1. Introduction

Context-oriented programming (COP) [10] modularizes context-dependent behavioral variations. Each variation is dynamically activated and deactivated in response to changes of execution contexts such as internal program states and external situations of the executing environment.

For example, a pedestrian navigation system on a smart phone displays a different map (behavioral variations) according to whether he is outdoors and indoors (execution contexts). Another example is an advanced user interface that provides a different set of menu items (behavioral variations) according to the focused component (execution contexts).

There are several languages and mechanisms to support and achieve COP. ContextJ [10], JCop [2], ContextL [5] and EventCJ [12] provide partial methods and layers to modularize behavioral variations. A partial method defines a variation that overrides the behavior of the original method. A layer groups partial methods and is used to activate/deactivate behavioral variations; i.e., activation/deactivation of behavioral variations is achieved through activating/deactivating layers. Subjective-C [9] achieves COP through annotated methods and contexts, which are similar to partial methods and layers, respectively. In ContextErlang [15], a variation is implemented within a standard Erlang module that uses the special API for contextual dispatch. There are also several models to manage active variations. ContextJ, JCop and ContextL manage activation of variations on a per calling context basis, i.e., active variations affect only the methods executed within the specific calling context. EventCJ and ContextErlang manage them on a per-object/agent basis. Subjective-C and also EventCJ manage them globally, i.e., active variations affect every object in the program.

From the viewpoint of semantics, a COP language can be explained in terms of dynamic dispatch based on active behavioral variations. An execution of a program manages active variations on a per object basis. When a program executes a partial method \( m \), the runtime dispatches to the body of \( m \) defined in one of active layers according to their precedence; i.e., the body in the active layer that take precedence over all other active layers is dispatched to. We assume in this paper that the precedence is given statically, while it changes dynamically in most practical COP languages such as JCop and EventCJ. We say that a partial method call is dynamic if there are two or more layers that defines the partial method \( m \). Conversely, we say that a partial method call is static if it specifies exactly one body to be dispatched to.

This paper proposes an activation order analysis in a COP language. An activation order is the order of activation/deactivation of behavioral variations in each object. The analysis checks, given a program and the set of acceptable orders of activation/deactivation of variations, whether all of the possible orders
of the activation/deactivation in the program are in the set of orders. We call the given set of orders specification on activation orders (or simply specification) in the paper.

The analysis is useful as a fundamental technique for model checking and optimization of context-oriented programs. For example, suppose we want to verify that two layers are not active at a time on any object. A basic technique for such a task would assume an order of activation and deactivation of each layer for each object and checks whether it is possible for the two layers to be active at a time if layers are activated/deactivated according to the assumed order. The activation order analysis helps to check whether the program follows the assumption. Optimization can be considered as a variant of a verification. For example, we can replace a dynamic partial method call with a static one if we can verify that the program never activates any layer except for one specific layer, or a layer with the highest precedence is always active when the partial method call is executed.

Technical contributions of the paper are as follows:

- We define the activation order analysis in terms of a resource usage analysis proposed by Igarashi and Kobayashi [11]. Each object that manages activation of layers is modeled as resource, and each activation and deactivation of layers is modeled as use.
- We give a call-by-value lambda calculus $\lambda^{\text{COP}}$ as a simple formalization of our analysis. It is developed on top of $\lambda^R$ [11]. A resource and access correspond to an object and operation that activates/deactivates a layer. The dynamic dispatch in a COP language is modeled by using a special conditional branch of the form case $x$ of $(L, M_1, M_2)$, which means that $M_1$ is executed if the layer $L$ is active in the resource bound to $x$ and $M_2$ is executed otherwise. This model reflects our assumption that the precedence of layers is given statically. For example, if the term $M_2$ also is case $y$ of $(L', M_1', M_2')$, it means that $L$ takes precedence over $L'$ statically.
- We give a novel type system for $\lambda^{\text{COP}}$ that checks whether every resource is accessed according to the specification. Path sensitivity is achieved by the non-determinism of applications of typing rules. In other words, our typing rules are not syntax directed. Type inference is still possible similarly to $\lambda^R$, but is costly especially if the program heavily uses the special conditional branch with no loop.

Rest of the paper is structured as follows. Section 2 gives a brief introduction of context-oriented programming by using a simple two-dimensional action game as an example. Section 3 presents our target language and type system formally with the sketch of the proofs of its correctness. After discussing related work in Section 4, we conclude in Section 5.

2. Motivation

This section introduces a simple two-dimensional action game to explain the basic concept of context-oriented programming and explains the benefits of analyzing activation orders.

We employ a hypothetical COP language similar to EventCJ [12] as an example COP language to make the readers free from non-essential parts in actual COP languages from the viewpoint of semantics. The hypothetical language provides layers and partial methods to modularize behavioral variations, layer transition rules to specify which layers are activated/deactivated, and a special function $\text{ev}(x, 1)$ that applies the layer transition rule 1 to the object bound to $x$. We assume that the precedence of layers are given statically in the language. We do not give the syntax of the layer transition rules in the language because it is enough to explain which layers are activated and deactivated.

2.1 Example: Two Dimensional Action Game

The task of the player of our two-dimensional action game is to move the player’s character ($pc$ for short) to the goal, avoiding bad guys and getting helpful items namely hearts and stars.

The game can be explained as follows. $pc$ gets an item (a heart or star) if he touches it. He can have at most two hearts and one star. He looks largely if he has one or two hearts. Moreover, he looks armed and shiny if he has two hearts and one star, respectively. Changes of the visual are smoothly displayed by using animations. If $pc$ touches an item or bad guy, he reacts as follows:

1. $pc$ is dead and the game is over if he touches a bad guy and has no item.

The readers familiar with EventCJ should understand that $\text{ev}(x, 1)$ is placed at every join point shadow [13] that matches the pointcut of the event 1. The object bound to $x$ is what is specified in the $\text{sendTo}$ clause.
(2) $pc$ may lose a heart if he touches a bad guy and has exactly one heart.
(3) $pc$ may lose two hearts if he has exactly two hearts and touches a bad guy.
(4) $pc$ may lose a star if he has it and touches a bad guy.

Fig. 1 A state transition diagram of $pc$

Fig. 2 Reactions to touching a bad guy in a COP language

<table>
<thead>
<tr>
<th>checked layers</th>
<th>deactivated layers</th>
<th>activated layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeSmall</td>
<td>Large</td>
<td>Large</td>
</tr>
<tr>
<td></td>
<td>Armed</td>
<td>Armed</td>
</tr>
<tr>
<td>BeKilled</td>
<td>Winning</td>
<td>Winning</td>
</tr>
<tr>
<td>BeMortal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1 The layer transition rules BeSmall, BeKilled and BeMortal

5–10 declare four layers Large, Armed, Winning and Dead, each of which define the reactions to touching a bad guy in $StL$, $StA$, $StW$ and $StD$. The reaction in $StS$ is defined in the Character class as the original behavior. We assume that Winning takes precedence over Large and Armed. The layers define partial methods attacked, lose and stopEffect that override the original behavior of attacked (line 2), lose (line 3) and stopEffect (line 4), respectively. For example, the partial method lose in Large overrides the behavior of the original lose to apply the layer transition rule BeSmall. Other partial methods are understood similarly.

The three layer transition rules BeSmall, BeKilled and BeMortal are defined as Table 1. BeSmall specifies Large and Armed as the layers that get inactive if Large and Armed are active, respectively. BeKilled specifies only the layer that gets active, that is, Dead. BeMortal specifies Winning as the layer that is checked and gets inactive.
The behavior of the program is as follows. When the runtime calls `attacked` to a `Character` object, it checks whether the layers `Armed` and `Winning` are active or not in the object. If they are active, the runtime executes the partial methods at lines 6 and 8, respectively. The original `attacked` is executed if neither `Armed` nor `Winning` is active on the object. If `Large` is active, it affects the behavior of `lose` called from the original `attacked`. The runtime checks whether `Large` as well as `Armed` are active or not, and if `Large` is active, the runtime executes the partial method in `Large`. Each partial method in `Winning` is always executed if `Winning` is active when `attacked` or `stopEffect` is called. This is because the layer takes precedence over the others.

Note that the transitions with the label $\mathbb{L}$ in Figure 1 agree with the behavior of the program. `Dead` is activated on the `Character` object with no active layer if `attacked` is called, which corresponds to the transition from $StS$ to $StD$ with the label $\mathbb{L}$. If `Large` is active when `attacked` is called, `Large` is deactivated, which corresponds to the transition from $StL$ to $StS$ with the label $\mathbb{L}$. Transitions from composite states (i.e., $StW$ & $StL$ and $StW$ & $StA$) are achieved by the cases when `Winning` and either `Large` or `Armed` are active. For example, if `Winning` and `Large` are active on the `Character` object and `attacked` is called, `BeMortal` is applied randomly. Thus `Winning` may be deactivated, but `Large` is still active. This corresponds to the transition from $StW$ & $StL$ to $StL$ with the label $\mathbb{L}$.

It is not possible to skip checking whether `Armed` is active or not when `lose` is called from the original `attacked`, even though `Large` and `Armed` is exclusively activated and the original `attacked` is not executed if `Armed` is active. This is because the runtime does not know in what order each layer is activated/deactivated. In other words, it is necessary to know in what order each layer is activated/deactivated to achieve this kind of optimization.

### 2.3 Activation Order Analysis

Activation order analysis checks, given a program and specification, whether every possible order of layer activation/deactivation is in the specification. It is useful to verify and optimize the program.

This section shows a simple example for each of verification and optimization. We assume for simplicity that the specification is given by using a regular expression over the names of layer transition rules.\footnote{This assumption is stronger than our formalization given in Section 3. Our formalization employs prefix closed sets rather than regular expressions as in \cite{11}.}

Suppose that the `main` method is defined as follows, which simulates that `pc` may first touch a heart and then touches a star and bad guy in the order:

```java
Character pc=/*initialize the player character*/;
if(random()) { ev(pc,BeStrong); }
ev(pc,BeImmortal); pc.attacked();
```

The two layer transition rules `BeStrong` and `BeImmortal` are defined as follows. `BeStrong` activates `Large` if there are no active layers on `pc` other than `Winning`, and `BeImmortal` activates `Armed` if `Large` is active on `pc`. `BeImmortal` simply deactivates `Winning`.

A simple verification of the program is to check whether every possible order of the applications of the layer transition rules is in the given specification. Suppose that the specification is `BeStrong + BeImmortal + BeMortal`, which means that `BeStrong` is applied zero or more times firstly, then `BeImmortal` is applied exactly once and finally `BeMortal` is applied zero or more times. We can conclude that the specification is satisfied because the possible order is either `BeStrong + BeImmortal + BeMortal` or `BeImmortal + BeMortal`, which can be inferred as follows. The runtime may apply `BeStrong` firstly during executing the program, then applies `BeImmortal` once. Therefore at least `Winning` is active in `pc` before `attacked` is called. As `Winning` precedes other layers, the partial method `attacked` within `Winning` is executed and the runtime reaches `ev(this,BeMortal)` (line 9).

An example of the optimization is similar. In the above program, it is easy to know that the runtime enters the partial method in `Winning` when `attacked` is called. Therefore, we can optimize the `main` method so that the runtime does not check the active layers to call `attacked` and just executes `attacked` in `Winning`. One easy way to realize the optimization is generating layer composite classes [6], each of which is specialized to the objects with a specific set of active layers. In our example, we should have three composite classes `CharacterW`, `CharacterL`, and `CharacterWL`, where `CharacterW` is specialized for the case that `Winning` is
active; Characterv is for Large; and Charactervl is for both of Winning and Large. Note that no other combination are possible because the activation orders satisfies Characterv, Charactervl. The layer transition rules are translated so that it specifies the transition among the composite and original classes. For example, BeImmortal changes Charactervl and Character to Charactervl and Characterv, respectively. In both of Charactervl and Characterv, the behavior of the method attacked is statically replaced with the behavior of the partial method in Winning. Thus the program can reach the partial method without checking the active layers.

3. Formalization of Our Activation Order Analysis

We can analyze in what order each transition rules are applied to an object via a resource usage analysis by considering each object with active layers as a resource and \( ev(x, l) \) as an primitive operation that accesses the object bound to \( x \).

In this section, we define \( \lambda^{\text{COP}} \), a call-by-value lambda calculus with resources and layers, as a formalization of our activation order analysis. It is developed on top of \( \lambda^{\text{R}} \) [11], and models the layered functions and layer transitions.

3.1 Syntax

We assume a finite set \( \mathcal{L} \) of labels, ranged over by \( l \), which correspond to names of layer transition rules. A trace is a sequence consisting of labels and the special label \( \bot \). Formally, the set \( \mathcal{L}^{\ast} \) of traces is \( \mathcal{L}^{\ast} \cup \{ s \mid s \in \mathcal{L}^{\ast} \} \) where \( \mathcal{L}^{\ast} \) is the set of finite sequences of elements of \( \mathcal{L} \) and \( \bot \) denotes the evaluation terminates normally.

A trace represents in what order the layer transition rules have been applied to an object at a certain point of the execution of a program. A trace \( l_1, l_2, \cdots, l_k \) means that the layer transition rules \( l_1, l_2, \cdots, l_k \) are applied to the object in the order. A trace \( l_1 l_2 \cdots l_k \bot \) means that the evaluation has terminated after the layer transition rules \( l_1 l_2 \cdots l_k \) are applied in the order.

A trace set is a set of traces that is closed under the prefix operation. We write \( S^{\#} \) for the set of prefixes of elements of the set \( S \). We call a set \( S \) of traces a trace set if \( S^{\#} = S \), and use the metavariable \( \Phi \) to denote a trace set.

The terms \( \mathcal{M} \) ranged over by \( M \), values \( \mathcal{V} \) ranged over by \( v \) and transition rules \( \mathcal{A} \) ranged over by \( a \) are defined as follows:

\[
M ::= \text{true} \mid \text{false} \mid x \mid \text{fun}(f, x, M) \mid \text{if } M_1 \text{ then } M_2 \text{ else } M_3 \\
    \mid M_1 M_2 \mid \text{new}(\Phi) \mid M^{(x)} \mid \text{case } x \text{ of } (L, M, M) \mid ev(M, l) \\
v ::= \text{true} \mid \text{false} \mid x \mid \text{fun}(f, x, M) \\
a ::= \text{activate}(L) \mid \text{deactivate}(L)
\]

The metavariable \( x \) ranges over variables; \( L \) and \( I \) range over layer names; \( l \) ranges over labels \( \mathcal{L} \) that correspond to names of layer transition rules. We use the overline to represent a sequence, e.g., \( \overline{L} \) is \( L_1 : L_2 : \cdots : L_n \).

\( \text{fun}(f, x, M) \) is a recursive function where \( x \) is bound to the argument and \( f \) is bound to the function itself within the term \( M \), i.e., \( f x = M \). \( \text{new}(\Phi) \) is the primitive operation for creating an object. The trace set \( \Phi \) specifies the order of layer transition rules (i.e., specification) applied to the object. \( M^{(x)} \) is the same as \( M \) except that the evaluation gets stuck if the object bound to \( x \) escapes from \( M \). The escape information helps us to improve the accuracy of our analysis. For that purpose, we assume that a separate escape analysis confirms that \( x \) does not escape from \( M \) in \( M^{(x)} \).

The last two forms are newly introduced. \( \text{case } x \text{ of } (L, M_1, M_2) \) is a special form of the conditional branch that checks the layer \( L \) is active on the object bound to \( x \) or not. \( M_1 \) is evaluated when it holds. \( M_2 \) is evaluated otherwise. It is supposed to be used to define the layered function. \( ev(M, l) \) sends the event \( l \) to the object \( M \) so that the active layers on the object \( M \) changes according to the transition rule associated with \( l \). The value is either \( \text{true} \) or \( \text{false} \).

For example, the method \( \text{lose} \) in Figure 2 can be modeled in our language as

\[
\text{fun}(\text{lose}, x, \text{case } x \text{ of } (\text{Large}, M_1, M_2)) \text{ where} \\
    M_1 \triangleq ev(x, l_s); \text{true} \\
    M_2 \triangleq \text{case } x \text{ of } (\text{Armed}, ev(x, l_s); \text{true}, ev(x, l_k); \text{true})
\]

Here, the labels \( l_s \) and \( l_k \) correspond to the names of the layer transition rules BeSmall and Bekilled, respectively. \( M_1; M_2 \) is a short form of if \( M_1 \) then \( M_2 \) else \( M_2 \).

A layer transition rule consists of either \( \text{activate}(L) \) or \( \text{deactivate}(L) \), which specifies the layer to be activated and deactivated respectively. We use the metavariable \( A \) that ranges over the set of pairs of a label and transition rule \( (l, a) \), i.e., \( A = \{(l, a)\} \).
\[ z \text{ fresh} \]
\[
\frac{(H, E[\text{new}^\Phi()]) \leadsto (H \cup \{z \mapsto (\Phi, \bullet), E[z]\})}{(H \cup \{z \mapsto (\Phi, T), E[z]\}) \leadsto (H \cup \{z \mapsto (\Phi^{-1}, T), E[z]\})}
\]
\[
\frac{\Phi^{-1} \neq \emptyset}{(H \cup \{x \mapsto (\Phi, T), E[ev(x, l)]\}) \leadsto (H \cup \{x \mapsto (\Phi^{-1}, T), E[true]\})}
\]
\[\Phi^{-1} = \emptyset \]
\[
\frac{(H \cup \{x \mapsto (\Phi, L), E[ev(x, l)]\}) \leadsto \text{Error}}{H = H' \cup \{x \mapsto (\Phi, T)\} \quad L \notin T}
\]
\[
\frac{(H, E[\text{case } x \text{ of } (L, M_1, M_2)]) \leadsto (H, E[M_1])}{(H, E[\text{case } x \text{ of } (L, M_1, M_2)]) \leadsto (H, E[M_2])}
\]
\[x \notin \text{FV}(v)\]
\[
\frac{(H, E[[v^{(x)}]]) \leadsto (H, E[v])}{(H, E[\text{fun}(f, x, M)] v]) \leadsto (H, E[[\text{fun}(f, x, M)]/f, v/x] M)}
\]
\[
\frac{(H, E[\text{if } \text{true then } M_1 \text{ else } M_2]) \leadsto (H, E[M_1])}{(H, E[\text{if } \text{false then } M_1 \text{ else } M_2]) \leadsto (H, E[M_2])}
\]

A λ_{\text{COP}} program is given as \((A, M)\). We assume that \(A\) is fixed in the following sections for simplicity.

### 3.2 Dynamic Semantics

Our operational semantics is given by the reduction relation among pairs of a term \(M\) and heap \(H\), which keeps track of the states of objects. The state merely captures the application order of layer transition rules allowed for the object and the set of active layers; i.e., we don’t model any values internal to the object such as values of fields.

A heap \(H\) is a finite mapping from variables to pairs of a trace set \(\Phi\) and set of active layers \(\mathcal{T}\). We write \((\mathcal{T} \mapsto (\Phi, L))\) for the heap \(H\) such that \(\text{dom}(H) = \{\mathcal{T}\}\) and \(H(x_i) = (\Phi_i, L_i)\) if \(x_i \in \text{dom}(H)\). If \(\text{dom}(H_1) \cap \text{dom}(H_2) = \emptyset\), we write \(H_1 \cup H_2\) for the heap \(H\) such that \(\text{dom}(H) = \text{dom}(H_1) \cup \text{dom}(H_2)\) and \(H(x) = H_1(x)\) for \(x \in \text{dom}(H_1)\).

We write \([M_1/x_1, \ldots, M_n/x_n]\) for the capture-avoiding simultaneous substitution of \(M_i\) for \(x_i\) and \(\Phi^{-1}\) for \(\{s|s \in \Phi\}\), which removes the label \(l\) at the head of traces in \(\Phi\). \(\text{FV}(M)\) denotes the set of free variables in \(M\). The auxiliary function \(\text{cly}(\mathcal{T}, l)\) applies the transition rule \(l\) to \(\mathcal{T}\), i.e.,

\[
\text{cly}(\mathcal{T}, l) = \begin{cases} 
\mathcal{T}_L : L & \text{if } (l, \text{activate}(L)) \in A \\
\mathcal{T}_L^{-1} & \text{if } (l, \text{deactivate}(L)) \in A \\
\mathcal{T} & \text{otherwise}
\end{cases}
\]

where \(\mathcal{T}_L^{-1}\) is the sequence of layers that contains the same layers to \(\mathcal{T}\) but for \(L\). For example, \(\text{cly}(\text{Winning} : \text{Armed}, \text{deactivate}(\text{Winning})) = \text{Armed} : \text{Winning}\).

The evaluation contexts, ranged over by \(E\), is defined as follows:

\[
E ::= [] \mid \text{if } E \text{ then } M_1 \text{ else } M_2 \mid E \ M \mid v \ E \mid E^{(x)} \mid \text{ev}(E, l)
\]

We write \(E[M]\) for the expressions obtained by replacing [] in \(E\) with \(M\).

The reduction relation \((H, M) \leadsto P\), where \(P\) is either a pair \((H', M')\) or \(\text{Error}\), is the least relation closed under the rules in Fig. 3. \(\text{R-New}\) is for fresh object creation. The following two rules \(\text{R-Ev}\) and \(\text{R-EvErr}\) express a successful and rejected application of the layer transition rule \(l\). If \(\Phi^{-1}\) is not empty, the result is \text{true} and the layer transition rules can be applied to \(x\) according to \(\Phi^{-1}\) in the following reductions (\(\text{R-Ev}\)). If no trace begins with \(l\) (i.e., \(\Phi^{-1} = \emptyset\)), the result is \text{Error} (\(\text{R-EvErr}\)). \(\text{R-CaseMatch}\) and \(\text{R-CaseNotMatch}\) selects the term to be reduced with respect to the active layers of the object \(x; M_1\) is reduced if \(L\) is active; \(M_2\) is reduced otherwise. \(\text{R-Echeck}\) confirms that the variable \(x\) does not escape. \(\text{R-App}\) reduces the function application by replacing the variable \(f\) with the function itself and \(x\) with the argument.

We write \(\leadsto^*\) for the reflexive and transitive closure of \(\leadsto\).

### 3.3 Usage

Let the set \(\mathcal{L}^l\), ranged over by \(l\), be \(\mathcal{L} \cup \{1\}\). The label 1 is a special label used to count the number of function applications. The set of usages \(\mathcal{U}\) ranged over
\( \emptyset \circ \emptyset = \emptyset \quad \bullet \circ = \emptyset \quad \emptyset \otimes U = U \quad 0; U = U \quad U; 0 = U \)

\( U_1 \otimes U_2 = U_2 \otimes U_1 \quad U_1 \oplus U_2 = U_2 \oplus U_1 \quad \mu \alpha.U \leq [\mu \alpha.U/\alpha]U \)

\( \emptyset (U_1 \circ U_2) = (U_1 \circ (U_2 \circ U_1)) \quad \bullet U_1; U_2 = \emptyset U_1 \odot U_2 \quad U_1 \odot U_2 \leq U_1 \)

\( A(l) \) is not related to \( L \quad A(l) = \text{activate}(L) \)

\( U;l; L; U' \leq U; L ; U'' \quad U;l; L; U' \leq U;l; U'' \)

\( U;l; -L; U' \leq U;l; U'' \quad U;l; -L; U' \leq U;l; U'' \)

**Fig. 4** Relation \( U \leq U \)

\[
\begin{align*}
\emptyset U & \xrightarrow{U} U' \\
\bullet U & \xrightarrow{U'} \\
\circ U & \xrightarrow{U''} \\
U_1 & \xrightarrow{U_1'} \\
\circ U_1 & \xrightarrow{U_1''} \\
U_1 \oplus U_2 & \xrightarrow{U_1' \odot U_2} \\
U_1; U_2 & \xrightarrow{U_1' ; U_2'} \\
U_1 \odot U_2 & \xrightarrow{U_1' \odot U_2'} \\
U_1 \oplus U_2 & \xrightarrow{U_1' \oplus U_2'} \\
U_1 & \xrightarrow{U_1'} \\
U_2 & \xrightarrow{U_2'} \\
U_1; U_2 & \xrightarrow{U_1' ; U_2'} \\
U_1 \odot U_2 & \xrightarrow{U_1' \odot U_2'} \\
U_1 \oplus U_2 & \xrightarrow{U_1' \oplus U_2'} \\
\end{align*}
\]

**Fig. 5** Relation \( U \leftarrow U \)

by \( U \) is defined as follows:

\( U := 0 \mid \ell | \alpha | U_1 \odot U_2 \mid U_1; U_2 \mid U_1 \odot U_2 \mid \mu \alpha.U \mid \circ U \mid \bullet U \mid L \mid \sim L \)

We assume that the unary usage constructors \( \circ \) and \( \bullet \) bind tighter than the binary constructors, i.e., \( ; \) and \( \otimes \).

The meaning of each usage constructor is basically the same to [11]. The usage \( 0 \) means that no layer transition rule can be applied at all. The usage \( \ell \) means that the layer transition rule \( \ell \) is applied if \( \ell = L \). \( \alpha \) is the usage variable bound in the form of \( \mu \alpha.U \), which denotes a recursive usage that satisfies \( \alpha = U \).

\( U_1; U_2 \) means that layer transition rules are applied first according to \( U_1 \) and then according to \( U_2 \). \( U_1 \odot U_2 \) means that layer transition rules are applied according to an order obtained by interleaving \( U_1 \) and \( U_2 \). \( U_1 \odot U_2 \) means that layer transition rules are applied according to either \( U_1 \) or \( U_2 \). \( \circ U \) means that some of the applications of layer transition rules expressed by \( U \) may be delayed, and \( \bullet U \) means that the applications represented by \( U \) must occur now. For example, \( \bullet \circ U_1; U_2 \) is equivalent to \( U_1; U_2 \).

The last two constructors \( L \) and \( \sim L \) mean that the layer \( L \) is active and not, respectively. For example, the usage \( l_k; \text{Dead} \) means that the resource is accessed according to \( l_k \) and after that the layer \( \text{Dead} \) is active. It is valid since \( l_k \) just activates \( \text{Dead} \). We say that a usage \( U \) is valid if no layer name appear in \( U \).

The subusage relation \( U_1 \leq U_2 \), which means that \( U_1 \) is more general than \( U_2 \), is defined via the transition relation \( U_1 \xrightarrow{t} U_2 \), which means that the applications represented by \( U_1 \) first applies \( t \) and then applies rules according to \( U_2 \).

Formally, \( U_1 \xrightarrow{t} U_2 \) is the least relation closed under the rules in **Fig. 5**. \( U_1 \leq U_2 \) is the least relation that satisfies the rules in **Fig. 4**, where \( U_1 \equiv U_2 \) means \( U_1 \leq U_2 \) and \( U_2 \leq U_1 \). The last four rules in **Fig. 4** defines how a usage \( U \) is translated into the valid one. Suppose \( A(l_k) = \text{activate}(\text{Dead}) \) and \( A(l_s) = \text{deactivate}(\text{Large}) \). Then \( l_k; l_s; \text{Dead} \leq l_k; \text{Dead}; l_s \leq l_k; l_s \) and \( l_s; \text{Large} \leq l_s \).

We define the multi-step transition relation \( U_1 \xrightarrow{t} U_2 \), where \( t \in L^1* \), as follows:

\[
\begin{align*}
\xrightarrow{t} & \leq \begin{cases} 
\xrightarrow{U_1} & \text{if } t = \ell \\
\xrightarrow{U_1' \circ U_2'} & \text{if } t = \ell' 
\end{cases} 
\end{align*}
\]

By using the labeled transition system, we define the trace set represented by a usage as follows. Let \( U \) be a usage. \( [U] \) denotes the following set:

\[
\{ t \mid [U],(U \xrightarrow{t} U')) \cup \{ t \mid [U'],(U \xrightarrow{t} 0) \}
\]

A usage context, written \( C \), is an expression obtained from a usage by replacing one occurrence of a free usage variable with \([ \] \). We write \( C[U] \) for the usage obtained by replacing \([ \] \) with \( U \) in \( C \) if the set of free usage variables in \( U \) are disjoint from the set of bound usage variables in \( C \).

The subusage relation \( U_1 \leq U_2 \) is the largest relation that satisfies the following conditions:

1. \( C[U_1] \leq C[U_2] \) for any usage context \( C \)
2. If \( U_2 \xrightarrow{t} U_2' \), then \( U_1 \xrightarrow{t} U_1' \) and \( U_1' \leq U_2' \) for some \( U_1' \)
3. If \( U_2 \xrightarrow{t} 0 \), then \( U_1 \xrightarrow{t} 0 \)

### 3.4 Type and Type Judgement

The type \( \tau \) in \( X^{\text{cop}} \) is defined as follows:

\[
\tau := \text{bool} \mid (\tau_1 \Rightarrow \tau_2, U) \mid (R, U)
\]

\[
\begin{align*}
\text{bool} \leq \text{bool} & \quad (\tau_1 \Rightarrow \tau_2, U) \leq (\tau_1 \Rightarrow \tau_2, U') \\
R \leq U & \quad (R, U) \leq (R, U')
\end{align*}
\]

**Fig. 6** Subtype relation rules \( \tau \leq \tau' \)


**bool** is the type of boolean values. \((\tau_1 \rightarrow \tau_2, U)\) is the type of functions that take a value of type \(\tau_1\) as an argument and return a value of type \(\tau_2\). \(U\) describes how the function is used. \((R, U)\) is the type of resources that are accessed according to \(U\). We write \(Use(\tau)\) for the outermost usage of \(\tau\); i.e., \(Use(\text{bool}) = 0, Use(\tau_1 \rightarrow \tau_2, U) = U\) and \(Use(R, U) = U\).

The subtype relation \(\tau \leq \tau'\) is the least relation closed under the rules in \(\Box\) 6. Our rules ignore for simplicity covariance and contravariance on functions’ return and argument types, respectively.

The type judgement \(\Gamma \vdash M : \tau\), read “term \(M\) is given type \(\tau\) under the type environment \(\Gamma\)”, is the least relation closed under the rules in \(\Box\) 7, where \(\Gamma\) is a finite mapping from variables to types. The intuitive meaning is that the term \(M\) can be evaluated to a value of type \(\tau\) if the evaluation terminates, and each free variable \(x\) in \(M\) is used according to type \(\Gamma(x)\) during the evaluation. We write \(\emptyset\) for the empty type environment, and \(\Gamma, x : \tau\) if \(x \notin dom(\Gamma)\) for the type environment \(\Delta\) such that \(dom(\Delta) = dom(\Gamma) \cup \{x\}\), \(\Delta(x) = \tau\) and \(\Delta(y) = \Gamma(y)\) for \(y \in dom(\Gamma)\).

The binary operations ‘;’ and ‘\&’ on types and type environments are defined as follows. Let \(\text{op}\) be a binary usage constructor ‘;’ or ‘\&’. Then \(\tau_1 \text{op} \tau_2\) is defined as follows:

\[
\begin{align*}
\text{bool} \quad \text{op} \quad \text{bool} & = \quad \text{bool} \\
(\tau_1 \rightarrow \tau_2, U_1) \quad \text{op} \quad (\tau_1 \rightarrow \tau_2, U_2) & = (\tau_1 \rightarrow \tau_2, U_1 \text{op} U_2) \\
(R, U_1) \quad \text{op} \quad (R, U_2) & = (R, U_1 \text{op} U_2)
\end{align*}
\]

Let \(\Gamma_1\) and \(\Gamma_2\) be type environments with the same domain, i.e., \(dom(\Gamma_1) = dom(\Gamma_2)\). Then we define \(\Gamma_1 \text{op} \Gamma_2\) as follows:

\[
\begin{align*}
dom(\Gamma_1 \text{op} \Gamma_2) & = \quad dom(\Gamma_1) \\
(\Gamma_1 \text{op} \Gamma_2)(x) & = \quad \Gamma_1(x) \text{op} \Gamma_2(x)
\end{align*}
\]

The type environment \(\bullet, \Gamma\) is defined as follows:

\[
\bullet, \Gamma = \begin{cases} 
\Gamma & \text{if } x \notin dom(\Gamma) \\
\Gamma', x : (R, \bullet U) & \text{if } \Gamma = \Gamma', x : (R, U)
\end{cases}
\]

The first nine rules in \(\Box\) 7 are almost the same to [11]. T-New ensures that if a term is typeable then all of the objects created in the term can be typed by using only valid usages. T-Ev is equivalent to the typing rule for accessing resources. \(\Delta_{\text{fun}}^{(U_1, U_2, R)}\) in T-Fun is defined as follows:

\[
\Delta_{\text{fun}}^{(U_1, U_2, R)} = \begin{cases} 
\emptyset & \text{if } 1 \notin [U_1] \\
\Gamma & \text{if } (1 \in [U_1] \subseteq \{1, 1 \downarrow\}) \land (1 \notin [U_2]) \\
\mu.0 \oplus (\Gamma \otimes \alpha) & \text{otherwise}
\end{cases}
\]

The last three rules T-CaseL, T-CaseNotL and T-Case are the typing rules for the terms of the form \(\text{case } x \text{ of } (L, M_1, M_2)\). If the term is typed by using T-CaseL, \(L\) is always active on the object bound to \(x\) when it is reduced. If T-CaseNotL is used, \(L\) is always inactive on the object bound to \(x\) when it is reduced. T-Case is used otherwise.

### 3.5 Type Soundness

The following theorem states that our type system is sound. We say that \(M\) is well-annotated if all the annotations on escape information \(\cdot^e\) are sound, i.e., \(\langle\rangle, M\rangle\) is never reduced to \((H, \cdot^e]\) such that \(x \in FV(v)\).

**Theorem 1 (Type Soundness)** Suppose \(M\) is well-annotated. If \(0 \vdash M : \tau\) and \(Use(U) \leq 0\), then all the following properties hold:

\(1\) \(\langle\rangle, M\rangle \rightarrow^* \text{Error}\).

\(2\) If \(\langle\rangle, M\rangle \rightarrow^* (H, v)\), then \(\forall x \in dom(H), \downarrow \in H(x)\).

The condition \(Use(\tau) \leq 0\) states that even if the term \(M\) is evaluated to a resource, the resource may not be accessed after the evaluation. Property 2 states that \(M\) never performs an illegal resource access. Property 2 means that all the resources are used up when the evaluation terminates.

**Proof 1** The proof of the theorem is similar to [11], i.e., using another operational semantics of our language that takes into account not only how but also where in the expression a layer transition rule is applied to each object during evaluation. The dynamic expressions, ranged over by \(D\), in [11] are extended as follows:

\[
D ::= \ldots \mid \text{ev}(D, l) \mid \text{case } x \text{ of } (L, D_1, D_2)
\]

The reduction rules for \(\text{ev}(D, l)\) and \(\text{case } x \text{ of } (L, D_1, D_2)\) are defined similarly to the ones for \(\text{acc'}(D)\) and if \(D_1\) then \(D_2\) else \(D_3\), respectively. 

### 4. Related Work

Typestate checking [1,3,7,8] can be seen as a closely related topic from the viewpoint of estimating statically the states of an object. A typestate is a static state
of an object, and changes when a specific operation on that object is performed. State changes in COP languages are more complicated because operations on other objects can also change the state.

Studies on optimizing trace-based aspects \cite{4,14} are also closely related to analyze the possible orders of layer activation/deactivation, as they check whether a state can be achieved or not when a statement or method is executed. The states of an object are modeled to change in response to the operations on other related objects as well as the object itself, which is similar to our case. Although these approaches could be more powerful than ours if applicable, it is not clear how the two problems relate to each other.

5. Conclusions and Future Work

We have proposed an activation order analysis in a context-oriented programming language similar to EventCJ and formalized it by developing \( \text{COP} \), a call-by-value lambda calculus with objects, layers and layer activation/deactivation, on top of \( \text{L}^R \) proposed by Igarashi and Kobayashi \cite{11}.

Our future work includes lifting our analysis to an object-oriented language so that we can implement the analysis in our EventCJ compiler.

References

\[ c = \text{true or false} \]
\[ 0 \vdash c : \text{bool} \quad \text{(T-Const)} \]
\[ x : \tau \vdash x : \tau \quad \text{(T-VAR)} \]
\[ [U] \subseteq \Phi \quad U \text{ is valid} \quad \text{(T-New)} \]
\[ 0 \vdash \text{new}^\delta() : (R, U) \quad \text{(T-EV)} \]
\[ \Gamma \vdash M : (R, l) \quad \text{(T-Ev)} \]
\[ \Gamma \vdash \text{ev}(M, l) : \text{bool} \]
\[ \Gamma, f : (τ_1 → τ_2, U_2), x : τ_1 \vdash M : τ_2 \quad \text{(T-FUN)} \]
\[ Δ^{\text{fun}}(U_1, U_2, \circ_γ) \vdash \text{fun}(f, x, M) : (τ_1 → τ_2, U_1) \quad \text{(T-App)} \]
\[ \Gamma_1 \vdash M_1 : (τ_2 → τ_1, 1) \quad \Gamma_2 \vdash M_2 : τ_2 \]
\[ \Gamma_1 ; \Gamma_2 \vdash M_1 M_2 : τ_1 \quad \text{(T-Now)} \]
\[ * \cdot \Gamma \vdash M^x : \tau \]
\[ \Gamma_1 \vdash M_1 : \text{bool} \quad \text{(T-IF)} \]
\[ \Gamma \vdash M_2 : \tau \quad \Gamma_3 \vdash M_3 : \tau \]
\[ \Gamma_1 ; \Gamma_2 \vdash \text{if } M_1 \text{ then } M_2 \text{ else } M_3 \quad \text{(T-Weak)} \]
\[ \Gamma' \vdash M : \tau' \quad \Gamma \leq \Gamma' \quad \tau' \leq \tau \]
\[ \Gamma \vdash M : \tau \quad \text{(T-CASEL)} \]
\[ \Gamma_0 \vdash x : (R, L) \quad \Gamma_1 \vdash M_1 : \tau \]
\[ \Gamma_0 ; \Gamma_1 \vdash \text{case } x \text{ of } (L, M_1, M_2) : \tau \quad \text{(T-CASENOTL)} \]
\[ \Gamma_0 \vdash x : (R, \neg L) \quad \Gamma_2 \vdash M_2 : \tau \]
\[ \Gamma_0 ; \Gamma_2 \vdash \text{case } x \text{ of } (L, M_1, M_2) : \tau \]
\[ \Gamma_0 \vdash x : (R, 0) \quad \Gamma_0 \vdash x : (R, 0) \quad \text{(T-CASE)} \]
\[ \Gamma_1 \vdash M_1 : \tau \quad \Gamma_2 \vdash M_2 : \tau \]
\[ \Gamma_0 ; (\Gamma_1 \oplus \Gamma_2) \vdash \text{case } x \text{ of } (L, M_1, M_2) : \tau \]

Fig. 7 Typing rules