Method Safety Mechanism for Asynchronous Layer Deactivation

Tetsuo KAMINA

Ritsumeikan University, Japan

Tomoyuki AOTANI and Hidehiko MASUHARA

Tokyo Institute of Technology, Japan

Atsushi IGARASHI

Kyoto University, Japan

Abstract

Context-oriented programming (COP) enhances the modularity of context-dependent behavior in context-aware systems, as it provides modules to implement context-dependent behavior (layers) and composes them dynamically in a disciplined manner (layer activation). We propose a COP language that enables layers to define base methods, while the layers can be asynchronously activated and deactivated. Base methods in layers enhance modularity because they extend the interface of classes without modifying original class definitions. However, calling such a method defined in a layer is problematic as the layer may be inactive when the method is called. We address this problem by introducing a method lookup mechanism that uses the static scope of method invocation for COP; i.e., in addition to currently activated layers, the layer where the method invocation is written, as well as the layers on which that layer depends, are searched during method lookup. We formalize this mechanism as a small calculus referred to as ContextFJ^a and prove its type soundness. We implement this mechanism in ServalCJ, a COP language that supports asynchronous, as well as synchronous, layer activation.

Keywords: Context-oriented programming, Layer-introduced base method, ContextFJ, ServalCJ

1. Introduction

For several years, context awareness has been a major concern in multiple application areas, and its importance is increasing. For example, with progress in sensor technologies, computing platforms have become more aware of physical environment, and user interfaces have become more adaptable to users' current operations. These interactions with the environment require the ability to change behavior with respect to *context*, such as a specific state of the physical environment or a user's current task. Such dynamic changes in behavior result in complicated system structures and behaviors that are difficult to predict using traditional programming abstractions. To address this difficulty, several context-oriented programming (COP) languages, which have successfully modularized such context-dependent behavior, have been developed [1, 2, 3, 4, 5, 6, 7, 8, 9].

COP languages provide language constructs that modularize the variations in behavior that depend on context using *layers*[1] and dynamically activate/deactivate them according to the executing contexts [2, 1]. A layer defines *partial methods*, which run before, after, or around a call of a method with the same signature defined in a class, only when the layer is active. These constructs make COP advantageous in terms of modularity, because partial methods can change the original behavior by activating layers without changing the base classes, and ensure consistency in dynamic changes using scoping [2] or model checking [5].

Even though several COP languages support only partial methods, *layer-introduced base methods*, i.e., methods in a layer that introduce a new signature and do not override other methods, are also known to be useful in COP [10]. Layer-introduced base methods considerably enhance modularity because they extend the interface of classes dynamically, which makes ensuring type-soundness in COP languages more challenging. Formal calculi have been

[1] In this study, we focus on layer-based COP languages.

proposed to support such an extension, e.g., (1) requiring a subordinate layer (a layer that provides base methods) to be activated while the dominant layer (a layer that uses these methods) is executing [10, 11], and (2) activating the subordinate layer on-demand when the dominant layer is executing [12].[2]

However, there is an issue in combining these approaches with *asynchronous* layer activation [13, 6, 14, 5, 8] in a type-safe manner. Asynchronicity is crucial to application domains where contexts change outside of a program, for example, ubiquitous computing applications and adaptive user interface. The method lookup in existing COP semantics searches all activated layers and the class of a method receiver to dispatch a called method. This semantics does not lead to a problem when layers are activated using `with`-blocks, where the corresponding layer activation is synchronous with the currently executing block. However, it leads to a problem in asynchronous layer activation, where layers are activated and deactivated by external events such as changes in external environment and user operations. These events may occur at any program execution point. Thus, it is possible that the layer that provides a base method to a currently executing method is eventually deactivated, resulting in a method-not-understood error.

In this paper, we propose another method lookup mechanism for type-safe layer deactivation[3]. In this mechanism, methods are searched in the layer *where method invocation is placed and in the layers on which this layer depends*, as well as in currently activated layers. This inclusion of the "static scope" for method lookup addresses the abovementioned problem of method safety. In other words, the proposed approach supports layer deactivation in the "best effort" manner; that is, the deactivation of layers is ensured to the maximum extent, while the deactivation can be canceled when the layers that require these

[2] To be precise, there is a flaw in the proof of type soundness for on-demand activation [12]. To ensure type soundness, we need to modify the reduction of method invocation to enclose the entire method execution within the activation of all required layers.

[3] This paper is an extended version of our previous work [15]. The main differences from the previous work are the `ensureDeactivate` mechanism, a complete set of computation rules and a type system, and the proof of type soundness.

layers are activated. This approach is a natural extension of *loyal strategy* [16], which most COP languages adopt, in that the execution of the required behavior is ensured during the execution of the partial method in the requiring layer.

This approach is applicable when layer deactivation is not a hard requirement. However, it leads to a problem in other cases. For example, we may consider a layer that performs a computation with highly precise results, thereby consuming considerable CPU power. Deactivation of this layer is a hard requirement when a battery is approaching exhaustion because a computation that consumes considerable CPU power should not be performed in this case. However, a layer that is not deactivated may require this high precision layer; thus, it cancels the deactivation.

To resolve this problem, we also introduce an additional mechanism that ensures deactivation of specified layers, i.e., a modifier, `ensureDeactivate`, for layer declarations, which indicates that the deactivation of the declared layer cannot be canceled. To ensure that the methods in layers, which are declared with `ensureDeactivate`, are not called accidentally by other layers that require those layers, layers cannot require the layers declared with `ensureDeactivate`.

We formalize this idea as a small COP calculus referred to as ContextFJ^a, and show that this calculus ensures deactivation of layers with `ensureDeactivate` and is type sound. This calculus is an extension of ContextFJ [10] and notably simple, even though it is sufficiently expressive to represent asynchronous layer activation and layer-introduced base methods.

The proposed mechanism is implemented in ServalCJ, a COP language with a generalized layer activation mechanism [17]. ServalCJ supports asynchronous as well as synchronous layer activation, and per-instance as well as global layer activation. Originally, ServalCJ did not support layer-introduced base methods. As the proposed mechanism ensures the safety of method with asynchronous layer activation, we safely realize layer-introduced base methods in a generalized layer activation mechanism.

The rest of this paper is structured as follows. In Section 2, an overview of COP mechanisms, such as layers, layer activation, and layer-introduced base methods, is provided. In this section, the problem that is addressed in this paper is also identified. In Section 3, we illustrate the proposed method lookup and formalize it as a small calculus, ContextFJ^a. In Section 4, we discuss the problem of layer deactivation cancellation and the proposed solution. In Section 5, we describe the type system of ContextFJ^a and prove its type soundness. In Section 6, the implementation of the proposed mechanism is described. In Section 7, related work is discussed. Lastly, conclusions are stated in Section 8.

2. Layers, Layer-Introduced Base Methods, and Their Problem

We demonstrate a motivating example of an adaptive user interface, which comprises a text editor program that was inspired by the program editor example proposed by Appeltauer et al. [18]. Our example includes class `Editor` (and other classes) to represent an editor view for the user. This user interface provides a menubar (and other widgets), which is displayed by calling `showMenuBar` when the display is refreshed, as shown below.

```
class Editor {
  JMenu menu;
  ...
  void showMenuBar() { menu.revalidate(); }
}
```

2.1. Layers and Partial Methods

We consider an additional feature to support programming using the above editor. This feature adaptively becomes available in response to the type of file opened by this editor and is dynamically composed with the system by *layers* in COP. The following `Programming` layer implements this feature:

```
layer Programming {
  class Editor {
    JMenuItem start = ..., stop = ..., resume = ...;
    void showMenuBar() {
      menu.add(start);
      menu.add(stop);
      menu.add(resume);
      proceed();
    }
  }
  /* other (partial) class declarations */
}
```

A *partial method*, `showMenuBar`, overrides the *base method* declared in `Editor` when `Programming` is *activated*, i.e., when it is dynamically composed with the application. The `proceed` call invokes the overridden behavior. We refer to the set of classes that provides their behavior when no layers are activated as the *base layer*.

In ContextJ [3], the following `with`-block is used to activate a specified layer:

```
Editor editor = new Editor();
with Programming { editor.showMenuBar(); }
```

Layer activation is effective in the *dynamic extent* of the `with`-block. Thus, the `Programming` layer is active when `showMenuBar` is called. Thus, the partial method defined in `Programming` (that adds several menu items to control program execution) is called.

However, in this example, it is difficult to enclose layer activation within a `with`-block, because events that activate layers are generated by the user's operations, e.g., opening a file that contains the source code, or clicking a button that switches the program editor to programming mode. Such operations are inherently asynchronous with the main thread of program execution. We cannot encode such an event-driven layer activation using `with`-blocks without boilerplate code [18].

Thus, several COP languages that provide asynchronous layer activation have been proposed [13, 6, 14, 5, 17, 8]. Figure 1 shows asynchronous layer activation in ServalCJ [17]. It declares that when the layer `Programming` is

```
global contextgroup ProgrammingCtl {
  activate Programming from startProgramming to endProgramming;
}
```

Figure 1: Layer activation in ServalCJ.

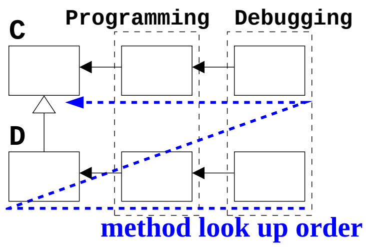

Figure 2: Method lookup order in ContextFJ. Suppose that `Debugging` is activated after `Programming`, and `D` is a subclass of `C`. Method lookup proceeds from `D` in `Debugging` to `C` in the base layer, as shown by the dashed arrow.

activated using the `activate` declarations, which specify events that activate the layer in the `from` clause and those that deactivate the layer in the `to` clause. Thus, `Programming` is activated when the `startProgramming` event is generated and deactivated when the `endProgramming` event is generated. These events can be declaratively specified using AspectJ pointcuts [19] (not shown in this paper) that specify, e.g., a join point when an event handler for the user's operation, such as "pressing the start programming button", starts.

We note that multiple layers can be activated, and in most COP languages, the most recently activated layer supersedes other layers. Furthermore, in COP languages based on class-based object-oriented languages, inheritance relations exist between classes, where the most specific subclass supersedes other superclasses. Thus, there are multiple directions for method lookup from the most recently activated layer that provides partial methods for the most specific subclass to the base class. ContextFJ (and similar COP languages) linearizes this method lookup, as shown in Figure 2, where method lookup starts from the most recently activated layers to the base layer on the most specific subclass, and then, these layers are searched again when the method lookup proceeds to the next subclass. We also note that the `proceed` calls follow only the horizontal direction (the sequence of the activated layers), while the `super` calls follow the vertical direction (the class inheritance relations).

2.2. Layer-Introduced Base Methods

In a few COP languages such as JCop [4], a layer may also declare a base method, as shown in the following example:

```
layer Programming {
  class Editor {
    void execute() { ... }
    JTextArea getConsole() { ... }
    ... /* same as above */
  }
}
```

In this example, two base methods, `execute` and `getConsole`, are declared in layer `Programming`. These methods are not visible from the base program. Thus, the primary purpose of a layer-introduced base method is that it is called from the same layer or other layers depending on this layer. For example, the `execute` method defines behavior to start the execution of the program. This behavior is registered as an action associated with the menu item added by `Programming` and is not visible from the base program.

We note that a layer-introduced base method is not a private method visible only within the layer; in some cases, it should be visible from other layers. For example, in the adaptive user interface example, we may also consider an additional feature for debugging the currently developed program. This feature is implemented by layer `Debugging`, as shown in Figure 3. This layer declares two partial methods, `showMenuBar` and `execute`, and the second method calls

```
layer Debugging requires Programming {
  class Editor {
    JMenuItem stopDebugging = ...;
    void showMenuBar() {
       // a menu item for stopping debugging
      menu.add(stopDebugging);
      proceed();
    }
    void execute() {
      ... /* enabling the step-by-step execution */
      ... getConsole() ...
     /* accessing console to display debug info */
    }
  }
}
```

Figure 3: Layer dependency.

getConsole. The getConsole method is added by Programming, which implies that Debugging assumes the existence of Programming. In Figure 3, this dependency is represented by the requires clause in the first line of the layer declaration.

This requires clause was first introduced by ContextFJ [10], which supports with-blocks, and it implies that when Debugging is activated, Programming must also be activated. To activate Debugging, we need to activate Programming beforehand, which implies that Debugging can be activated only within the with-block that activates Programming.

```
Editor editor = new Editor();
with Programming {
  with Debugging { editor.execute(); }
}
```

Within with Debugging, both Programming and Debugging are active, and the partial methods in Debugging override those in Programming because Debugging is the most recently activated layer. Thus, the above execute call safely calls the getConsole provided by Programming. The type system can detect the erroneous activation of Debugging that is not enclosed within activation of Programming.

2.3. Problem

The existing method lookup semantics in COP, in which only the activated layers and the base layer of the method receiver are searched, cannot operate with asynchronous layer (de)activation, where layer activation is not enclosed within the statically known with-blocks, in a type-safe manner.[4] Unlike with-blocks, where layer activation is synchronous with the currently executing block, in asynchronous layer activation, layer (de)activation may occur at any program execution point. This makes ensuring method safety difficult. For example, in

[4] Moreover, as discussed in Section 7.1.1, ContextFJ does not support layer deactivation.

Figure 3, deactivation of `Programming` may occur immediately before the call of `proceed` in `execute`, resulting in a method-not-understood error because this method is introduced by `Programming`.

3. Safe Method Lookup for COP

3.1. Overview of Our Approach

There are two approaches to addressing the problem of dynamic layer deactivation. The one is to prohibit layer deactivation when it is not safe and postpone it until it becomes safe. If we adopt this approach, we need to significantly change the underlying dynamic or static semantics provided by ContextFJ. The other approach is to change the manner of method lookup to prevent the method-not-understood error, where changes are required only in the method lookup, and the dynamic semantics and proof of type soundness can be defined and proven by simply following the manner applied in ContextFJ.

We decided to adopt the second approach because it keeps the underlying semantics simple. We use the enclosing layer for method lookup; i.e., during method lookup, the layer where the method invocation is written and the layers on which this layer depends are searched in addition to currently activated layers and the base layer. For example, in the following layer `L1`, assume that layer `L3` is activated before the method invocation `n()` is evaluated.

```
layer L1 requires L2 {
  ...  m() { ... n() ... } ...
}
```

The proposed method lookup mechanism remembers the layer where the method invocation is written, and this layer and the layer required by it are used for the method lookup. In this case, the method lookup is performed in the order of `L1`, `L2`, `L3`, and the base layer.

3.2. Technical Challenges

Even though it may appear that this approach can be applied easily, it leads to non-trivial problems. In the following paragraphs we introduce the problem about duplication of partial method execution.

Since this method lookup mechanism searches in the static scope, where the method invocation is written, and in the currently activated layers, a situation may occur where the same layer is searched twice if it exists in the static scope and currently activated layers. For example, suppose `Debugging` and `Programming`, described in Section 2, are activated simultaneously in the order of `Programming`; `Debugging` (`Debugging` is most recently activated). If the method lookup mechanism is naively realized, assuming that `showMenuBar` is called in the body of a method in `Debugging`, partial method `showMenuBar` in `Programming` is called twice, because `showMenuBar` in `Debugging` calls `proceed`, which results in the execution of `showMenuBar` in `Programming`, which is *required by* `Debugging`, and this partial method also calls `proceed`. Thus, `showMenuBar` in `Programming` (in the currently activated layers) is dispatched again, resulting in multiple appearances of the same menu items.

The proposed method lookup mechanism addresses this problem by constructing the sequence of layers and removing such a duplication from that sequence at the time the method lookup is started. We note that, to ensure type soundness, the `requires` relation should not be broken when removing the duplication. For example, even when the abovementioned layers are activated in the order of `Debugging`;`Programming`, where `Programming` is the most recently activated layer, the method body is searched in the order of `Programming`;`Debugging` (the rightmost layer is searched first), because the partial method `showMenuBar` is called from a method in `Debugging`. This implies that the proposed mechanism does not follow the COP convention in which the most recently activated layer always dominates over others.

$$
\begin{array}{lcl}
\texttt{CL} & ::= & \texttt{class C} \lhd \texttt{C \{ } \overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\texttt{ \}} \\
\texttt{K} & ::= & \texttt{C(}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{)\{ super(}\overline{\texttt{f}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{; \}} \\
\texttt{M} & ::= & \texttt{C m(}\overline{\texttt{C}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}} \\
\texttt{e},\texttt{d} & ::= & \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m(}\overline{\texttt{e}}\texttt{)@X} \mid \texttt{new C(}\overline{\texttt{e}}\texttt{)} \\
 & & \mid \texttt{proceed(}\overline{\texttt{e}}\texttt{)} \mid \texttt{v<C,}\overline{\texttt{L}}\texttt{,}\overline{\texttt{L}}\texttt{>.m(}\overline{\texttt{v}}\texttt{)} \\
\texttt{v},\texttt{w} & ::= & \texttt{new C(}\overline{\texttt{v}}\texttt{)} \\
\texttt{X} & ::= & \texttt{L} \mid \cdot
\end{array}
$$

Figure 4: ContextFJ^a: Abstract syntax.

3.3. ContextFJ^a

We formalize the proposed method lookup as a simple calculus, ContextFJ^a, which is based on ContextFJ [10]. The abstract syntax for ContextFJ^a is shown in Figure 4. Let metavariables `C`, `D`, `E`, and `F` range over class names; `L` ranges over layer names; `f` and `g` range over field names; `m` ranges over method names; `e` ranges over expressions; `v` and `w` range over values; and `x` ranges over variables, which include a special variable, `this` (unlike ContextFJ, `super` is not formalized in this calculus). `X` ranges over static context. This can be a layer name or a base layer (denoted by ·), indicating that the method invocation is written in the base class. Syntactically, a difference from ContextFJ is the annotation, `@X`, on method invocation, which indicates the static context where the method invocation is evaluated.

Overlines denote sequences, e.g., $\overline{\texttt{f}}$ stands for a possibly empty sequence $\texttt{f}_1, \cdots, \texttt{f}_n$. An empty sequence is written by •. We also abbreviate a sequence of pairs by writing "$\overline{\texttt{C}}\ \overline{\texttt{f}}$" for "$\texttt{C}_1\ \texttt{f}_1, \cdots, \texttt{C}_n\ \texttt{f}_n$," where n denotes the length of $\overline{\texttt{C}}$ and $\overline{\texttt{f}}$. Similarly, we write "$\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{;}$" as shorthand for the sequence of declarations "$\texttt{C}_1\ \texttt{f}_1\texttt{;} \cdots \texttt{;C}_n\ \texttt{f}_n\texttt{;}$" and "$\texttt{this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;}$" as shorthand for "$\texttt{this.f}_1\texttt{=f}_1\texttt{;}\cdots\texttt{;this.f}_n\texttt{=f}_n\texttt{;}$". We use commas and semicolons for concatenations.

Unlike existing COP languages, the calculus does not provide syntax for layers. Partial methods are registered in a partial method table, PT, which maps a triple, $\mathtt{C}$, $\mathtt{L}$, and $\mathtt{m}$, of class, layer, and method names to a method definition. The runtime expression, $\mathtt{new\ C(\overline{v})<C,\overline{L}',\overline{L}>.m(\overline{e})}$, where $\overline{\mathtt{L}}'$ is assumed to be a prefix of $\overline{\mathtt{L}}$, means that $\mathtt{m}$ is going to be invoked on $\mathtt{new\ C(\overline{v})}$. Annotation $\mathtt{<C,\overline{L}',\overline{L}>}$ indicates the cursor where method lookup should start. The manner in which this cursor is used in the method lookup will be explained in Section 3.5.

A method invocation is annotated with a static context $\mathtt{X}$. It is assumed that if $\mathtt{C\ m(\overline{C}\ \overline{x})}\ \{\ \mathtt{return\ e;}\ \} \in PT(\mathtt{D},\mathtt{L},\mathtt{m})$ and $\mathtt{e_0.m_1(\overline{e})@X}$ is a subexpression of $\mathtt{e}$, then $\mathtt{X} = \mathtt{L}$. Similarly, if $\mathtt{C\ m(\overline{C}\ \overline{x})}\ \{\ \mathtt{return\ e;}\ \}$ is defined in some class $\mathtt{D}$ and not in PT, and $\mathtt{e_0.m_1(\overline{e})@X}$ is a subexpression of $\mathtt{e}$, then $\mathtt{X} = \cdot$.

The dependency between layers indicated by $\mathtt{requires}$ is modeled by a binary relation, $\mathcal{R}$, on layer names; $(\mathtt{L_1},\mathtt{L_2}) \in \mathcal{R}$ intuitively means that $\mathtt{L_1}$ $\mathtt{requires}$ $\mathtt{L_2}$. We assume a fixed dependency relation and write $\mathtt{L}\ \mathtt{req}\ \Lambda$, read as "layer $\mathtt{L}$ requires layers Λ," when $\Lambda = \{\mathtt{L}' | (\mathtt{L},\mathtt{L}') \in \mathcal{R}\}$. We write $\overline{\mathtt{L}}\ \mathtt{req}\ \Lambda$ if $\mathtt{L}_i\ \mathtt{req}\ \Lambda_i$ for all $\mathtt{L}_i \in \overline{\mathtt{L}}$ and $\Lambda = \Lambda_1 \cup \cdots \cup \Lambda_n$ where n is the length of $\overline{\mathtt{L}}$. We assume that the relation induced by $\mathtt{req}$ is acyclic. We write $\{\overline{\mathtt{L}}\}$ to denote a set whose members are equivalent to those of $\overline{\mathtt{L}}$ but the ordering is not important. We define two auxiliary functions, *requires* and *filter*, that calculate the reflexive and transitive closure of $\mathtt{req}$ and remove the duplication of layer names, respectively, as follows[5]:

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

$$\begin{array}{rcll} \mathit{filter}(\overline{\mathtt{L}};\mathtt{L};\overline{\mathtt{L}}';\mathtt{L};\overline{\mathtt{L}}'') & = & \mathit{filter}(\overline{\mathtt{L}};\overline{\mathtt{L}}';\mathtt{L};\overline{\mathtt{L}}'') & \\ \mathit{filter}(\overline{\mathtt{L}}) & = & \overline{\mathtt{L}} & \text{if } \forall \mathtt{L_1},\mathtt{L_2} \in \overline{\mathtt{L}}.\ \mathtt{L_1} \neq \mathtt{L_2} \end{array}$$

A program $(CT, PT, \mathcal{R}, \mathtt{e})$ consists of a class table CT that maps a class name $\mathtt{C}$ to a class definition $\mathtt{CL}$, a partial method table PT, the binary relation $\mathcal{R}$,

[5] We note that we use set Λ instead of a sequence. The order in the set is not important in this case, because in the core language, there is no syntax for layers; thus, we cannot specify the precedence in Λ as in Section 3, e.g., `L requires L1,L2 {...}`. In other words, we model a more general language in this case compared to that presented in Section 3.

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

$$\mathit{fields}(\mathtt{Object}) = \bullet \qquad \text{(F-OBJECT)}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \cdots\ \} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}}} \qquad \text{(F-CLASS)}$$

Figure 5: ContextFJ^a: Field access.

and an expression e that corresponds to the body of the main method. In other words, the class table and the partial method table define functions CT and PT, respectively. We assume that $CT(\mathtt{C}) = \mathtt{class\ C\ \ldots}$ for any $\mathtt{C} \in dom(CT)$, $PT(\mathtt{m}, \mathtt{C}, \mathtt{L}) = \ldots\ \ \mathtt{m(\ldots)\{\ldots\}}$ for any $(\mathtt{m}, \mathtt{C}, \mathtt{L}) \in dom(PT)$, $\overline{\mathtt{C}}, \mathtt{C}_0 = \overline{\mathtt{D}}, \mathtt{D}_0$ if $PT(\mathtt{m}, \mathtt{C}, \mathtt{L}_1) = \mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\cdots\}$ and $PT(\mathtt{m}, \mathtt{C}, \mathtt{L}_2) = \mathtt{D}_0\ \mathtt{m}(\overline{\mathtt{D}}\ \overline{\mathtt{x}})\{\cdots\}$ for all m and C (i.e., there is no conflict between partial methods), and no cycles exist in the transitive closure of $\lhd$ (extends).

3.4. Auxiliary Definitions

We first define a few auxiliary functions for the lookup of fields and method definitions, which are defined in Figures 5 and 6. The field lookup function, $\mathit{fields}(\mathtt{C})$, returns a sequence $\overline{\mathtt{C}}\ \overline{\mathtt{f}}$ of the pairs of a field name and its type by placing all field declarations from C at the tail of those from its superclasses. The method lookup function, $\mathit{mbody}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}_1, \overline{\mathtt{L}}_2)$, returns a pair $\overline{\mathtt{x}}.\mathtt{e}$ of parameters and an expression of method m in class C when the search starts from the sequence of layers $\overline{\mathtt{L}}_1$. $\overline{\mathtt{L}}_2$ keeps track of the layers that are active when the search starts; i.e., the method lookup follows the order described in Figure 2. $\overline{\mathtt{L}}_2$ is used to reset the cursor when the lookup proceeds to the superclass. In addition, it returns the name of the class and the sequence of layers where the method has been found, which will be used in reduction rules to deal with proceed. We note that the proposed method lookup traverses the sequence of layers from last to first.

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

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C} \triangleleft \mathtt{D}\ \{\ \cdots\ \mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ \mathtt{return}\ \mathtt{e};\ \}\ \cdots\ \}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}, \bullet, \overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e}\ \mathtt{in}\ \mathtt{C}, \bullet} \quad \text{(MB-Class)}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C} \triangleleft \mathtt{D}\ \{\ \cdots\ \overline{\mathtt{M}}\ \} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \qquad \mathit{mbody}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{L}}, \overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e}\ \mathtt{in}\ \mathtt{E}, \overline{\mathtt{L}}'}{\mathit{mbody}(\mathtt{m}, \mathtt{C}, \bullet, \overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e}\ \mathtt{in}\ \mathtt{E}, \overline{\mathtt{L}}'} \quad \text{(MB-Super)}$$

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

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

Figure 6: ContextFJ^a: Method body lookup.

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{).f}_i \mid \overline{\mathtt{L}} \longrightarrow \mathtt{v}_i \mid \overline{\mathtt{L}}} \qquad (\textsc{R-Field})$$

$$\frac{\overline{\mathtt{L}}' = \mathit{filter}(\overline{\mathtt{L}}; \mathit{requires}(\mathtt{X}))}{\mathtt{new\ C(}\overline{\mathtt{w}}\mathtt{).m(}\overline{\mathtt{v}}\mathtt{)@X} \mid \overline{\mathtt{L}} \longrightarrow \mathtt{new\ C(}\overline{\mathtt{w}}\mathtt{)<C,}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}'\mathtt{>.m(}\overline{\mathtt{v}}\mathtt{)} \mid \overline{\mathtt{L}}} \qquad (\textsc{R-Invk})$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}'', \overline{\mathtt{L}}') = \overline{\mathtt{x}}\mathtt{.e\ in\ C}', \bullet}{\mathtt{new\ C_0(}\overline{\mathtt{v}}\mathtt{)<C,}\overline{\mathtt{L}}''\mathtt{,}\overline{\mathtt{L}}'\mathtt{>.m(}\overline{\mathtt{w}}\mathtt{)} \mid \overline{\mathtt{L}} \longrightarrow \left[\begin{array}{ll} \mathtt{new\ C_0(}\overline{\mathtt{v}}\mathtt{)} & \mathtt{/this,} \\ \overline{\mathtt{w}} & \mathtt{/}\overline{\mathtt{x}} \end{array}\right] \mathtt{e} \mid \overline{\mathtt{L}}} \qquad (\textsc{R-InvkB})$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}'', \overline{\mathtt{L}}') = \overline{\mathtt{x}}\mathtt{.e\ in\ C}', (\overline{\mathtt{L}}'''; \mathtt{L}_0)}{\begin{array}{c}\mathtt{new\ C_0(}\overline{\mathtt{v}}\mathtt{)<C,}\overline{\mathtt{L}}''\mathtt{,}\overline{\mathtt{L}}'\mathtt{>.m(}\overline{\mathtt{w}}\mathtt{)} \mid \overline{\mathtt{L}} \longrightarrow \\ \left[\begin{array}{ll} \mathtt{new\ C_0(}\overline{\mathtt{v}}\mathtt{)} & \mathtt{/this,} \\ \overline{\mathtt{w}} & \mathtt{/}\overline{\mathtt{x}}\mathtt{,} \\ \mathtt{new\ C_0(}\overline{\mathtt{v}}\mathtt{)<C}'\mathtt{,}\overline{\mathtt{L}}'''\mathtt{,}\overline{\mathtt{L}}'\mathtt{>.m} & \mathtt{/proceed} \end{array}\right] \mathtt{e} \mid \overline{\mathtt{L}}\end{array}} \qquad (\textsc{R-InvkP})$$

$$\frac{f(\mathtt{L}, \overline{\mathtt{L}}) = \overline{\mathtt{L}}' \qquad f = \mathit{activate} \text{ or } \mathit{deactivate}}{\mathtt{e} \mid \overline{\mathtt{L}} \longrightarrow \mathtt{e} \mid \overline{\mathtt{L}}'} \qquad (\textsc{R-Activate})$$

Figure 7: ContextFJ^a: Operational semantics (1).

3.5. Operational Semantics

The operational semantics of ContextFJ^a, which is shown in Figure 7, is given by a reduction relation of the form $\mathtt{e} \mid \overline{\mathtt{L}} \longrightarrow \mathtt{e}' \mid \overline{\mathtt{L}}'$, which is read as "expression $\mathtt{e}$ under activated layers $\overline{\mathtt{L}}$ reduces to $\mathtt{e}'$ under $\overline{\mathtt{L}}'$."

Rule R-Field is defined for field access and it is straightforward: *fields* specifies the argument to $\mathtt{new\ C(\cdots)}$ that corresponds to $\mathtt{f}_i$.

Rule R-Invk represents the method invocation. This rule is for the method invocation where the cursor of the method lookup has not been "initialized"; the cursor is set as the receiver's class and the sequence of layers computed by *filter* and *requires* using currently activated layers $\overline{\mathtt{L}}$ and layer $\mathtt{L}$ attached to the method invocation. This ensures that in addition to currently activated layers $\overline{\mathtt{L}}$, static context $\mathtt{L}$ and all layers required by $\mathtt{L}$ are always searched when the method is called.

Two rules, R-InvkB and R-InvkP, represent invocations of a base method and a partial method, respectively. Both rules are straightforward adaptations of the method invocation on the runtime expression of the form `new C(`$\overline{\mathtt{v}}$`)<C,`$\overline{\mathtt{L}}$`,`$\overline{\mathtt{L}}$`>.m(`$\overline{\mathtt{v}}$`)` from ContextFJ. In R-InvkB, the receiver is `new C(`$\overline{\mathtt{v}}$`)` and the location of the cursor is `<`$\mathtt{C}'$`,`$\overline{\mathtt{L}}''$`,`$\overline{\mathtt{L}}'$`>`. When the method body is found in the base-layer class $\mathtt{C}'$ (denoted by "`in` $\mathtt{C}', \bullet$"), the expression reduces to the method body, where the formal parameters, $\overline{\mathtt{x}}$ and `this`, are replaced with the actual arguments $\overline{\mathtt{w}}$ and the receiver, respectively. Rule R-InvkP addresses 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, where the receiver's cursor points to the next layers $\overline{\mathtt{L}}'''$.

We also introduce a reduction rule, R-Activate, to model asynchronous layer activation and deactivation. This rule represents layer activation and deactivation that occur non-deterministically ($\mathtt{L}$ in the rule is arbitrarily chosen). Auxiliary functions *activate* and *deactivate* are defined as

$$\begin{array}{rcll}
\mathit{activate}(\mathtt{L},\overline{\mathtt{L}}) & = & \overline{\mathtt{L}};\mathtt{L} & \text{if } \mathtt{L} \notin \overline{\mathtt{L}} \\
\mathit{activate}(\mathtt{L},(\overline{\mathtt{L}}';\mathtt{L};\overline{\mathtt{L}}'')) & = & \overline{\mathtt{L}}';\overline{\mathtt{L}}'';\mathtt{L} & \text{otherwise}
\end{array}$$

$$\begin{array}{rcll}
\mathit{deactivate}(\mathtt{L},\overline{\mathtt{L}}) & = & \overline{\mathtt{L}} & \text{if } \mathtt{L} \notin \overline{\mathtt{L}} \\
\mathit{deactivate}(\mathtt{L},(\overline{\mathtt{L}}';\mathtt{L};\overline{\mathtt{L}}'')) & = & \overline{\mathtt{L}}';\overline{\mathtt{L}}'' & \text{otherwise}
\end{array}$$

Function *activate* places the specified layer $\mathtt{L}$ at the right most position of the activated layer $\overline{\mathtt{L}}$; if $\mathtt{L}$ is already in $\overline{\mathtt{L}}$, this function changes the order of the activated layers such that the most recently activated layer has the highest priority. Thus, it keeps the invariant that the activated layers do not contain duplications. Function *deactivate* removes only the specified layer $\mathtt{L}$ (if it exists) from the currently activated layers $\overline{\mathtt{L}}$.

Figure 8 shows other trivial congruence rules that enable subexpressions

$$\frac{\texttt{e}_0 \mid \overline{\texttt{L}} \longrightarrow \texttt{e}_0' \mid \overline{\texttt{L}}}{\texttt{e}_0\texttt{.f} \mid \overline{\texttt{L}} \longrightarrow \texttt{e}_0'\texttt{.f} \mid \overline{\texttt{L}}} \qquad \text{(RC-}\textsc{Field}\text{)}$$

$$\frac{\texttt{e}_0 \mid \overline{\texttt{L}} \longrightarrow \texttt{e}_0' \mid \overline{\texttt{L}}}{\texttt{e}_0\texttt{.m(}\overline{\texttt{e}}\texttt{)@X} \mid \overline{\texttt{L}} \longrightarrow \texttt{e}_0'\texttt{.m(}\overline{\texttt{e}}\texttt{)@X} \mid \overline{\texttt{L}}} \qquad \text{(RC-}\textsc{InvkRecv}\text{)}$$

$$\frac{\texttt{e}_i \mid \overline{\texttt{L}} \longrightarrow \texttt{e}_i' \mid \overline{\texttt{L}}}{\texttt{e}_0\texttt{.m(}\cdots\texttt{,e}_i\texttt{,}\cdots\texttt{)@X} \mid \overline{\texttt{L}} \longrightarrow \texttt{e}_0\texttt{.m(}\cdots\texttt{,e}_i'\texttt{,}\cdots\texttt{)@X} \mid \overline{\texttt{L}}} \qquad \text{(RC-}\textsc{InvkArg}\text{)}$$

$$\frac{\texttt{e}_i \mid \overline{\texttt{L}} \longrightarrow \texttt{e}_i' \mid \overline{\texttt{L}}}{\texttt{new C(}\cdots\texttt{,e}_i\texttt{,}\cdots\texttt{)} \mid \overline{\texttt{L}} \longrightarrow \texttt{new C(}\cdots\texttt{,e}_i'\texttt{,}\cdots\texttt{)} \mid \overline{\texttt{L}}} \qquad \text{(RC-}\textsc{InvkNew}\text{)}$$

Figure 8: ContextFJ^a: Operational semantics (2).

to reduce. We note that ContextFJ^a reduction is call by value; similar to ContextFJ, the order of reduction of subexpressions is unspecified.

Example 1. Suppose the situation where `execute` is called on an instance of `Editor` with activated layer `Debugging`, and `Debugging` is asynchronously deactivated during the execution:

`new Editor().execute()`|`Debugging`

(method body in `Debugging` is dispatched)

$\longrightarrow^*$ `new Editor().getConsole()`@`Debugging`|`Debugging`

(`Debugging` is deactivated)

$\longrightarrow$ `new Editor().getConsole()`@`Debugging`

(method body is found in *filter*(*requires*(`Debugging`)))

$\longrightarrow^*$ {the body of `getConsole` in `Editor` in `Programming`}

$\longrightarrow \cdots$

These reductions demonstrate that the call of `getConsole`, which is written in layer Debugging, succeeds, even though Debugging is not active when this method is called, illustrating the type safety of the proposed calculus.

Example 2. Suppose the situation where showMenuBar is called on an instance of Editor from the static context of Debugging with activated layers Programming and Debugging, and Debugging is most recently activated (we abbreviate Programming; Debugging as $\overline{\mathtt{L}}$):

$\texttt{new Editor().showMenuBar()@Debugging}|\overline{\mathtt{L}}$

(method body in Debugging is dispatched)

$\longrightarrow \texttt{new Editor<Editor,}\overline{\mathtt{L}}\texttt{,}\overline{\mathtt{L}}\texttt{>.showMenuBar()}|\overline{\mathtt{L}}$

(by applying *filter*, the cursor is set to $\overline{\mathtt{L}}$)

$\longrightarrow^{*}$

This example shows the situation explained in Section 3.2. The duplication of showMenuBar in Debugging will not occur, as the cursor is set to Programming; Debugging.

4. Guaranteed Layer Deactivation

In the calculus defined above, an issue arises regarding handling layer deactivation. The proposed method lookup can cancel layer deactivation, even when it is mandatory.

At first, the proposed method lookup looks natural as compared to existing COP strategies for layer deactivation. Desmet et al. questioned the layered design approach for context-aware systems regarding handling layer switching when the layer is currently executing, and proposed two strategies, *loyal* and *prompt* [16]. Most COP languages adopt the loyal strategy, which ensures completion of partial method execution; thus, the deactivation is put into effect later. The proposed approach is a natural extension of this strategy; while executing the partial method in the requiring layer, the execution of the required behavior is ensured. This is thanks to the fact that the layer sequences in a cursor are not affected by layer deactivation. Furthermore, this approach ensures the execution of the required layer even when it is not activated.

However, this approach leads to a problem when layer deactivation is a hard requirement. For example, we may consider a layer, namely, `HighPrecision`, that performs a computation with highly precise results, thereby consuming considerable CPU power. Thus, it should be deactivated when a battery is approaching exhaustion. If other layers that require `HighPrecision` are still activated, then the deactivation is canceled in the proposed mechanism.

To address this problem, we introduce an additional mechanism that ensures deactivation of specified layers. A layer declared with the `ensureDeactivate` modifier is not included in the layers searched during the method lookup after it is deactivated.

```
ensureDeactivate layer HighPrecision { ... }
```

Deactivation is ensured by prohibiting the layer with `ensureDeactivate` from appearing on the right hand side of the `requires` clause. This restriction prevents the situation where the behavior in the layer with `ensureDeactivate` is eventually called by another layer that requires this layer.

To formalize this mechanism, we introduce a fixed set of layers, $\mathcal{D}$, which corresponds to a set of layers declared with `ensureDeactivate`. As $\mathcal{D}$ is a part of the program, the definition of a program is also extended to $(CT, PT, \mathcal{R}, \mathcal{D}, \mathtt{e})$.

We note that it is not necessary to extend the operational semantics of ContextFJ^a using $\mathcal{D}$. This is because, for $\mathtt{L} \in \mathcal{D}$ and $\mathtt{L}_1 \neq \mathtt{L}$, the type system may ensure that $\mathtt{L} \notin \mathit{requires}(\mathtt{L}_1)$, which means that, in the hypothesis of R-Invk, $\mathtt{L} \notin \mathit{filter}(\overline{\mathtt{L}}; \mathit{requires}(\mathtt{L}_1))$ if $\mathtt{L} \notin \overline{\mathtt{L}}$.

5. Type System

The type system for ContextFJ^a is a straightforward adaptation of the type system of FJ [20] and a simplification of ContextFJ, except that we must handle the layers with `ensureDeactivate`. The details are described below.

$$\frac{CT(\texttt{C}) = \texttt{class C} \triangleleft \texttt{D} \ \{ \ \cdots \ \texttt{C}_0 \ \texttt{m}(\overline{\texttt{C}} \ \overline{\texttt{x}})\{ \ \texttt{return e;} \ \} \ \cdots \ \}}{\mathit{mtype}(\texttt{m}, \texttt{C}, \Lambda_1, \Lambda_2) = \overline{\texttt{C}} \rightarrow \texttt{C}_0} \quad \text{(MT-Class)}$$

$$\frac{\texttt{L} \in \Lambda_1 \qquad PT(\texttt{m}, \texttt{C}, \texttt{L}) = \texttt{C}_0 \ \texttt{m}(\overline{\texttt{C}} \ \overline{\texttt{x}})\{ \ \texttt{return e;} \ \}}{\mathit{mtype}(\texttt{m}, \texttt{C}, \Lambda_1, \Lambda_2) = \overline{\texttt{C}} \rightarrow \texttt{C}_0} \quad \text{(MT-PMethod)}$$

$$\frac{\begin{array}{c}\texttt{class C} \triangleleft \texttt{D} \ \{ \ \cdots \ \overline{\texttt{M}} \ \} \qquad \texttt{m} \not\in \overline{\texttt{M}} \\ \forall \texttt{L} \in \Lambda_1 . PT(\texttt{m,C,L}) \text{ undefined} \qquad \mathit{mtype}(\texttt{m}, \texttt{D}, \Lambda_2, \Lambda_2) = \overline{\texttt{C}} \rightarrow \texttt{C}_0\end{array}}{\mathit{mtype}(\texttt{m}, \texttt{C}, \Lambda_1, \Lambda_2) = \overline{\texttt{C}} \rightarrow \texttt{C}_0} \quad \text{(MT-Super)}$$

Figure 9: ContextFJ^a: Method type lookup.

5.1. Subtyping

The subtyping relation, `C <: D`, which is the same as that in FJ, is defined as the reflexive and transitive closure of the `extends` (◁) clauses.

$$\frac{}{\texttt{C <: C}} \quad \text{(S-Refl)}$$

$$\frac{\texttt{C <: D} \qquad \texttt{D <: E}}{\texttt{C <: E}} \quad \text{(S-Trans)}$$

$$\frac{\texttt{class C} \triangleleft \texttt{D} \ \{\cdots\}}{\texttt{C <: D}} \quad \text{(S-Extends)}$$

5.2. Method Type Lookup

The method type lookup is defined in Figure 9. The function $\mathit{mtype}(\texttt{m}, \texttt{C}, \Lambda_1, \Lambda_2)$ takes a method name `m`, a class name `C`, and two sets Λ_1 and Λ_2 of layer names and returns a pair $\overline{\texttt{C}} \rightarrow \texttt{C}_0$ of argument types $\overline{\texttt{C}}$ and a return type $\texttt{C}_0$. Its definition is identical to that of *mtype* in ContextFJ. Sets Λ_1 and Λ_2 represent statically known active layers, in which `m` is searched. The first is used to lookup `m` in `C`, whereas the second is used when `m` is not found in `C` and the search continues to `C`'s superclass. This distinction is necessary for the typing of `proceed` because it cannot proceed to where it is executed; however, it may proceed to a method of the same name in a superclass in the same layer. In other uses, these two sets are the same and we write $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda)$ for $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda, \Lambda)$. Rule MT-Class is used when `m` is defined in the base layer, rule MT-PMethod is used when `m` is defined in one of the activated layers, and rule MT-Super is used when `m` is not defined in class `C`. We note that *mtype* is a (partial) function under the assumption that there is no conflict between partial methods, as explained at the end of Section 3.3.

5.3. Typing

The typing rules for expressions are shown in Figure 10. A type environment Γ is a finite mapping from variables to class names. We use $\mathcal{L}$ to represent a *context*, 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`). A type judgment for expressions is of the form $\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e} : \mathtt{C}$, read as "expression `e` has type `C` under type environment Γ, context $\mathcal{L}$, and layers Λ." Layers Λ are supposed to be a subset of layers that are set as a cursor when the method lookup starts at runtime. We note that Λ is not a sequence, it is a set; i.e., the type system does not know the order in which the layers are linearized.

The rules, T-Var for variables, T-Field for field access, and T-New for object instantiation, are straightforward adaptations of those of FJ. The rule T-Invk for method invocation applies *mtype* to Λ, which includes all and only the layers required by the static context `X`, denoting the layers that may change the interface of $\mathtt{C}_0$; i.e., Λ is used to confirm that an use of a layer-introduced base method is checked. The condition $\Lambda = \mathit{requires}(\mathtt{X})$ always holds, as `X` is the name of the enclosing layer (see T-PMethod in Figure 11). The next rule is regarding `proceed` calls (T-Proceed), and it constitutes straightforward adaptations of the corresponding typing rule of ContextFJ. In this rule, the

$$\frac{\Gamma = \overline{\mathtt{x}} : \overline{\mathtt{C}}}{\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{x}_i : \mathtt{C}_i} \qquad \text{(T-}\textsc{Var}\text{)}$$

$$\frac{\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e}_0\mathtt{.f}_i : \mathtt{C}_i} \qquad \text{(T-}\textsc{Field}\text{)}$$

$$\frac{\begin{array}{c}\Lambda = \mathit{requires}(\mathtt{X}) \qquad \mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathcal{L}; \Lambda; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} \\ \overline{\mathtt{E}}\ \mathtt{<:}\ \overline{\mathtt{D}} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0, \Lambda) = \overline{\mathtt{D}} \rightarrow \mathtt{D}_0\end{array}}{\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)@X} : \mathtt{D}_0} \qquad \text{(T-}\textsc{Invk}\text{)}$$

$$\frac{\mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \qquad \mathcal{L}; \Lambda; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}}\ \mathtt{<:}\ \overline{\mathtt{D}}}{\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{new}\ \mathtt{C}_0\mathtt{(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{C}_0} \qquad \text{(T-}\textsc{New}\text{)}$$

$$\frac{\begin{array}{c}\mathit{requires}(\mathtt{L}) = \Lambda \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda \setminus \{\mathtt{L}\}, \Lambda) = \overline{\mathtt{D}} \rightarrow \mathtt{D}_0 \\ \mathtt{L.C.m}; \Lambda; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} \qquad \overline{\mathtt{E}}\ \mathtt{<:}\ \overline{\mathtt{D}}\end{array}}{\mathtt{L.C.m}; \Lambda; \Gamma \vdash \mathtt{proceed(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{D}_0}$$

$$\text{(T-}\textsc{Proceed}\text{)}$$

$$\frac{\begin{array}{c}\overline{\mathtt{L}}' \text{ is a prefix of } \overline{\mathtt{L}}'' \qquad \mathtt{C}_0\ \mathtt{<:}\ \mathtt{D} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{D}, \{\overline{\mathtt{L}}'\}, \{\overline{\mathtt{L}}''\}) = \overline{\mathtt{F}} \rightarrow \mathtt{F}_0 \\ \mathcal{L}; \Lambda; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} \qquad \overline{\mathtt{E}}\ \mathtt{<:}\ \overline{\mathtt{F}} \qquad \mathcal{L}; \Lambda; \Gamma \vdash \mathtt{v}_0 : \mathtt{C}_0\end{array}}{\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{v}_0\mathtt{<D,}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}''\mathtt{>.m(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{F}_0}$$

$$\text{(T-}\textsc{InvkA}\text{)}$$

Figure 10: ContextFJ^a: Expression typing.

$$\frac{\texttt{C.m}; \emptyset; \overline{\texttt{x}} : \overline{\texttt{C}}, \texttt{this} : \texttt{C} \vdash \texttt{e}_0 : \texttt{D}_0 \qquad \texttt{D}_0 \texttt{ <: } \texttt{C}_0}{\texttt{C}_0\ \texttt{m(}\overline{\texttt{C}}\ \overline{\texttt{x}}\texttt{)}\ \{\ \texttt{return e}_0\texttt{;}\ \}\ \textit{ok in}\ \texttt{C}} \qquad \text{(T-}\textsc{Method}\text{)}$$

$$\frac{\Lambda = \textit{requires}(\texttt{L}) \qquad \texttt{L.C.m}; \Lambda; \overline{\texttt{x}} : \overline{\texttt{C}}, \texttt{this} : \texttt{C} \vdash \texttt{e}_0 : \texttt{D}_0 \qquad \texttt{D}_0 \texttt{ <: } \texttt{C}_0}{\texttt{C}_0\ \texttt{m(}\overline{\texttt{C}}\ \overline{\texttt{x}}\texttt{)}\ \{\ \texttt{return e}_0\texttt{;}\ \}\ \textit{ok in}\ \texttt{L.C}} \qquad \text{(T-P}\textsc{Method}\text{)}$$

$$\frac{\texttt{K} = \texttt{C(}\overline{\texttt{D}}\ \overline{\texttt{g}}\texttt{,}\ \overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{)\{ super(}\overline{\texttt{g}}\texttt{); this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{; \}} \qquad \textit{fields}(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}} \qquad \overline{\texttt{M}}\ \textit{ok in}\ \texttt{C}}{\texttt{class C} \lhd \texttt{D}\ \{\ \overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{;}\ \texttt{K}\ \overline{\texttt{M}}\ \}\ \textit{ok}} \qquad \text{(T-}\textsc{Class}\text{)}$$

Figure 11: Method and class typing.

third argument to *mtype* is $\Lambda \setminus \{\texttt{L}\}$, which means that a `proceed` call cannot proceed to the same method recursively.

To prove type soundness, we also need to provide a typing rule for expressions that appear only at runtime, i.e., method invocation on an object with a cursor. This rule is provided by T-InvkA, which is basically a combination of T-Invk and T-New. In this rule, the cursor information, $\texttt{D}$, $\overline{\texttt{L}}'$, and $\overline{\texttt{L}}''$, is used to lookup the type of `m`, instead of the receiver's runtime class $\texttt{C}_0$.

The typing rules for methods and classes are shown in Figure 11. These rules are almost identical to those in ContextFJ. A type judgment for methods in a base class is of the form `M` *ok in* `C`, read as "method `M` is well-formed in `C`." A type judgment for partial methods is of the form `M` *ok in* `L.C`, read as "method `M` is well-formed in `L` in `C`." These judgments are represented by typing rules T-Method and T-PMethod, respectively, which are straightforward. Both rules check that the method body is well-typed under the given type environment, and the type of the method body is a subtype of the declared return type. For partial methods, the layers Λ that the current layer `L` requires can be assumed, as well as the current layer itself. Similar to ContextFJ, valid method overriding is not checked in this case because it requires that the entire program be checked.

$$\frac{\begin{array}{l}\forall \mathtt{m}.\ \text{if } CT(\mathtt{C}) = \mathtt{class\ C} \lhd \mathtt{D}\ \{\cdots\ \mathtt{C_0\ m(\overline{C}\ \overline{x})}\{\cdots\}\ \cdots\} \text{ and } PT(\mathtt{m},\mathtt{C},\mathtt{L}) = \mathtt{D_0\ m(\overline{D}\ \overline{y})}\{\cdots\}, \\ \qquad \text{then } \overline{\mathtt{C}},\mathtt{C_0} = \overline{\mathtt{D}},\mathtt{D_0}\end{array}}{\mathit{override}^h(\mathtt{L},\mathtt{C})}$$

$$\frac{\begin{array}{l}\forall \mathtt{m}.\ \text{if } \mathit{mtype}(\mathtt{m},\mathtt{C},\mathit{dom}(PT)) = \overline{\mathtt{C}} \rightarrow \mathtt{C_0} \text{ and } \mathit{mtype}(\mathtt{m},\mathtt{D},\mathit{dom}(PT)) = \overline{\mathtt{D}} \rightarrow \mathtt{D_0} \text{ and } \mathtt{C} \mathrel{\mathtt{<:}} \mathtt{D}, \\ \qquad \text{then } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ and } \mathtt{C_0} \mathrel{\mathtt{<:}} \mathtt{D_0}\end{array}}{\mathit{override}^v(\mathtt{C},\mathtt{D})}$$

Figure 12: Valid overriding.

A class is well-formed (written `CL` *ok*) when the constructor matches the field declarations and all methods are well-formed.

A program is well-formed when all classes are well-formed, all partial methods are well-formed, no layers in $\mathcal{D}$ are required by other layers, and the main expression is well-typed under the empty assumption, as expressed in the following rule (T-Prog):

$$\frac{\begin{array}{c}\forall \mathtt{C} \in \mathit{dom}(CT).CT(\mathtt{C})\ \mathit{ok} \qquad \forall (\mathtt{m},\mathtt{C},\mathtt{L}) \in \mathit{dom}(PT).PT(\mathtt{m},\mathtt{C},\mathtt{L})\ \mathit{ok\ in}\ \mathtt{L.C} \\ \bullet;\emptyset \vdash \mathtt{e} : \mathtt{C} \qquad \forall (\mathtt{L_1},\mathtt{L_2}) \in \mathcal{R}.\mathtt{L_2} \notin \mathcal{D} \\ \forall \mathtt{C} \in \mathit{dom}(CT), \mathtt{L} \in \mathit{dom}(PT).\mathit{override}^h(\mathtt{L},\mathtt{C}) \qquad \forall \mathtt{C},\mathtt{D} \in \mathit{dom}(CT).\mathit{override}^v(\mathtt{C},\mathtt{D})\end{array}}{\vdash (CT, PT, \mathcal{R}, \mathcal{D}, \mathtt{e}) : \mathtt{C}} \quad \text{(T-Prog)}$$

The other conditions used in T-Prog are shown in Figure 12. The predicate $\mathit{override}^h(\mathtt{L},\mathtt{C})$ (h represents "horizontally") means that all overriding partial methods in `L` have the same signatures as the corresponding methods in `C`. The predicate $\mathit{override}^v(\mathtt{C},\mathtt{D})$ (v represents "vertically") means that a method overridden by a subclass `C` of `D` is valid in a sense that it checks the covariant method overriding.

5.4. Properties

It is easy to prove that the deactivation of layer $\texttt{L}$ annotated with `ensureDeactivate` is ensured, which means that if $\texttt{L}$ is not in the sequence of currently activated layers $\overline{\texttt{L}}$, it is not searched during the method lookup, i.e., $\texttt{L}$ is not included in the cursor for the method lookup. This property is formalized as the following theorem (we say that $\mathcal{R}$ is well-formed with respect to $\mathcal{D}$ if $\forall(\texttt{L}_1, \texttt{L}_2) \in \mathcal{R}.\texttt{L}_2 \notin \mathcal{D}$):

Theorem 1 (Guarantee of Deactivation)**.** *Suppose given class and partial method tables are well-formed, and $\mathcal{R}$ is well-formed with respect to $\mathcal{D}$. If $\mathcal{L}; \Lambda; \Gamma \vdash$ $\texttt{v.m}(\overline{\texttt{v}})\texttt{@L}_0 : \texttt{C}$ and $\texttt{v.m}(\overline{\texttt{v}})\texttt{@L}_0 \mid \overline{\texttt{L}} \longrightarrow \texttt{e} \mid \overline{\texttt{L}}''$, then $\texttt{e} = \texttt{v.m}(\overline{\texttt{v}})\texttt{@L}_0$, or $\overline{\texttt{L}}'' = \overline{\texttt{L}}$ and $\texttt{e} = \texttt{v<C,}\overline{\texttt{L}}'\texttt{,}\overline{\texttt{L}}'\texttt{>.m}(\overline{\texttt{v}})$ for some $\overline{\texttt{L}}'$ and $\texttt{L} \notin \overline{\texttt{L}}'$ for all $\texttt{L} \in \mathcal{D} \setminus (\overline{\texttt{L}}; \texttt{L}_0)$.*

We sketch the proof of this theorem, which is given by case analysis on the last used reduction rule. There are two cases, namely, R-Invk and R-Activate, and the latter case is trivial. In case R-Invk, since $\forall(\texttt{L}_1, \texttt{L}_2) \in \mathcal{R}.\texttt{L}_2 \notin \mathcal{D}$, it is easy to show that $\forall\texttt{L} \in \mathcal{D}.\texttt{L} \notin \mathit{requires}(\texttt{L}_0)$. Then, by the definition of *filter*, it is obvious that $\texttt{L} \notin \overline{\texttt{L}}'$, finishing the case.

We note that the proposed calculus does not ensure deactivation *after setting the cursor*. For example, it is possible that a layer with `ensureDeactivate`, namely $\texttt{L}'$, is removed from the currently activated layers $\overline{\texttt{L}}$ by R-Activate, immediately after R-Invk is applied. Then, $\texttt{L}'$ is still included in the cursor and searched during the method lookup. This ensures compatibility with the original ContextFJ. The `ensureDeactivate` mechanism prevents the layer that is not activated *when the method lookup starts* from being searched during the method lookup.

We show the type soundness for ContextFJ^a. The primary difference from ContextFJ is that while ContextFJ requires that the sequence of activated layers $\overline{\texttt{L}}$ is well-formed (i.e., for all layers $\texttt{L}_i$ in $\overline{\texttt{L}}$, all layers required by $\texttt{L}_i$ exist on the left hand side of $\texttt{L}_i$ in $\overline{\texttt{L}}$ [10]), ContextFJ^a does not impose such a restriction. Instead, ContextFJ^a requires that the sequence of layers constructed by the R-Invk rule is well-formed. Proofs are given in Appendix A.

Theorem 2 (Subject Reduction)**.** *Suppose given class and partial method tables are well-formed. If* $\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e} : \mathtt{C}$ *and* $\mathtt{e} \mid \overline{\mathtt{L}} \longrightarrow \mathtt{e}' \mid \overline{\mathtt{L}}'$ *for some* $\overline{\mathtt{L}}$*,* $\overline{\mathtt{L}}'$*, and* Λ*, then* $\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e}' : \mathtt{D}$ *for some* D *such that* D <: C*.*

Theorem 3 (Progress)**.** *Suppose given class and partial method tables are well-formed. If* $\bullet; \Lambda; \bullet \vdash \mathtt{e} : \mathtt{C}$ *for some* Λ*, then either* e *is a value or* $\mathtt{e} \mid \overline{\mathtt{L}} \longrightarrow \mathtt{e}' \mid \overline{\mathtt{L}}$ *for some* e′ *and* $\overline{\mathtt{L}}$*.*

Theorem 4 (Type Soundness)**.** *If* $\vdash (CT, PT, \mathcal{R}, \mathcal{D}, \mathtt{e}) : \mathtt{C}$ *and* e *reduces to a normal form, then* e *is* new C($\overline{\mathtt{v}}$) *for some* $\overline{\mathtt{v}}$ *and* D *such that* D <: C*.*

6. Implementation

We implemented the proposed mechanism in ServalCJ, a COP language with a generalized layer activation mechanism [17]. The ServalCJ compiler is built on top of the AspectBench Compiler [21] by extending the front-end. This compiler translates a ServalCJ program into an AspectJ program, and the proposed mechanism is straightforwardly built on top of this translation mechanism. Thus, we first overview this translation briefly; then, we explain the implementation of the proposed mechanism.

The ServalCJ compiler comprises the following two parts: translation of layers and translation of layer activation. As the proposed mechanism addresses only method dispatch, which is relevant to the implementation of layers, we explain only the first part. A layer in ServalCJ comprises a partial definition of classes, and each partial definition is translated into an inner class of the class enclosed by this layer. Each partial method in this layer is translated into a method in the inner class. The base class is also translated to realize layer-based method dispatch. First, each base class is equipped with a new field to store the list of currently activated layers (a list of instances of the inner classes translated from the layers). Second, for each partial method, the body of the method in the base class is translated to the code that first obtains the list of currently activated layers, and then calls the instance method at the tail position of the list. In addition, the proceed call is translated to the code that calls the method on the instance at the preceding position of the list of currently activated layers.

Based on this translation, the implementation of the proposed mechanism consists of two parts, i.e., embedding the static scope used in the method lookup in Java and removing duplicated partial method calls. Instead of collecting all required methods and removing the duplicated partial method calls prior to the method dispatch, which may result in significant runtime overhead, embedding the static scope is realized by translating the requires relation to the extends relation in Java. For example, layers Debugging and Programming are translated to the classes that implement these layers, by the original ServalCJ translation mechanism. However, if Debugging requires Programming, the class translated from Debugging (_Layer$Debugging) extends the class translated from Programming. When Debugging is activated, the method lookup is performed on an instance of _Layer$Debugging. This embedding preserves the semantics, because the scope of method lookup is represented as the runtime type of the method receiver.

Note that this translation ensures that there are no conflicts between an inheritance relation written in the source code and that introduced by the translation. First, each layer cannot introduce new "extends" declarations for the existing classes. Second, in ServalCJ, no inheritance relations exist between layers.

We describe the manner in which the proposed implementation prevents duplication of partial method calls using an example scenario where a method is called when Programming and Debugging are activated, and Debugging requires Programming (Figure 13). First, the proposed implementation calls the method on the tail of the sequence of activated layers (1). This method call invokes the method on class _Layer$Debugging that is translated from layer Debugging. Then, it executes the code that is copied from the partial method, and calls the method on the superclass (2). This superclass, _Layer$Programming, encodes layer Programming that is required by Debugging. In the superclass method, it first records that this superclass method has been called. Then, it executes

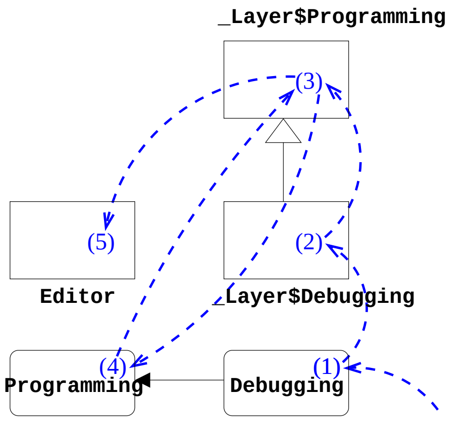

Figure 13: Example of the chain of partial method lookups. Rectangles denote classes, and rounded rectangles denote activated layers.

the code translated from the partial method in `Programming` (3). Then, it calls the method on the next instance of the sequence of activated layers, which results in the invocation of the method in `_Layer$Programming` (4). Because the invocation of this method call had been recorded, it skips the main code to prevent duplication, and calls the method of the base class because there are no layers after `Programming` (5).

We note a case where a layer requires multiple layers. Even though the current implementation does not enable a layer to require multiple layers, this feature can be implemented by applying the translation mechanism from mixin-based inheritance into the Java class hierarchy. For example, McJava [22] addresses this issue by linearizing the mixin-based inheritance to form a single inheritance chain, and duplicating the definitions of mixins into several class hierarchies induced by a mixin composition.

7. Related Work

7.1. Layer-Introduced Base Method

We first compare the proposed approach with other existing approaches that provide layer-introduced base methods.

7.1.1. ContextFJ

We reviewed the ContextFJ approach in Section 2.2. A problem in this approach is that it does not interact suitably with dynamic layer deactivation. ContextFJ does not support `without`, a language construct for dynamic layer deactivation in ContextJ. If layers can be deactivated dynamically, the invocation of a method introduced by the deactivated layer fails when the layer that depends on the deactivated layer calls this method. To statically check such an error, we need to gather information about "which layer is absent" at each deactivation point, which is difficult, particularly in an open-world setting. The proposed mechanism is a simple means to support layer-introduced base methods in COP languages with dynamic and asynchronous layer deactivation in a type-safe manner.

Nevertheless, we do not argue that `requires` in ContextFJ should be replaced with the mechanism. In particular, `requires` in ContextFJ is useful if we would like to require the *interface* of layers; i.e., we may extend ContextFJ so as to assume that at least one of the layers providing this interface is activated without considering the concrete implementation. With this extension, we may also write the `requires` clause like "`requires LayerA` or `LayerB`." This requiring of layer interfaces is hard to achieve in our setting, because the required layers are used for the lookup of the method body.

7.1.2. On-Demand Activation

On-demand activation [12] has been proposed to prevent checking an absent layer when `requires` is used with `without`. Instead of requiring that the subordinate layers are activated, it implicitly activates the layers on which the currently activated layer depends when these layers are required, as specified by the following `activates` clause:

```
layer Debugging activates Programming {
  /* The body is the same as above */ }
```

We can activate Debugging anywhere, regardless of the condition that this activation is enclosed with the activation of Programming; if Programming is not active, it is implicitly activated when the currently executing behavior requires it.

This mechanism implicitly activates the layer specified by requires when this layer is asynchronously deactivated. However, this mechanism fails if the currently executing layer is deactivated, which is explained in the following code:

```
layer L {
  class C {
    Object m() {  // deactivating L
      return this.n(); }
    Object n() {  // introduced by L
      return new Object(); }
  }
}
```

Class C in layer L defines partial methods m and n, and m calls n, which is introduced by L. As asynchronous layer deactivation may occur at any execution point in the on-demand activation mechanism, it is possible that L is deactivated immediately before n is called. Then, the method lookup for n is performed without the existence of L, leading to a method-not-understood error.

7.1.3. Layer Inheritance

The proposed approach is very similar to the layer inheritance approach; i.e., the requires relations resemble the extends relations between layers, where requires is implemented using extends as described previously. JCop [4] supports such a layer inheritance mechanism. The proposed mechanism is a simplified unification of requires in ContextFJ and extends in JCop, both of which call layer-introduced base methods.

There is a fundamental difference between the proposed approach and JCop. In JCop, layers are instantiated, and not directly activated; however, instances of layers are. If instance `p1` of `Programming` and instance `d1` of `Debugging` are activated, the partial method defined in `Programming` is executed individually for `p1` and `d1`, which results in the same menu items being displayed multiple times for the programming feature. On the other hand, the proposed approach is based on the COP model, where a layer is not a first-class citizen and prohibits such a duplication.

7.2. Other Related Work

Inoue et al. discussed a safe type system for JCop [23, 11]. It constitutes an application of the type system developed in ContextFJ; however, it supports layer inheritance and first-class layers. It does not handle layer deactivation directly; instead, it supports an idiom, which is referred to as layer swapping, for layer deactivation. It is prohibited to deactivate layers using `without`; however, we may "swap" a layer with one that is compatible with the swapped layer.

The dependency between layers may also be represented in the form of *composite layers* [24, 25]. In [24], an extension of ContextL [2] with layer composition operators, such as and-composition and or-composition, was proposed. At each layer activation point, ContextL calculates the set of subordinate layers and activates them. If this set is ambiguous, it suspends the execution until the user resolves this ambiguity. In [25], a similar mechanism is discussed under event-based layer transition [5]. FECJ° [26] formalizes the operational semantics of composite layers implemented in EventCJ [5]. The dependency between layers can also be specified in some COP languages such as Subjective-C [6] and Ambience [14]. In these languages, this dependency is checked at runtime.

The proposed mechanism is similar to Newspeak's method lookup [27] in that the static scope is used for method lookup. While Newspeak uses the static scope to resolve method collisions between the outer class and super classes, the proposed mechanism uses the static scope (and the definitions on which this scope depends) as a safety net, ensuring that there is a behavior while executing the current method, even when the enclosing layer is immediately deactivated. Furthermore, the proposed method lookup includes dynamically activated layers, and we need to define appropriate ordering of these activated layers, the layer comprising the static scope, the layers on which this scope depends, and the base class.

8. Concluding Remarks

This paper addressed the problems that occur when layers declare base methods that are used with asynchronous layer deactivation. This paper provides a formal definition of method lookup, which uses the static scope of a method invocation, proves its type soundness, and ensures deactivation of layers with `ensureDeactivate`. This mechanism is a type-safe application of an on-demand activation mechanism to asynchronous layer deactivation, which addresses the problem of ContextFJ regarding support of dynamic layer deactivation. This mechanism is also considered as an alternative to the layer inheritance mechanism, where duplicated calls of partial methods do not occur. This mechanism is implemented in our COP language, ServalCJ.

9. Acknowledgements

This work was supported by JSPS KAKENHI Grant Number 25330078.

References

[1] R. Hirschfeld, P. Costanza, O. Nierstrasz, Context-oriented programming, Journal of Object Technology 7 (3) (2008) 125–151.

[2] P. Costanza, R. Hirschfeld, Language constructs for context-oriented programming – an overview of ContextL, in: Dynamic Language Symposium (DLS) '05, 2005, pp. 1–10.

[3] M. Appeltauer, R. Hirschfeld, M. Haupt, H. Masuhara, ContextJ: Context-oriented programming with Java, Computer Software 28 (1) (2011) 272–292.

[4] M. Appeltauer, R. Hirschfeld, H. Masuhara, M. Haupt, K. Kawauchi, Event-specific software composition in context-oriented programming, in: Proceedings of the International Conference on Software Composition 2010 (SC'10), Vol. 6144 of LNCS, 2010, pp. 50–65.

[5] T. Kamina, T. Aotani, H. Masuhara, EventCJ: a context-oriented programming language with declarative event-based context transition, in: AOSD '11, 2011, pp. 253–264.

[6] S. González, M. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, J. Goffaux, Subjective-C: Bringing context to mobile platform programming, in: SLE'11, Vol. 6563 of LNCS, 2011, pp. 246–265.

[7] M. von Löwis, M. Denker, O. Nierstrasz, Context-oriented programming: beyond layers, in: ICDL '07: Proceedings of the 2007 International Conference on Dynamic languages, 2007, pp. 143–156.

[8] G. Salvaneschi, C. Ghezzi, M. Pradella, ContextErlang: Introducing context-oriented programming in the actor model, in: AOSD'12, 2012.

[9] J. Lincke, M. Appeltauer, B. Steinert, R. Hirschfeld, An open implementation for context-oriented layer composition in ContextJS, Science of Computer Programming 76 (12) (2011) 1194–1209.

[10] A. Igarashi, R. Hirschfeld, H. Masuhara, A type system for dynamic layer composition, in: FOOL'12, 2012.

[11] H. Inoue, A. Igarashi, A sound type system for layer subtyping and dynamically activated first-class layers, in: APLAS'15, 2015.

[12] T. Kamina, T. Aotani, A. Igarashi, On-demand layer activation for type-safe deactivation, in: COP'14, 2014.

[13] E. Bainomugisha, J. Vallejos, C. D. Roover, A. L. Carreton, W. D. Meuter, Interruptible context-dependent executions: A fresh look at programming context-aware applications, in: Onward! 2012, 2012, pp. 67–84.

[14] S. González, K. Mens, A. Cádiz, Context-oriented programming with the ambient object systems, Journal of Universal Computer Science 14 (20) (2008) 3307–3332.

[15] T. Kamina, T. Aotani, H. Masuhara, A. Igarashi, Method safety mechanism for asynchronous layer deactivation, in: COP'15, 2015.

[16] B. Desmet, J. Vallejos, P. Costanza, R. Hirschfeld, Layered design approach for context-aware systems, in: VaMoS'07, 2007.

[17] T. Kamina, T. Aotani, H. Masuhara, Generalized layer activation mechanism through contexts and subscribers, in: MODULARITY'15, 2015, pp. 14–28.

[18] M. Appeltauer, R. Hirschfeld, H. Masuhara, Improving the development of context-dependent Java application with ContextJ, in: COP'09, 2009.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Grisword, An overview of AspectJ, in: ECOOP'01, 2001, pp. 327–353.

[20] A. Igarashi, B. Pierce, P. Wadler, Featherweight Java: A minimal core calculus for Java and GJ, ACM TOPLAS 23 (3) (2001) 396–450.

[21] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, abc: an extensible AspectJ compiler, in: AOSD'05, 2005, pp. 87–98.

[22] T. Kamina, T. Tamai, McJava – a design and implementation of Java with mixin-types, in: 2nd ASIAN Symposium on Programming Languages and Systems (APLAS 2004), Vol. 3302 of LNCS, Springer, 2004, pp. 398–414.

[23] H. Inoue, A. Igarashi, M. Appeltauer, R. Hirschfeld, Towards type-safe JCop: A type system for layer inheritance and first-class layers, in: COP'14, 2014.

[24] P. Costanza, T. D'Hondt, Feature descriptions for context-oriented programming, in: 2nd International Workshop on Dynamic Software Product Lines (DSPL'08), 2008.

[25] T. Kamina, T. Aotani, H. Masuhara, Introducing composite layers in EventCJ, IPSJ Transactions on Programming 6 (1) (2013) 1–8.

[26] T. Kamina, T. Aotani, H. Masuhara, A core calculus of composite layers, in: FOAL'13, 2013, pp. 7–12.

[27] G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, E. Miranda, Modules as objects in Newspeak, in: ECOOP'10, 2010, pp. 405–428.

Appendix A. Proofs

We first show the lemmas required in the proof of Theorem 2.

Lemma 1 (Weakening)**.** *If* $\mathcal{L};\Lambda;\Gamma \vdash \mathtt{e} : \mathtt{C}$ *and* $\mathtt{x} \notin \Gamma$*, then* $\mathcal{L};\Lambda;\Gamma,\mathtt{x} : \mathtt{D} \vdash \mathtt{e} : \mathtt{C}$*.*

Proof. By straightforward induction on $\mathcal{L};\Lambda;\Gamma \vdash \mathtt{e} : \mathtt{C}$. □

Lemma 2. *If fields*$(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$ *and* $\mathtt{D}$ `<:` $\mathtt{C}$*, then fields*$(\mathtt{D}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}, \overline{\mathtt{D}}\ \overline{\mathtt{g}}$ *for some* $\overline{\mathtt{D}}$ *and* $\overline{\mathtt{g}}$*.*

Proof. By straightforward induction on $\mathtt{D}$ `<:` $\mathtt{C}$. □

Lemma 3. *If mtype*$(\mathtt{m},\mathtt{C},\Lambda) = \overline{\mathtt{D}} \rightarrow \mathtt{D}_0$ *and* $\mathtt{D}$ `<:` $\mathtt{C}$*, then mtype*$(\mathtt{m},\mathtt{D},\Lambda) = \overline{\mathtt{D}} \rightarrow \mathtt{E}_0$ *and* $\mathtt{E}_0$ `<:` $\mathtt{D}_0$ *for some* $\mathtt{E}_0$*.*

Proof. By induction on $\mathtt{D}$ `<:` $\mathtt{C}$. □

Lemma 4 (Substitution)**.** *If* $\mathcal{L};\Lambda;\Gamma,\overline{\mathtt{x}} : \overline{\mathtt{C}} \vdash \mathtt{e}_0 : \mathtt{C}_0$ *and* $\mathcal{L};\Lambda;\Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{D}}$ *and* $\overline{\mathtt{D}}$ `<:` $\overline{\mathtt{C}}$*, then* $\mathcal{L};\Lambda;\Gamma \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{D}_0$ *and* $\mathtt{D}_0$ `<:` $\mathtt{C}_0$ *for some* $\mathtt{D}_0$*.*

Proof. By induction on $\mathcal{L};\Lambda;\Gamma,\overline{\mathtt{x}} : \overline{\mathtt{C}} \vdash \mathtt{e}_0 : \mathtt{C}_0$.
Case T-Var: Immediate by Lemma 1.
Case T-Field:

$$\mathtt{e}_0 = \mathtt{e.f}_i \qquad \mathcal{L};\Lambda;\Gamma,\overline{\mathtt{e}}:\overline{\mathtt{C}} \vdash \mathtt{e}:\mathtt{C} \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{E}}\ \overline{\mathtt{f}} \qquad \mathtt{C}_0 = \mathtt{E}_i$$

By the induction hypothesis, $\mathcal{L};\Lambda;\Gamma \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}]\mathtt{e} : \mathtt{C}'$ and $\mathtt{C}'$ `<:` $\mathtt{C}$. By Lemma 2, $\mathit{fields}(\mathtt{C}') = \overline{\mathtt{E}}\ \overline{\mathtt{f}}, \overline{\mathtt{F}}\ \overline{\mathtt{g}}$ for some $\overline{\mathtt{F}}$ and $\overline{\mathtt{g}}$. Then, by T-Field, $\mathcal{L};\Lambda;\Gamma \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{E}_i$, finishing the case.

Case T-Invk:

$$\mathtt{e}_0 = \mathtt{e.m}(\overline{\mathtt{e}})\mathtt{@X} \qquad \mathcal{L};\Lambda;\Gamma,\overline{\mathtt{x}}:\overline{\mathtt{C}} \vdash \mathtt{e}:\mathtt{C} \qquad \Lambda = \mathit{requires}(\mathtt{X})$$
$$\mathit{mtype}(\mathtt{m},\mathtt{C},\Lambda) = \overline{\mathtt{E}} \to \mathtt{C}_0 \qquad \mathcal{L};\Lambda;\Gamma,\overline{\mathtt{x}}:\overline{\mathtt{C}} \vdash \overline{\mathtt{e}}:\overline{\mathtt{F}} \qquad \overline{\mathtt{F}} \mathrel{\mathtt{<:}} \overline{\mathtt{E}}$$

By the induction hypothesis, $\mathcal{L};\Lambda;\Gamma \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}]\mathtt{e} : \mathtt{C}'$, $\mathtt{C}' \mathrel{\mathtt{<:}} \mathtt{C}$ and $\mathcal{L};\Lambda;\Gamma \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}]\overline{\mathtt{e}} : \overline{\mathtt{F}}'$, $\overline{\mathtt{F}}' \mathrel{\mathtt{<:}} \overline{\mathtt{F}}$. By Lemma 3, $\mathit{mtype}(\mathtt{m},\mathtt{C}',\Lambda) = \overline{\mathtt{E}} \to \mathtt{E}_0$ and $\mathtt{E}_0 \mathrel{\mathtt{<:}} \mathtt{C}_0$. By T-Invk, $\mathcal{L};\Lambda;\Gamma \vdash \mathtt{e}_0 : \mathtt{E}_0$, finishing the case.

Cases T-New, T-Proceed, and T-InvkA: Immediate by the induction hypothesis. □

Lemma 5. *If $\Lambda_1 \subset \Lambda_2$ and $\mathit{mtype}(\mathtt{m},\mathtt{C}_0,\Lambda_1) = \overline{\mathtt{D}} \to \mathtt{D}_0$, then $\mathit{mtype}(\mathtt{m},\mathtt{C}_0,\Lambda_2) = \overline{\mathtt{D}} \to \mathtt{D}_0$.*

Proof. By induction on $\{\overline{\mathtt{L}}\}$. The base case, which is $\Lambda_1 = \Lambda_2$, is trivial. If $\Lambda_2 = \{\mathtt{L}'\} \cup \{\overline{\mathtt{L}}'\}$ and $\{\overline{\mathtt{L}}'\} \supset \Lambda_1$, by the induction hypothesis, $\mathit{mtype}(\mathtt{m},\mathtt{C}_0,\{\overline{\mathtt{L}}'\}) = \overline{\mathtt{D}} \to \mathtt{D}_0$. Then, by the rules in Figure 12, we have $\mathit{mtype}(\mathtt{m},\mathtt{C}_0,\{\mathtt{L}'\} \cup \{\overline{\mathtt{L}}'\}) = \overline{\mathtt{D}} \to \mathtt{D}_0$. □

Lemma 6. *Suppose $\overline{\mathtt{L}}'$ is a prefix of $\overline{\mathtt{L}}''$ and $\mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}',\overline{\mathtt{L}}'') = \overline{\mathtt{x}}.\mathtt{e}_0$ `in` $\mathtt{C}',\overline{\mathtt{L}}$ and $\mathit{mtype}(\mathtt{m},\mathtt{C},\{\overline{\mathtt{L}}'\},\{\overline{\mathtt{L}}''\}) = \overline{\mathtt{D}} \to \mathtt{D}_0$.*

1. *If $\overline{\mathtt{L}} = \overline{\mathtt{L}}''';\mathtt{L}_0$, then $\mathtt{L}_0.\mathtt{C}'.\mathtt{m};\{\overline{\mathtt{L}}''\};\overline{\mathtt{x}}:\overline{\mathtt{D}},\mathtt{this}:\mathtt{C}' \vdash \mathtt{e}_0 : \mathtt{E}_0$ and $\mathtt{C} \mathrel{\mathtt{<:}} \mathtt{C}'$ and $\mathtt{E}_0 \mathrel{\mathtt{<:}} \mathtt{D}_0$ for some $\mathtt{E}_0$.*
2. *If $\overline{\mathtt{L}} = \bullet$, then $\mathtt{C}'.\mathtt{m};\emptyset;\overline{\mathtt{x}}:\overline{\mathtt{D}},\mathtt{this}:\mathtt{C}' \vdash \mathtt{e}_0 : \mathtt{E}_0$ and $\mathtt{C} \mathrel{\mathtt{<:}} \mathtt{C}'$ and $\mathtt{E}_0 \mathrel{\mathtt{<:}} \mathtt{D}_0$ for some $\mathtt{E}_0$.*

Proof. By induction on $\mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}',\overline{\mathtt{L}}'') = \overline{\mathtt{x}}.\mathtt{e}_0$ `in` $\mathtt{C}',\overline{\mathtt{L}}$.

Case MB-Class:

$$\overline{\mathtt{L}}' = \bullet \qquad \overline{\mathtt{L}} = \bullet \qquad \mathtt{C}' = \mathtt{C}$$
$$\mathtt{class\ C} \lhd \mathtt{D}\ \{\ \cdots\ \mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return}\ \mathtt{e}_0;\ \}\ \cdots\ \}$$

By T-Class, T-Method, and MT-Class, it must be the case that $\mathtt{C.m}; \emptyset; \overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{E}_0$, $\overline{\mathtt{C}} = \overline{\mathtt{D}}$, $\mathtt{E}_0$ `<:` $\mathtt{C}_0$, and $\mathtt{C}_0 = \mathtt{D}_0$ for some $\mathtt{E}_0$, finishing the case.

Case MB-Layer:

$$\overline{\mathtt{L}}' = \overline{\mathtt{L}}'''; \mathtt{L}_0 \qquad \mathtt{C} = \mathtt{C}' \qquad \overline{\mathtt{L}} = \overline{\mathtt{L}}'$$
$$PT(\mathtt{m}, \mathtt{C}, \mathtt{L}_0) = \mathtt{C}_0\ \mathtt{m(}\overline{\mathtt{C}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e_0;\ \}}$$

By T-PMethod and MT-PMethod, it must be the case that $\mathtt{L}_0\mathtt{.C.m}; \Lambda; \overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{E}_0$ where $\Lambda = \mathit{requires}(\mathtt{L}_0)$, $\overline{\mathtt{C}} = \overline{\mathtt{D}}$, $\mathtt{E}_0$ `<:` $\mathtt{D}_0$, and $\mathtt{C}_0 = \mathtt{D}_0$ for some $\mathtt{E}_0$. It is easy to show that $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \{\overline{\mathtt{L}}'\}, \{\overline{\mathtt{L}}''\}) = \mathit{mtype}(\mathtt{m}, \mathtt{C}, \{\overline{\mathtt{L}}'\}, \Lambda)$ because $\Lambda \subset \{\overline{\mathtt{L}}''\}$, finishing the case.

Cases MB-Super and MB-NextLayer: The induction hypothesis finishes the cases. □

Lemma 7 (Substitution for `proceed`). *If* $\overline{\mathtt{L}}'; \mathtt{L}$ *is a prefix of* $\overline{\mathtt{L}}$ *and* $\Lambda = \{\overline{\mathtt{L}}\}$ *and* $\mathtt{L.C.m}; \Lambda; \Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0$ *and* $\mathtt{D}_0$ `<:` `C` *and* $\mathit{fields}(\mathtt{D}_0) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}$ *and* $\bullet; \Lambda; \Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{E}}$ *and* $\overline{\mathtt{E}}$ `<:` $\overline{\mathtt{D}}$*, then* $\bullet; \Lambda; \Gamma \vdash [\mathtt{new\ D_0(}\overline{\mathtt{v}}\mathtt{)<C,}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m/proceed}]\mathtt{e}_0 : \mathtt{C}_0$.

Proof. By induction on $\mathtt{L.C.m}; \Lambda; \Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0$ with case analysis on the last typing rule used. We show only the main case below.

Case T-Proceed:

$$\mathtt{e}_0 = \mathtt{proceed(}\overline{\mathtt{e}}\mathtt{)} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda \setminus \{\mathtt{L}\}, \Lambda) = \overline{\mathtt{F}} \to \mathtt{C}_0$$
$$\mathtt{L.C.m}; \Lambda; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{G}} \qquad \overline{\mathtt{G}}\mathtt{<:}\overline{\mathtt{F}} \qquad \mathit{requires}(\mathtt{L}) = \Lambda$$

Let $S = [\mathtt{new\ D_0(}\overline{\mathtt{v}}\mathtt{)<C,}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m/proceed}]$. In this case,

$$S\mathtt{e}_0 : \mathtt{C}_0 = \mathtt{new\ D_0(}\overline{\mathtt{v}}\mathtt{)<C,}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m(}S\overline{\mathtt{e}}\mathtt{)}.$$

As $\Lambda = \mathit{requires}(\mathtt{L}) = \{\overline{\mathtt{L}}\}$ and $\overline{\mathtt{L}}'; \mathtt{L}$ is a prefix of $\overline{\mathtt{L}}$, it is easy to show that

$$\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda \setminus \{\mathtt{L}\}, \Lambda) = \mathit{mtype}(\mathtt{m}, \mathtt{C}, \{\overline{\mathtt{L}}'\}, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{F}} \to \mathtt{C}_0.$$

By T-InvkA, T-New, and the induction hypothesis, we obtain

$$\bullet; \Lambda; \Gamma \vdash \mathtt{new\ D_0(}\overline{\mathtt{v}}\mathtt{)<C,}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m(}S\overline{\mathtt{e}}\mathtt{)} : \mathtt{C}_0.$$

□

Proof of Theorem 2. By induction on $\mathtt{e} \mid \overline{\mathtt{L}} \longrightarrow \mathtt{e}' \mid \overline{\mathtt{L}}'$ with case analysis on the last reduction rule used.

Case R-Field:

$$\mathtt{e} = \mathtt{new\ C_0(\overline{v}).f}_i \qquad \mathtt{e}' = \mathtt{v}_i \qquad \mathit{fields}(\mathtt{C_0}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$$

By T-Field and T-New, $\mathcal{L};\Lambda;\Gamma \vdash \mathtt{new\ C_0(\overline{v})} : \mathtt{C_0}$, $\mathtt{C} = \mathtt{C}_i$, $\mathcal{L};\Lambda;\Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{D}}$, and $\overline{\mathtt{D}}$ `<:` $\overline{\mathtt{C}}$, finishing the case.

Case R-Invk:

$$\mathtt{e} = \mathtt{new\ C_0(\overline{w}).m(\overline{v})@X} \qquad \mathtt{e}' = \mathtt{new\ C_0{<}C_0,\overline{L}'',\overline{L}''{>}.m(\overline{v})}$$
$$\overline{\mathtt{L}}'' = \mathit{filter}(\overline{\mathtt{L}}; \mathit{requires}(\mathtt{X}))$$

Without loss of generality, we can let $\Lambda = \mathit{requires}(\mathtt{X})$. By T-Invk and T-New, $\mathcal{L};\Lambda;\Gamma \vdash \mathtt{new\ C_0(\overline{w})} : \mathtt{C_0}$, $\mathit{mtype}(\mathtt{m}, \mathtt{C_0}, \Lambda) = \overline{\mathtt{D}} \rightarrow \mathtt{C}$, $\mathcal{L};\Lambda;\Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{E}}$, and $\overline{\mathtt{E}}$ `<:` $\overline{\mathtt{D}}$. By the definition of *filter*, $\{\overline{\mathtt{L}}''\} \supset \Lambda$. By Lemmas 3 and 5, $\mathit{mtype}(\mathtt{m}, \mathtt{C_0}, \{\overline{\mathtt{L}}''\}) = \overline{\mathtt{D}} \rightarrow \mathtt{C}$. By T-InvkA, $\mathcal{L};\Lambda;\Gamma \vdash \mathtt{e}' : \mathtt{C}$, finishing the case.

Case R-InvkP:

$$\mathtt{e} = \mathtt{new\ C_0(\overline{v}){<}D,\overline{L}'',\overline{L}{>}.m(\overline{w})}$$
$$\mathtt{e}' = \begin{bmatrix} \mathtt{new\ C_0(\overline{v})} & /\mathtt{this} \\ \overline{\mathtt{w}} & /\overline{\mathtt{x}} \\ \mathtt{new\ C_0(\overline{v}){<}C',\overline{L}''',\overline{L}'{>}.m} & /\mathtt{proceed} \end{bmatrix} \mathtt{e_0}$$
$$\mathit{mbody}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{L}}'', \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e_0}\ \mathtt{in}\ \mathtt{C}', (\overline{\mathtt{L}}'''; \mathtt{L_0})$$

By T-InvkA and T-New, $\overline{\mathtt{L}}''$ is a prefix of $\overline{\mathtt{L}}'$, $\mathcal{L};\Lambda;\Gamma \vdash \mathtt{new\ C_0(\overline{v})} : \mathtt{C_0}$, $\mathtt{C_0}$ `<:` $\mathtt{D}$, $\mathit{mtype}(\mathtt{m}, \mathtt{D}, \{\overline{\mathtt{L}}''\}, \{\overline{\mathtt{L}}'\}) = \overline{\mathtt{F}} \rightarrow \mathtt{C}$, $\mathcal{L};\Lambda;\Gamma \vdash \overline{\mathtt{w}} : \overline{\mathtt{E}}$, and $\overline{\mathtt{E}}$ `<:` $\overline{\mathtt{F}}$ for some Λ. Without loss of generality, we can let $\Lambda = \{\overline{\mathtt{L}}'\}$. By Lemma 6, $\mathtt{L_0.C'.m};\Lambda;\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{F}}, \mathtt{this} : \mathtt{D}' \vdash \mathtt{e_0} : \mathtt{E}$ and $\mathtt{E}$ `<:`$\mathtt{C}$ and $\mathtt{D}$ `<:` $\mathtt{D}'$ for some $\mathtt{E}$ and $\mathtt{D}'$. By Lemmas 4 and 7, $\mathcal{L};\Lambda;\Gamma \vdash \mathtt{e_0} : \mathtt{F}$ and $\mathtt{F}$ `<:` $\mathtt{E}$ for some $\mathtt{F}$, finishing the case.

Case R-InvkB: Similar to the case R-InvkP.

Case R-Activate: Trivial.

Cases RC-Field, RC-InvkRecv, RC-InvkArg, and RC-InvkNew: Immediate from the induction hypothesis.

The following lemma is required for the proof of Theorem 3.

Lemma 8. *If* $\mathcal{L}; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e.m(\overline{e})@L} : \mathtt{C}$ *and* $\mathcal{L}; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e} : \mathtt{C}_0$ *and* $\mathit{mtype}(\mathtt{m}, \mathtt{C}_0, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{D}} \rightarrow \mathtt{D}_0$ *where* $\overline{\mathtt{L}} = \mathit{requires}(\mathtt{L})$*, then there exist* $\overline{\mathtt{x}}$ *and* $\mathtt{e}_0$ *and* $\overline{\mathtt{L}}''$ *and* $\mathtt{C}'$ *(*$\neq$ $\mathtt{Object}$*) such that* $\mathit{mbody}(\mathtt{m}, \mathtt{C}_0, \overline{\mathtt{L}}, \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in}\ \mathtt{C}', \overline{\mathtt{L}}''$ *where* $\mathit{filter}(\overline{\mathtt{L}}'''; \overline{\mathtt{L}}) = \overline{\mathtt{L}}'$ *for some* $\overline{\mathtt{L}}'''$ *and the lengths of* $\overline{\mathtt{x}}$ *and* $\overline{\mathtt{D}}$ *are equal.*

Proof. By lexicographic induction on $\mathit{mtype}(\mathtt{m}, \mathtt{C}_0, \Lambda_1, \Lambda_2) = \overline{\mathtt{D}} \rightarrow \mathtt{D}_0$ and the length of $\overline{\mathtt{L}}$.

Case: $\overline{\mathtt{L}} = \bullet$

$\mathtt{class}\ \mathtt{C}_0 \vartriangleleft \mathtt{D}\ \{\cdots\ \mathtt{C}_0'\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ \mathtt{return}\ \mathtt{e}_0;\ \}\ \cdots\}$

By MT-Class, it must be the case that $\overline{\mathtt{D}}, \mathtt{D}_0 = \overline{\mathtt{C}}, \mathtt{C}_0'$ and the lengths of $\overline{\mathtt{C}}$ and $\overline{\mathtt{x}}$ are equal. Then, by MB-Class, $\mathit{mbody}(\mathtt{m}, \mathtt{C}_0, \bullet, \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in}\ \mathtt{C}_0, \bullet$.

Case: $\overline{\mathtt{L}} = \overline{\mathtt{L}}''''; \mathtt{L}_0$

$\mathit{PT}(\mathtt{m}, \mathtt{C}_0, \mathtt{L}_0) = \mathtt{E}_0\ \mathtt{m}(\overline{\mathtt{E}}\ \overline{\mathtt{x}})\{\ \mathtt{return}\ \mathtt{e}_0;\ \}$

By MT-PMethod, it must be the case that $\overline{\mathtt{E}}, \mathtt{E}_0 = \overline{\mathtt{D}}, \mathtt{D}_0$ and the lengths of $\overline{\mathtt{E}}$ and $\overline{\mathtt{x}}$ are equal. By MB-Layer, $\mathit{mbody}(\mathtt{m}, \mathtt{C}_0, \overline{\mathtt{L}}, \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in}\ \mathtt{C}_0, \overline{\mathtt{L}}$ where $\overline{\mathtt{L}}' = \mathit{filter}(\overline{\mathtt{L}}'''; \overline{\mathtt{L}})$.

Case: $\overline{\mathtt{L}} = \bullet$ $\quad$ $\mathtt{class}\ \mathtt{C}_0 \vartriangleleft \mathtt{D}\ \{\cdots\ \overline{\mathtt{M}}\ \}$

$\mathtt{m} \notin \overline{\mathtt{M}}$ $\quad$ $\Lambda_1 = \emptyset$

By MT-Super, we have $\mathit{mtype}(\mathtt{m}, \mathtt{D}, \emptyset, \Lambda_2) = \overline{\mathtt{D}} \rightarrow \mathtt{D}_0$. By the induction hypothesis, there exists $\overline{\mathtt{x}}$ and $\mathtt{e}_0$ and $\overline{\mathtt{L}}''$ and $\mathtt{C}'$ ($\neq$$\mathtt{Object}$) such that $\mathit{mbody}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{L}}, \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in}\ \mathtt{C}', \overline{\mathtt{L}''}$ and the lengths of $\overline{\mathtt{x}}$ and $\overline{\mathtt{D}}$ are equal. By MB-Super, $\mathit{mbody}(\mathtt{m}, \mathtt{C}_0, \bullet, \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in}\ \mathtt{C}', \overline{\mathtt{L}}''$, finishing the case.

Case: $\overline{\mathtt{L}} = \overline{\mathtt{L}}''''; \mathtt{L}_0$

$\mathit{PT}(\mathtt{m}, \mathtt{C}_0, \mathtt{L}_0)$ undefined

By the induction hypothesis, there exist $\overline{\mathtt{x}}$ and $\mathtt{e}_0$ and $\overline{\mathtt{L}}''$ and $\mathtt{C}'$ ($\neq$$\mathtt{Object}$) such that $\mathit{mbody}(\mathtt{m}, \mathtt{C}_0, \overline{\mathtt{L}}'''', \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in}\ \mathtt{C}', \overline{\mathtt{L}}''$ and the lengths of $\overline{\mathtt{x}}$ and $\overline{\mathtt{D}}$ are equal. By MB-NextLayer, $\mathit{mbody}(\mathtt{m}, \mathtt{C}_0, \overline{\mathtt{L}}, \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in}\ \mathtt{C}', \overline{\mathtt{L}}''$, finishing the case. □

Proof of Theorem 3. By induction on $\bullet; \Lambda; \bullet \vdash \mathtt{e} : \mathtt{C}$ with case analysis on the last typing rule used.

Cases T-Var and T-Proceed: Cannot occur.

Case T-Field:

$$\mathtt{e} = \mathtt{e}_0\mathtt{.f}_i \qquad \mathtt{C} = \mathtt{C}_i \qquad \bullet; \Lambda; \bullet \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$$

By the induction hypothesis, either $\mathtt{e}_0$ is a value or there exists $\mathtt{e}_0'$ such that $\mathtt{e}_0 \mid \overline{\mathtt{L}} \longrightarrow \mathtt{e}_0' \mid \overline{\mathtt{L}}$. In the second case, RC-Field finishes the case. In the first case, by T-New, we have $\mathtt{e}_0 = \mathtt{new}\ \mathtt{C}_0\mathtt{(}\overline{\mathtt{v}}\mathtt{)}$, $\mathcal{L}; \bullet \vdash \overline{\mathtt{v}} : \overline{\mathtt{D}}$, and $\overline{\mathtt{D}}\ \mathtt{<:}\ \overline{\mathtt{C}}$. Thus, we have $\mathtt{e} \mid \overline{\mathtt{L}} \longrightarrow \mathtt{v}_i \mid \overline{\mathtt{L}}$, finishing the case.

Case T-Invk:

$$\mathtt{e} = \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)@X} : \mathtt{C} \qquad \bullet; \Lambda; \bullet \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \Lambda = \mathit{requires}(\mathtt{X})$$
$$\bullet; \Lambda; \bullet \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} \qquad \overline{\mathtt{E}} \subset \overline{\mathtt{D}} \qquad \mathcal{L} \vdash \mathtt{E} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0, \Lambda) = \overline{\mathtt{D}} \to \mathtt{C}$$

By the induction hypothesis, there exist $i \geq 0$ and $\mathtt{e}_i'$ such that $\mathtt{e}_i \mid \overline{\mathtt{L}} \longrightarrow \mathtt{e}_i' \mid \overline{\mathtt{L}}$ or all $\mathtt{e}_i'$s are values $\mathtt{v}_0, \overline{\mathtt{v}}$. In the second case, RC-InvkRecv or RC-InvkArg finish the case. In the first case, by T-New, $\mathtt{e}_0 = \mathtt{new}\ \mathtt{C}_0\mathtt{(}\overline{\mathtt{w}}\mathtt{)}$. By Lemma 8, there exist $\overline{\mathtt{x}}, \mathtt{e}_0, \overline{\mathtt{L}}'', \overline{\mathtt{L}}'''$, and $\mathtt{C}'(\neq \mathtt{Object})$ such that $\mathit{mbody}(\mathtt{m}, \mathtt{C}_0, \overline{\mathtt{L}}', \mathit{filter}(\overline{\mathtt{L}}'''; \overline{\mathtt{L}}')) = \overline{\mathtt{x}}\mathtt{.e}_0\ \mathtt{in}\ \mathtt{C}', \overline{\mathtt{L}}''$ where $\{\overline{\mathtt{L}}'\} = \Lambda$ and the lengths of $\overline{\mathtt{x}}$ and $\overline{\mathtt{D}}$ are equal. As $\mathtt{C}' \neq \mathtt{Object}$, there exists $\mathtt{D}'$ such that $\mathtt{class}\ \mathtt{C}' \triangleleft \mathtt{D}'\ \{\cdots\}$. We have two subcases here depending on whether $\overline{\mathtt{L}}''$ is empty or not. If $\overline{\mathtt{L}}''$ is not empty, let $\overline{\mathtt{L}}'' = \overline{\mathtt{L}}''''; \mathtt{L}_0$ for some $\overline{\mathtt{L}}''''$ and $\mathtt{L}_0$. Then, the expression

$$\mathtt{e}' = \begin{bmatrix} \mathtt{new}\ \mathtt{C}_0\mathtt{(}\overline{\mathtt{w}}\mathtt{)} & /\mathtt{this} \\ \overline{\mathtt{v}} & /\overline{\mathtt{x}} \\ \mathtt{new}\ \mathtt{C}_0\mathtt{(}\overline{\mathtt{w}}\mathtt{)<C'},\overline{\mathtt{L}}'''',\mathit{filter}(\overline{\mathtt{L}}'''; \overline{\mathtt{L}}')\mathtt{>.m} & /\mathtt{proceed} \end{bmatrix} \mathtt{e}_0'$$

is well defined (note that the lengths of $\overline{\mathtt{x}}$ and $\overline{\mathtt{v}}$ are equal). Then, by R-InvkP and R-Invk, $\mathtt{e} \mid \overline{\mathtt{L}} \longrightarrow \mathtt{e}' \mid \overline{\mathtt{L}}$. The case where $\overline{\mathtt{L}}''$ is empty is similar.

Case T-New:

$$\mathtt{e} = \mathtt{new}\ \mathtt{C(}\overline{\mathtt{e}}\mathtt{)} \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}} \qquad \bullet; \Lambda; \bullet \vdash \overline{\mathtt{e}} : \overline{\mathtt{D}} \qquad \overline{\mathtt{D}}\ \mathtt{<:}\ \overline{\mathtt{C}}$$

By the induction hypothesis, either $\overline{\mathtt{e}}$ are all values, in which case $\mathtt{e}$ is also a value, or there exist i and $\mathtt{e}_i'$ such that $\mathtt{e}_i \mid \overline{\mathtt{L}} \longrightarrow \mathtt{e}_i' \mid \overline{\mathtt{L}}$, in which case RC-New finishes the case.

Case T-InvkA: Similar to the case for T-Invk.

The proof of Theorem 4 easily follows from the proofs above.