On-Demand Layer Activation for Type-Safe Deactivation

Tetsuo Kamina
Ritsumeikan University
kamina@acm.org

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

Atsushi Igarashi
Kyoto University
igarashi@kuis.kyoto-u.ac.jp

ABSTRACT

Dynamic layer deactivation in context-oriented programming (COP) allows a layer to be dynamically disabled in the running application in a disciplined way. Deactivating a layer may lead to an error if there is another layer which has been activated but depends on the deactivated layer in the sense that the latter calls a method that exists only in the former. A type system or static analysis might be able to check the absence of such depending layers at each deactivation point but it would not be very easy, especially in the open-world setting.

We argue that the *on-demand activation*, which implicitly activates all layers on which currently activated layer depends, addresses this problem. In this mechanism, the precedent layer deactivation is canceled when the depending layer requires the implementation of the deactivated layer. This means that this mechanism can ensure that all method calls succeed without performing the checks of absent depending layers, which simplifies the type system. We formalize this idea as an extension of ContextFJ, a COP extension of Featherweight Java, and prove its type soundness.

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

Dynamic layer deactivation, ContextFJ, Type soundness

COP'14 July 29, 2014, Uppsala, Sweden
ACM 978-1-4503-2861-6 ...$15.00.
http://dx.doi.org/10.1145/2637066.2637070

1. INTRODUCTION

Context-Oriented Programming (COP) is an approach to improve modularity of variations of behavior that depend on contexts [9]. A number of COP languages provide linguistic constructs that modularize such variations using *layers*, and that activate/deactivate them according to the executing contexts [2, 3, 6, 14]. A layer defines a number of *partial methods*. A partial method is a method that can run before, after, or around a (partial) method with the same name and signature defined in a different layer or a class. Thus, it provides the specific behavior of the system only when the layer is active.

In this paper, we consider a language mechanism in COP, *method introduction by layers*, i.e., allowing a layer to declare partial methods that introduce new behavior to existing classes. This mechanism makes the type system interesting, and actually there are a number of cases where such mechanism is useful in particular when we can describe dependency between layers using, e.g., the *requires* relations [11]. However, this mechanism makes the type system complex when we consider dynamic layer deactivation, which disables the layer dynamically. If a layer can introduce new methods, deactivating a layer may lead to an error if there is another layer that has been activated but depends on the deactivated layer in the sense that the latter calls a method that exists only in the former. We might develop a type system that can check the absence of such depending layers at each deactivation point. However, it would not be very easy, especially in the open-world setting. In fact, although a number of COP calculi have been developed thus far [4, 10, 1, 11, 15], none of them combines the method introduction by layers with dynamic layer deactivation.

In this paper, we argue that *on-demand activation*, which implicitly activates all layers on which currently activated layer depends, addresses this problem. This mechanism is formerly known in [5, 16], but is not discussed in the setting of method introduction by layers. We show that this mechanism simplifies the type system because it can ensure that all method calls succeed without performing the checks of absent depending layers. The idea is that, instead of activating the depended layers when the depending layer becomes active, our mechanism postpones the activation of depended layers until when each method call occurs. In this sense, our mechanism is different from those proposed in [5, 16]. We formalize this idea as an extension of ContextFJ [11], a COP extension of Featherweight Java [12]. Our calculus, ContextFJ\, includes layer deactivation (i.e., `without`), which is not included in ContextFJ, and provides dynamic semantics

and a type system modified from ContextFJ according to the above idea. We prove its type soundness.

The rest of this paper is organized as follows. Section 2 reviews the language mechanisms for COP, in particular the method introduction by layers and dynamic layer deactivation, and argues that how on-demand activation addresses the aforementioned problem. Section 3 introduces the syntax, operational semantics, and type system of ContextFJ\. Section 4 concludes this paper and discusses related work.

2. TYPE-SAFE LAYER DEACTIVATION

2.1 Reviewing COP Mechanisms

We show the motivation for combining dynamic layer deactivation and method introduction by layers by using the telecom simulation example. This example includes classes `Customer` and `Connection` to represent customers and phone calls between them, respectively.

```
class Customer { .. }
class Connection {
  Connection(Customer a, Customer b) { .. }
  void complete() { .. }
  void drop() { .. }
}
```

The usage of these classes is demonstrated as follows:

```
Connection simulate() {
  Customer tetsuo = .., tomoyuki = ..;
  Connection c =
    new Connection(tetsuo, tomoyuki);
                       // Tetsuo calls Tomoyuki
  c.complete();        // Tomoyuki accepts
  c.drop();            // Tomoyuki hangs up
  return c;
}
```

Then, we consider two additional features, measuring the duration of phone calls, and calculating and charging the cost of them, which are dynamically composed with the system. In COP, such dynamically composed features are implemented using layers. The following `Timer` layer implements the former feature:

```
layer Timing {
  class Connection {
    Timer timer;
    void complete() { proceed(); timer.start(); }
    void drop() { timer.stop(); proceed(); }
    int getTime() { return timer.getTime(); }
  }
}
```

Two *partial methods*, `complete` and `drop`, override the original methods when `Timing` is *active* (as explained below). This layer also introduces a method, `getTime`, and a field, `timer`. The `proceed()` calls delegate behavior to overridden methods.

Layers can dynamically be composed with the system by using layer activation. The following `ensure` construct [11] is provided for this purpose.

```
ensure Timing {
  Connection c = simulate();
  System.out.println(c.getTime());
}
```

```
without Timing { // free talk
  Connection c = Connection(tetsuo, naoko);
  c.complete();
  ensure Billing {
      // checks the amount of charge for
      // the past charged calls
    System.out.println(c.getAmount());
  }
  c.drop();
}
```

Figure 1: Activating `Billing` within the activation of `Timing`.

The layer activation is effective for the *dynamic extent* of the execution of the `ensure` block. Thus, during the execution of the `simulate` method, the `Timing` layer is active; when `complete` and `drop` are called, the corresponding partial methods defined in `Timing` that start and stop the timer respectively are called. Note that we can call `getTime`, which is introduced in `Timing`, within the `ensure` block.

The latter feature is implemented by the layer `Billing`, which is defined as follows:

```
layer Billing requires Timing {
  class Connection {
    int amount = 0;
    int getAmount() {
      return amount;
    }
    void charge() {
      int cost = getTime(); amount += cost;
      .. charge the cost on the caller .. }
    void drop() { proceed(); charge(); }
  }
}
```

This layer overrides the `drop` method and introduces two methods, `getAmount` and `charge`, which calculate the cost of phone calls and charge that cost, respectively. The cost is calculated based on the duration of phone calls. This means that this layer assumes that `Timing` is active when the partial methods defined in this layer is called. This assumption is denoted by the `requires` clause in the first line of the layer declaration.

In ContextFJ [11], to activate `Billing`, we need to activate `Timing` before, which means that `Billing` can be activated only within the `ensure` block that activates `Timing`.

```
ensure Timing {
  ensure Billing { simulate(); }
}
```

Within `ensure Billing`, both `Timing` and `Billing` are active, and the partial methods in `Billing` override the ones in `Timing` because `Billing` is the most recently activated layer. Thus, `drop` called in `simulate` charges the cost of the phone call on the caller.

2.2 Problem Statements

Method introduction by layers interacts badly with dynamic layer deactivation. The piece of code in Figure 1 representing a free phone call illustrates this problem. Within

`without Timing`, which deactivates `Timing` so as to make the phone call free, `Billing` is activated just to check the total amount of charges for the past calls. Of course, this check of charges does not require any `Timing` functions; the `getAmount` method just returns the current amount of charges that are accumulated during the past phone calls. There are two problems in this code. First, assuming that ContextFJ supports `without`, this code is rejected by the compiler just because the activation of `Billing` is not enclosed within `Timing`. ContextFJ forces us to activate `Timing` whenever we want to activate `Billing`. This enforcement is applied even when we use the feature of `Billing` that does not depend on `Timing`.

Second, actually ContextFJ does not support `without`. If layers can dynamically be deactivated, the invocation of a method introduced by the deactivated layer results in a failure when the layer depending on the deactivated layer calls that method. To statically check such an error, we need to gather information about "which layer is absent" at each deactivation point, which is not be very easy, especially in the open-world setting.

2.3 On-Demand Activation

We argue that *on-demand activation* addresses the aforementioned problems. It implicitly activates layers on which currently activated layer depends. To represent this mechanism, we propose the `activates` clause that specifies the layers that are implicitly activated when the declared layer is used.

```
layer Billing activates Timing {
  /* The body is the same as above */
}
```

We can activate `Billing` anywhere, regardless of the condition whether this activation is enclosed with the activation of `Timing`. For example, the activation of `Billing` within `without Timing` shown in Figure 1 is now allowed.

On-demand activation also simplifies programs when we require both `Billing` and `Timing`. For example, when we call the `simulate` method with the feature of charging, we enclose this method call just within `ensure Billing` instead of activating `Timing` explicitly:

```
ensure Billing { simulate(); }
```

The layer specified by the `activates` clause (`Timing`) becomes active just before the partial method defined in `Billing` is called, and is deactivated after that call. Thus, the above piece of code safely executes the feature of charging of the phone call as in the case where we explicitly enclose this piece of code within `ensure Timing`.

Note that the activation of `Timing` is *not* performed when `Billing` is activated but postponed until when each method call occurs. If the activation of `Timing` is performed when `Billing` is activated by `ensure`, the following code

```
ensure Billing { without Timing { c.charge(); }}
```

would result in an error because `charge` calls `getTime`, which is introduced by `Timing` but it is deactivated within the context of the call of `charge`. Thus, it is necessary to activate `Timing` at each method call. In this sense, our mechanism is different from those proposed in [5, 16].

We can even call the method introduced by `Timing` within the `ensure Billing` block as follows.

```
ensure Billing {
  Connection c = ..;
  c.complete();
  c.drop();
  System.out.println(c.getTime());
}
```

This piece of code activates `Billing` and calls `getTime` introduced by `Timing`. This is allowed because we know that within `ensure Billing`, `Timing` will be active around each method call.

This mechanism makes the type system as simple as that of ContextFJ. It is not necessary to gather the information about the absent layers, because the on-demand activation ensures that a call of method introduced by the layer specified by `activates` always succeed. We further discuss the type system in Section 3.

One may wonder if on-demand activation makes it difficult to reason about the fact that the body of layer specified by `without` is never executed within the dynamic extent of the `without` block. For example, we do not want to execute any "timing" functions during the free talk (Figure 1). Actually, the same problem exists in the existing COP languages supporting `without`, because they also allow us to activate a layer within the `without` block that deactivate that layer. To address this problem, we can apply other static analyses such as model checking to validate such a requirement.

One may also wonder if, instead of introducing `activates`, we could wrap the body of a partial method that requires another layer in an `ensure` statement to activate that layer. For example, at first, enclosing the bodies of `charge` and `drop` belonging to `Billing` by `ensure Timing {..}` seems to work. Actually, this workaround does not work in ContextFJ (as well as many of COP languages), where the behavior of `proceed` calls is fixed when the method is invoked and does not change by surrounding layer (de)activation statements. Thus, `proceed` in `Billing` may not proceed to partial methods in `Timing`.

2.3.1 Activation order

If multiple layers are active, there may be multiple partial methods with the same name and signature and thus the partial method lookup should be performed in a well-defined order. Most of COP languages take the strategy that the most recently activated layer has the highest priority.

When we consider the on-demand activation, we also need to define the order of layers that are activated implicitly. For example, the method calls within `ensure Billing { ..}` activate `Timing` if it is not active before the execution of `ensure` block. Since the body of `Billing` assumes that `Timing` is already active, it is necessary that `Billing` has a higher priority than `Timing`. Similarly, if `Timing` activates another layer, that layer should have the lowest priority, followed by `Timing` and `Billing`.

We also have to consider the situation where a layer activates multiple layers as follows.

```
layer A activates L1,..,Ln { .. }
```

In this case, the layers specified by the `activates` clause become active in the order specified by the programmer: $\mathtt{L}_1, \mathtt{L}_2, \cdots, \mathtt{L}_n$. Furthermore, there is a set of layers that $\mathtt{L}_1$ activates, another set of layers that each element of that set activates, and so on; i.e., we need to obtain a transitive closure Λ_1 of the `activates` relation for $\mathtt{L}_1$. Similarly, there

are transitive closures $\Lambda_2, \cdots, \Lambda_n$ of the `activates` relation for $\mathtt{L}_2, \cdots, \mathtt{L}_n$, respectively. We need to carefully consider the order of those active layers. For example, we cannot put the layers Λ_2 after $\mathtt{L}_1$ if $\mathtt{L}_1$ activates some elements in Λ_2.

In general, the ordering of active layers is determined as follows.

1. Insert layers specified by the `activates` clause of each activated layer into the tail of the list of activated layers (the head of that list has the highest priority).

2. Repeat 1 for each newly activated layer until when there are no layers declaring `activates`.

This rule ensures that the layers specified by `activates` always have lower priority than that of the layer declaring `activates`. Thus the call of method introduced by the layer declared in `activates` never fails[1].

Note that we also need to remove the duplicated layers from the list of active layers. To avoid duplicate calls of the same partial method in one single method call, most COP languages disallow the same layer to be active twice at the same time. However, the above rule does not eliminate such duplicate layers. For example, if the layer `A` activates layers `C` and `B`, and the layer `B` activates the layer `C`, `ensure A { .. }` results in the list of active layers: `C,B,C,A`. The duplicated layer `C` should be removed before the execution of the method body starts.

Again, we need to be careful to remove such duplicated layers. Removing the layer activated by other layer may result in failure when the activating layer calls a method introduced by the activated layer. For example, if we remove `C` at the left-hand side of `B` in the above list of active layers, the call of partial method declared in `B` that uses methods introduced by `C` may result in failure. Thus, to finalize the creation of the list of active layers, we need to remove duplicate layers from that list according to the rule described as follows:

> If the same layer is activated twice or more in the same list of active layers, we remove those layers other than the one that has the lowest priority from that list.

2.3.2 On-demand activation vs requires relation

Instead of the `requires` construct in [11] where the requiring layer assumes that the required layers are already active, the `activates` construct activates all the "required" layers one after another. This mechanism can eliminate unnecessary and tedious nesting of `ensure` blocks, and easily enables type checking of layer activation for COP languages with the method introduced by layers and `without`.

We do not argue that `requires` should be replaced with `activates`, however. The `requires` construct would exert its usefulness on requiring the *interface* of layers (although the current version of `requires` in [11] requires the *implementation* of the specified layers.) In `requires`, we may assume that the layers providing this interface are active but do not have to concern about the concrete implementations. Likewise, we may write the `requires` clause like "`requires LayerA` or `LayerB`." The `activates` construct is not suitable for specifying such interfaces, because the resulting activation (of the implementation) would be ambiguous. Thus, we consider that `requires` and `activates` are complementary.

[1]We assume that there are no cycles in the `activates` relation.

```
CL    ::=  class C ◁ C { C̄ f̄; K M̄ }                    (classes)
K     ::=                                              (constructors)
           C(C̄ f̄){ super(f̄); this.f̄ = f̄; }
M     ::=  C m(C̄ x̄){ return e; }                        (methods)
e, d  ::=  x | e.f | e.m(ē) | new C(ē)                 (expressions)
         | ensure L e | without L e
         | proceed(ē) | super.m(ē)
         | new C(v̄)<C,L̄,L̄>.m(ē)
v, w  ::=  new C(v̄)                                    (values)
```

Figure 2: ContextFJ\: abstract syntax

3. FORMALIZATION

We formalize the aforementioned idea as a core calculus ContextFJ\. Due to the limited space, we only present key rules throughout this section. Omitted rules are identical to those in [11].

3.1 Syntax

Let metavariables `C`, `D`, `E`, and `F` range over class names; `L` over layer names; `f` and `g` over field names; `m` over method names; and `x` and `y` over variables, which contain a special variable `this`. The abstract syntax of ContextFJ\ is shown in Figure 2. As in FJ, overlines are used to denote sequences: i.e., $\overline{\mathtt{f}}$ stands for a possibly empty sequence $\mathtt{f}_1, \cdots, \mathtt{f}_n$ and similarly for $\overline{\mathtt{C}}$, $\overline{\mathtt{x}}$, $\overline{\mathtt{e}}$, and so on. Layers in a sequence are separated by semicolon. The empty sequence is denoted by $\bullet$. We write "$\overline{\mathtt{C}}\ \overline{\mathtt{f}}$" for "$\mathtt{C}_1\ \mathtt{f}_1, \cdots, \mathtt{C}_n\ \mathtt{f}_n$," where n is the length of $\overline{\mathtt{C}}$ and $\overline{\mathtt{f}}$, and similarly "$\overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}$" as shorthand for the sequence of declarations "$\mathtt{C}_1\mathtt{;} \cdots \mathtt{C}_n\ \mathtt{f}_n\mathtt{;}$," "$\mathtt{this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}\mathtt{;}$" for "$\mathtt{this.f}_1\mathtt{=f}_1\mathtt{;} \cdots \mathtt{;this.f}_n\mathtt{=}_n\mathtt{;}$," and "$\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{e}}$" for "$\mathtt{f}_1\mathtt{=e}_1, \cdots, \mathtt{f}_n\mathtt{=e}_n$." 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.

Class declarations, constructors, methods are the same as those of FJ (and thus as those of ContextFJ). A class declaration `CL` consists of its name, its superclass name, field declarations $\overline{\mathtt{C}}\ \overline{\mathtt{f}}$, a constructor `K`, and method definitions $\overline{\mathtt{M}}$. A constructor `K` is a trivial one that takes initial values of all fields and sets them to the corresponding fields. A method `M` takes $\overline{\mathtt{x}}$ as arguments and returns an expression `e` (and thus it is a functional calculus).

An expression `e` can be a variable, field access, method invocation, object instantiation, layer activation/deactivation, and `proceed`/`super` call. It can also be special run-time expressions that are not supposed to appear in classes like `new C(`$\overline{\mathtt{v}}$`)<C,`$\overline{\mathtt{L}}$`,`$\overline{\mathtt{L}}$`>.m(`$\overline{\mathtt{e}}$`)`. The expression `new C(`$\overline{\mathtt{v}}$`)<C,`$\overline{\mathtt{L}}'$`,`$\overline{\mathtt{L}}$`>.m(`$\overline{\mathtt{e}}$`)`, where $\overline{\mathtt{L}}'$ is assumed to be a prefix of $\overline{\mathtt{L}}$, basically means that `m` is going to be invoked on `new C(`$\overline{\mathtt{v}}$`)`. The annotation `<C,`$\overline{\mathtt{L}}'$`,`$\overline{\mathtt{L}}$`>` indicates where method lookup should start, and is used to give a semantics of `super` and `proceed` by simple substitution-based reduction.

Unlike the existing COP languages, the calculus does not provide syntax for layers. Partial methods are registered in a partial method table. Let $\mathcal{R}$ be a binary relation on layer names; $(\mathtt{L}_1, \mathtt{L}_2) \in \mathcal{R}$ intuitively means that layer $\mathtt{L}_1$

$$\frac{\mathit{filter}(\mathit{fix}(\overline{\mathtt{L}})) = \overline{\mathtt{L}}' \quad \overline{\mathtt{L}}' \vdash \mathtt{new\ C(\overline{v})\langle C,\overline{L}',\overline{L}'\rangle.m(\overline{w})} \longrightarrow \mathtt{e}'}{\overline{\mathtt{L}} \vdash \mathtt{new\ C(\overline{v}).m(\overline{w})} \longrightarrow \mathtt{e}'} \quad \text{(R-Invk)}$$

$$\frac{\mathit{mbody}(\mathtt{m},\mathtt{C}',\overline{\mathtt{L}}'',\overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e\ in\ C}'', (\overline{\mathtt{L}}''';\mathtt{L}_0) \quad \mathtt{class\ C}'' \lhd \mathtt{D\{...\}}}{\overline{\mathtt{L}} \vdash \mathtt{new\ C(\overline{v})\langle C',\overline{L}'',\overline{L}'\rangle.m(\overline{w})} \longrightarrow \left[\begin{array}{ll} \mathtt{new\ C(\overline{v})} & /\mathtt{this}, \\ \overline{\mathtt{w}} & /\overline{\mathtt{x}}, \\ \mathtt{new\ C(\overline{v})\langle C'',\overline{L}''',\overline{L}'\rangle.m} & /\mathtt{proceed}, \\ \mathtt{new\ C(\overline{v})\langle D,\overline{L}',\overline{L}'\rangle} & /\mathtt{super} \end{array}\right] \mathtt{e}} \quad \text{(R-InvkP)}$$

$$\frac{\mathit{ensure}(\mathtt{L},\overline{\mathtt{L}}) = \overline{\mathtt{L}}' \quad \overline{\mathtt{L}}' \vdash \mathtt{e} \longrightarrow \mathtt{e}'}{\overline{\mathtt{L}} \vdash \mathtt{ensure\ L\ e} \longrightarrow \mathtt{ensure\ L\ e}'} \quad \text{(R-Ensure)}$$

$$\frac{\mathit{remove}(\mathtt{L},\overline{\mathtt{L}}) = \overline{\mathtt{L}}' \quad \overline{\mathtt{L}}' \vdash \mathtt{e} \longrightarrow \mathtt{e}'}{\overline{\mathtt{L}} \vdash \mathtt{without\ L\ e} \longrightarrow \mathtt{without\ L\ e}'} \quad \text{(R-Without)}$$

Figure 3: ContextFJ\: computation (selected)

`activates` $\mathtt{L}_2$. In the following sections, we assume a fixed dependency relation between layers and write $\mathtt{L\ act}\ \Lambda$, read "layer L activates layers Λ," when $\Lambda = \{\mathtt{L}' | (\mathtt{L},\mathtt{L}') \in \mathcal{R}\}$.

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

1. $CT(\mathtt{C}) = \mathtt{class\ C\ ...}$ 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 $\lhd$ clauses.
5. There are no cycles in the transitive closure of the act clauses.
6. $PT(\mathtt{m},\mathtt{C},\mathtt{L}) = \mathtt{...\ \ m(...)\{...\}}$ for any $(\mathtt{m},\mathtt{C},\mathtt{L}) \in \mathit{dom}(PT)$.

3.2 Operational semantics

The operational semantics is given by a reduction relation of the form $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$, read "expression e reduces to $\mathtt{e}'$ under the activated layers $\overline{\mathtt{L}}$." The reduction rules are shown in Figure 3. We only show the rules for method invocation, partial method invocation, `ensure` and `without`. For other rules, the reader can consult the ContextFJ paper [11].

The rule R-Invk is for method invocation. It initializes the cursor of the method lookup to be at the receiver's class and the currently activated layers. Note that the activated layers $\overline{\mathtt{L}}'$ is updated in the hypothesis part. The auxiliary function *fix* is defined as follows. We write $\overline{\mathtt{L}}\ \mathtt{act}\ \Lambda$ to denote Λ as shorthand for the set union "$\Lambda_1 \cup \Lambda_2 \cup \cdots \cup \Lambda_n$" where $\mathtt{L}_1\ \mathtt{act}\ \Lambda_1$, $\mathtt{L}_2\ \mathtt{act}\ \Lambda_2$, $\cdots$, and $\mathtt{L}_n\ \mathtt{act}\ \Lambda_n$. We apply set operators (such as $\setminus$) to sequences by regarding the operand sequence as a set. We assume that the elements in Λ are ordered as specified by the programmer when Λ is used as an argument to *fix*.

$$\frac{\overline{\mathtt{L}}\ \mathtt{act}\ \Lambda \qquad \mathit{fix}(\Lambda); \overline{\mathtt{L}} = \overline{\mathtt{L}}'}{\mathit{fix}(\overline{\mathtt{L}}) = \overline{\mathtt{L}}'} \qquad \frac{\overline{\mathtt{L}}\ \mathtt{act}\ \emptyset}{\mathit{fix}(\overline{\mathtt{L}}) = \overline{\mathtt{L}}}$$

The auxiliary function *filter* removes duplication of active layers as specified in Section 2.3.1.

The rule R-InvkP deals with the case where the method body is found in layer $\mathtt{L}_0$ in class $\mathtt{C}''$. In this case, `proceed` in the method body is replaced with the invocation of the same method with the cursor pointing to the next layers $\overline{\mathtt{L}}''$. The auxiliary function *mbody*, which is defined in [11], returns the parameters and body $\overline{\mathtt{x}}.\mathtt{e}$ of method m in class $\mathtt{C}'$ when the search starts from $\overline{\mathtt{L}}''$. $\overline{\mathtt{L}}'$ keeps track of the layers that are active when the search initially started. It also returns the information on where the method has been found.

The following two rules relate to layer activation and deactivation. The rule R-Ensure means that e in `ensure L e` should be executing by activating L and all layers in transitive closure of `act` for L. The auxiliary function *ensure* is defined as

$$\begin{array}{lcll} \mathit{ensure}(\mathtt{L},\overline{\mathtt{L}}) & = & \overline{\mathtt{L}} & (\text{if } \mathtt{L} \in \overline{\mathtt{L}}) \\ \mathit{ensure}(\mathtt{L},\overline{\mathtt{L}}) & = & \overline{\mathtt{L}};\mathtt{L} & (\text{otherwise}) \end{array}$$

Similarly, the rule R-Without means that e in `without L e` should be executing under the context where L is absent. The auxiliary function $\mathit{remove}(\mathtt{L},\overline{\mathtt{L}})$ removes L from $\overline{\mathtt{L}}$ (or returns $\overline{\mathtt{L}}$ if L is not in $\overline{\mathtt{L}}$). Once the evaluation of the body of `ensure`/`without` is finished, it returns the value of the body, which is omitted in this paper.

3.3 Type system

In this section, we give a type system of ContextFJ\ with type-safe layer deactivation. Typing rules for classes, methods, and partial methods are identical to those of ContextFJ. Thus, we only discuss expression typing.

A *type environment*, denoted by Γ, is a finite mapping from variables to class names (that are also types). We write $\overline{\mathtt{x}} : \overline{\mathtt{C}}$ for a type environment Γ such that $\mathit{dom}(\Gamma) = \{\overline{\mathtt{x}}\}$ and $\Gamma(\mathtt{x}_i) = \mathtt{C}_i$ for any i. We use $\mathcal{L}$ to stand for a *location*, which is either $\bullet$ (the main expression), C.m (the body of method m in class C in the base layer), or L.C.m (the body of method m in class C in layer L).

The typing rules for expressions are shown in Figure 4. A type judgment for expressions is of the form $\mathcal{L};\Lambda;\Gamma \vdash \mathtt{e} : \mathtt{C}$, read "expression e is given type C under context Γ, location $\mathcal{L}$, and a set of statically-known activated layers Λ." Activated layers Λ are supposed to be a subset of layers actually activated when the expression is evaluated at run time. We only show the typing rules for method invocation, partial method invocation, `ensure` and `without`.

The rule T-Invk is for method invocation. The auxiliary function *mtype* searches the method m in C under the set of active layers Λ', and returns a pair, written $\overline{\mathtt{C}} \to \mathtt{C}_0$, of argument types $\overline{\mathtt{C}}$ and a return type $\mathtt{C}_0$. Note that this rule ensures that all layers in transitive closure of `act` for Λ are active in the arguments for *mtype*. The rule T-Proceed

$\boxed{\mathcal{L}; \overline{\mathtt{L}}; \Gamma \vdash \mathtt{e} : \mathtt{C}}$

$$\frac{\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0, \Lambda', \Lambda') = \overline{\mathtt{D}} \to \mathtt{D}_0 \qquad \mathit{fix}(\Lambda) = \Lambda' \qquad \mathcal{L}; \Lambda; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} \qquad \overline{\mathtt{E}}\mathtt{<:}\overline{\mathtt{D}}}{\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{D}_0} \quad \text{(T-Invk)}$$

$$\frac{\mathtt{L}\ \mathtt{act}\ \Lambda' \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda', \Lambda' \cup \{\mathtt{L}\}) = \overline{\mathtt{D}} \to \mathtt{D}_0 \qquad \mathtt{L.C.m}; \Lambda; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} \qquad \overline{\mathtt{E}}\mathtt{<:}\overline{\mathtt{D}}}{\mathtt{L.C.m}; \Lambda; \Gamma \vdash \mathtt{proceed}(\overline{\mathtt{e}}) : \mathtt{D}_0} \quad \text{(T-Proceed)}$$

$$\frac{\mathcal{L}; \Lambda \cup \{\mathtt{L}\}; \Gamma \vdash \mathtt{e} : \mathtt{C}}{\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{ensure\ L\ e} : \mathtt{C}} \quad \text{(T-Ensure)}$$

$$\frac{\mathcal{L}; \Lambda \setminus \{\mathtt{L}\}; \Gamma \vdash \mathtt{e} : \mathtt{C}}{\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{without\ L\ e} : \mathtt{C}} \quad \text{(T-Without)}$$

Figure 4: ContextFJ\: expression typing (selected)

for partial method invocation is similar. The set Λ of active layers is updated in the rule T-Ensure for `ensure L e`, which requires that the layer `L` is active in the hypothesis part. The typing rule for `without L e` requires that the layer `L` is deactivated in the hypothesis part.

3.4 Properties

The type soundness theorem for ContextFJ\ is stated below. We assume that the typing rule for the program $\vdash (CT, PT, \mathtt{e}) : \mathtt{C}$ is provided as specified in [11].

Theorem 3.1 (Subject Reduction). *Suppose given class and partial method tables are well-formed. If* $\bullet; \overline{\mathtt{L}}; \Gamma \vdash \mathtt{e} : \mathtt{C}$ *and* $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$*, then* $\bullet; \overline{\mathtt{L}}; \Gamma \vdash \mathtt{e}' : \mathtt{D}$ *for some* `D` *such that* $\mathtt{D}\mathtt{<:}\mathtt{C}$.

Theorem 3.2 (Progress). *Suppose given class and partial method tables are well-formed. If* $\bullet; \overline{\mathtt{L}}; \bullet \vdash \mathtt{e} : \mathtt{C}$*, then either* `e` *is a value or* $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$ *for some* $\mathtt{e}'$.

Theorem 3.3 (Type Soundness). *If* $\vdash (CT, PT, \mathtt{e}) : \mathtt{C}$ *and* `e` *reduces to a normal form, then* `e` *is* `new D(`$\overline{\mathtt{v}}$`)` *for some* $\overline{\mathtt{v}}$ *and* `D` *such that* $\mathtt{D}\mathtt{<:}\mathtt{C}$.

4. CONCLUSIONS AND RELATED WORK

We have formalized a type system for dynamic layer deactivation with on-demand layer activation and proved its soundness. One key idea is to activate all the depended layers one after another at each method call, and to compute the activation order of them. We have not implement this mechanism. The performance issue remains as future work.

This result is sufficient for a particular layer activation mechanism, `ensure`, which does not change the order of already activated layers. We believe that this mechanism can also be applied to other layer activation mechanisms such as `with` that always activates the specified layer as the first layer to be executed by changing the order of already activated layers (by applying the proof of safety of proceed calls even when the order of active layers is changed at runtime [13]).

Layer activation that is implicitly performed when the layer that depends on that layer becomes active is proposed in the setting of *composite layers* [5, 16]. In [5], an extension of ContextL [6] with layer composition operators that are as expressive as compositions in feature diagrams [17] (such as and-composition and or-composition). At each layer activation point, it calculates the set of depended layers and activates them. If that set is ambiguous, it suspends the execution until when the user resolves this ambiguity. In [16], the similar mechanism is discussed in the setting of event-based layer transition [14]. FECJ° [15] formalizes the operational semantics of composite layers, but does not provide method introduced by layers and its type system.

The dependency between layers can also be specified in some COP languages such as Subjective-C [7] and Ambience [8]. In these languages, such dependency is checked at runtime.

5. REFERENCES

[1] Tomoyuki Aotani, Tetsuo Kamina, and Hidehiko Masuhara. Featherweight EventCJ: a core calculus for a context-oriented language with event-based per-instance layer transition. In *COP'11*, 2011.

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

[3] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *SC'10*, volume 6144 of *LNCS*, pages 50–65, 2010.

[4] Dave Clarke and Ilya Sergey. A semantics for context-oriented programming with layers. In *COP'09*, 2009.

[5] Pascal Costanza and Theo D'Hondt. Feature descriptions for context-oriented programming. In *DSPL'08*, 2008.

[6] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming – an overview of ContextL. In *DLS'05*, pages 1–10, 2005.

[7] Sebastián González, Micolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-C: Bringing context to mobile platform programming. In *SLE'11*, volume 6563 of *LNCS*, pages 246–265, 2011.

[8] Sebastián González, Kim Mens, and Alfredo Cádiz. Context-oriented programming with the ambient object systems. *Journal of Universal Computer Science*, 14(20):3307–3332, 2008.

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

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

[11] Atsushi Igarashi, Robert Hirschfeld, and Hidehiko Masuhara. A type system for dynamic layer composition. In *FOOL'12*, pages 13–24, 2012.

[12] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java

and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[13] H Inoue. A proof of soundness of type system for dynamic layer composition. Undergraduate honors thesis, Kyoto University, 2013.

[14] 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.

[15] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. A core calculus of composite layers. In *FOAL'13*, pages 7–12, 2013.

[16] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Introducing composite layers in EventCJ. *IPSJ Transactions on Programming*, 6(1):1–8, 2013.

[17] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.