A Sound Type System for Layer Subtyping and Dynamically Activated First-Class Layers

Hiroaki Inoue and Atsushi Igarashi

Graduate School of Informatics, Kyoto University

Abstract. Key features of context-oriented programming (COP) are *layers*—modules to describe context-dependent behavioral variations of a software system—and their *dynamic activation*, which can modify the behavior of multiple objects that have already been instantiated. Type-checking programs written in a COP language is difficult because the activation of a layer can even change objects' interfaces. We formalize a small COP language called ContextFJ$_{<:}$ with its operational semantics and type system and show its soundness. The language features (1) dynamically activated *first-class* layers, (2) *inheritance* of layer definitions, and (3) layer *subtyping*.

1 Introduction

Software is much more interactive than it used to be: it interacts with not only users but also external resources such as network and sensors and changes its behavior according to inputs from these resources. For example, an e-mail reader may switch to a text-based mode when network throughput is low. Such external information that affects the behavior of software is often referred to as *contexts*. However, such context-dependent software is hard to develop and maintain, because the description of context-dependent behavior, which we desire to be modularized, often crosscuts with the dominating module structure. To address such a problem from a programming-language perspective, Context-Oriented Programming (COP) [9] has been proposed by Hirschfeld et al.

The main language constructs for COP are *layers*, which are modules to specify context-dependent behavior, and their *dynamic layer activation*. A layer is basically a collection of what are called *partial methods*, which add new behavior to existing objects or override existing methods. When a layer is activated at run time by a designated construct, the partial methods defined in it become effective, changing the behavior of objects until the activation ends. Roughly speaking, a layer abstracts a context and dynamic layer activation abstracts change of contexts.

JCop language [1] is an extension of Java with language constructs for COP. It not only supports basic COP constructs described above, but also introduces many advanced features such as inheritance of layer implementations and first-class layers. However, typechecking implemented in the JCop compiler does not take into account the fact that layer activation can change objects' interface by partial methods that add new methods and, as a result, not all "method not found" errors are prevented statically. In our previous work [14], we have studied this problem, proposed a type-safe version of JCop (we call Safe JCop in this paper) with informal discussions on how JCop can be made type-safe.

In this paper, we formalize most of the ideas proposed in the previous work and show they really make the language sound. More concretely, we develop a small COP language called ContextFJ$_{<:}$, which extends ContextFJ by Igarashi, Hirschfeld, and Masuhara [10, 11] to layer inheritance, subtyping of layer types, first-class layers and layer swapping; and we show type soundness of ContextFJ$_{<:}$. Main issues we have to deal with are (1) the semantics of layer inheritance, which adds another "dimension" to method lookup, (2) sound subtyping for first-class layers, which led us to two kinds of subtyping relation, and (3) (a limited form of) type-safe deactivation, which we realize by layer swapping, first proposed in the previous work [14]. We also have implemented a prototype of the proposed type system by extending the JCop compiler.

The rest of the paper is organized as follows. After informally reviewing features of Safe JCop in Section 2, we develop ContextFJ$_{<:}$ in Section 3 and state type soundness. In Section 4, we discuss related work and then conclude in Section 5. We omit some rules of ContextFJ$_{<:}$ and proofs for brevity; the full definitions and proofs can be found in the full version of the paper. The implementation of the type system and the full version are available at `http://www.fos.kuis.kyoto-u.ac.jp/~hinoue/`.

2 Language Constructs of Safe JCop

In this section, we review language constructs of Safe JCop, first described in [14], including first-class layers, layer inheritance/subtyping, and layer swapping along informal discussions about the type system.

As a running example, we consider programming a graphical computer game called *RetroAdventure* [2]. In this game, a player has a character "hero" that wanders around the game world. Here, we introduce class `Hero` that represents the hero, which has method `move` to walk around, and class `World` that represents the game world.

```
public class Hero {
  Position pos;
  public void move(Direction dir){
    pos = /* changes pos according to dir */;
  }
}
public class World { ... }
```

2.1 Layers and Partial Methods

As mentioned already, a first distinctive feature of COP is *layers*—collections of *partial methods* to modify the behavior of existing objects. A partial method is syntactically similar to an ordinary method declared in a class, except that the name is given in a qualified form Hero.move(); this means the partial method is going to override method move defined in Hero or (if it does not exist) add to Hero. A layer can contain partial methods for different classes, so, when it is activated, it can affect objects from various classes at once. Similarly to super calls in Java, the body of a partial method can contain proceed calls to invoke the original method overridden by this partial method.

Here, suppose that the hero's behavior is influenced by weather conditions in the game world. For example, in a rainy weather, the hero gets slow and, in a stormy weather, the hero cannot move as he likes. Here are layers that denote weathers of the game world.

```
public layer Rainy {
  /* partial method */
  public void Hero.move(Direction dir){
    pos = /* the distance of move is smaller */;
  }
}
public layer Stormy {
  /* partial method */
  public void Hero.move(Direction dir){
    proceed(randomDirection(dir));
  }
  /* baseless partial method */
  public Direction Hero.randomDirection(Direction dir){
    return /* add randomness to dir */;
  }
}
public layer Sunny { ... }
```

Rainy and Stormy have the definitions of Hero.move, which change the behavior of the original definition in different ways. In particular, Hero.move in Stormy uses proceed, replacing the arguments to calls to move. It also has Hero.randomDirection, used to determine a new randomized direction to which the hero is going to move.

Methods defined in classes are often referred to as *base methods* and partial methods without corresponding base methods as *baseless partial methods*. Notice that activating a layer with baseless partial methods extends object interfaces and proceed in a baseless method is unsafe unless another layer activation provides a baseless method of the same signature.

2.2 Layer Activation and First-Class Layers

In Safe JCop, a layer can be activated by using a layer instance (created by a new expression, just as an ordinary Java object, from a layer definition) in a with statement. The following code snippet shows how Rainy can be activated.

```
with(new Rainy()){
  hero.move(); /* The hero will get slow by Rainy weather. */
}
```

Inside the body of with, dynamic method dispatch is affected by the activated layers so that partial methods are looked up first. So, movement of the hero will be slow.

Layer activation has a dynamic extent in the sense that the behavior of objects changes even in methods called from inside with. If more than one layer is activated, a more recent activation has precedence and a proceed call in a more recently activated layer may call another partial method (of the same name) in another layer.

In Safe JCop, a layer instance is a first-class citizen and can be stored in a variable, passed to, or returned from a method. A layer name can be used as a type. Combining with layer subtyping discussed later, we can switch layers to activate by a run-time condition. For example, suppose that the game has *difficulty* levels, determined at run time according to some parameters, and each level is represented by an instance of a sublayer of Difficulty. Then, we can set the initial difficulty level by code like this:

```
Difficulty dif = /* an expression to compute difficulty */ ;
with(dif){...}
```

Moreover, a layer can declare fields (although we do not model fields in layers in this paper). So, first-class layers significantly enhances expressiveness of the language.

2.3 Dependencies between Layers

Baseless partial methods and layer activation that has dynamic extent pose a challenge on typechecking because activation of a layer including baseless partial methods can change object interfaces. So, a method invocation, including a proceed call, may or may not be safe depending on what layers are activated at the program point. Safe JCop adopts requires clauses [11] for layer definitions to express which layers should have been activated before activating each layer (instance). The type system checks whether each activation satisfies the requires clause associated to the activated layer and also uses requires clauses to estimate interfaces of objects at every program point.

For example, consider another layer ThunderStorm, which expresses an event in a game. It affects the way how the hero's direction is randomized during a storm and includes a baseless partial method with a proceed call. To prevent ThunderStorm from being activated in a weather other than a storm, the layer requires Stormy as follows:

```
public layer ThunderStorm requires Stormy {
  public Direction Hero.randomDirection(Direction dir){
    Direction tmpd = proceed(dir);
    ... /* change tmpd to speed up */
    return tmpd;
  }
}
```

An attempt at activating `ThunderStorm` without activating `Stormy` will be rejected by the type system (unless the activation appears in a layer requiring `Stormy`). Thanks to the `requires` clause, the type system knows the `proceed` call will not fail. (It will call the partial method in `Stormy` or some other depending on what layers are activated at run time.)

In general, a layer can `require` any number of layers.

2.4 Layer Inheritance

In Safe JCop, a layer can inherit definitions from another layer by using the keyword `extends` and the `extends` relation between layers yields subtyping, just like Java classes. If weather layers have many definitions in common, it is a good idea to define a superlayer `Weather` and concrete weather layers as its sublayers.

```
public abstract layer Weather {
  public String World.getWorldText(){
    ..... + this.getWeatherInfo() + ....;
  }
  public abstract String World.getWeatherInfo();
}
public layer Rainy extends Weather {
  public String World.getWeatherInfo(){
    return "rain";
  }
}
public layer Stormy extends Weather {
  public String World.getWeatherInfo(){
    return "storm";
  }
}
```

Here, `Weather` provides partial method `getWorldText` to retrieve the status of the world. Although the implementation of `World.getWeatherInfo` is not given here, concrete weather layers provide it by overriding.

Naturally, we expect an instance of a sublayer can be substituted for that of its super-layer. However, substitutability is more subtle that one might expect and we are led to distinguishing two kinds of substitutability and introducing two kinds of subtyping relation, called weak and normal subtyping.

Since a sublayer defines more partial methods than its superlayer, an instance of a sublayer can be used where a superlayer is `required`. For example, to activate the following layer called `Thunder`, which `requires Weather`, it suffices to activate `Rainy`, a sublayer of `Weather`, beforehand.

```
public layer Thunder requires Weather {
  public String World.getWeatherInfo(){
    return "thunder and " + proceed();
  }
}
```

```
...
with(new Rainy()){
  with(new Thunder()){...}
}
```

We will formalize substitutability about `requires` as *weak subtyping*, which is the reflexive transitive closure of the `extends` relation between layer types. For the weak subtyping to work, we require a sublayer declare what its superlayer `requires` because partial methods inherited from the superlayer may depend on them. We could relax this condition when a sublayer overrides all the partial methods but such a case is expected to be rare.[1] Therefore, we do not consider the case.

This notion of subtyping is called weak because it does *not* guarantee safe substitutability for *first-class* layers. Consider layer `Difficulty` again and assume that it requires no other layers and has sublayers `Easy` and `Hard`. In the following code snippet

```
Difficulty dif = someCondition() ? new Easy() : new Hard();
with(dif){...}
```

activation of `dif` appears safe because its static type `Difficulty` does not require any layer to have been activated. However, the case where `Easy` or `Hard` requires some layers breaks the expected invariant that `requires` is satisfied at run time. So, for assignments and parameter passing, we need one more condition for subtyping, namely, `requires` of a sublayer must be the same as that of its superlayer. We call this strong notion of subtyping *normal subtyping*.

Just like `Object` in Java, there is `Base`, which is a superlayer of all layers, in Safe JCop. If a layer omits the `extends` clause, it is implicitly assumed that the layer `extends Base`.

2.5 Layer Swapping and Deactivation

The original JCop provides constructs to *de*activate layers. However, only with `requires`, it is not easy to guarantee that layer deactivation does not lead to an error. For safe deactivation, it has to be checked that there is no layer that `requires` the deactivated layer, but the type system is not designed to keep track of *absence* of certain layers. Instead of general-purpose layer deactivation mechanisms, Safe JCop introduces a special construct to express one important idiom that uses deactivation, namely *layer swapping* to deactivate some layers and activate a layer at once.

In Safe JCop, we can define a layer as *swappable*, which means that all its sublayers can be swapped with each other, by adding the modifier `swappable`. The `swap` statement for layer swapping is of the following form:

`swap(`*activation_layer*, *deactivation_layer_type*`){ ... }`

[1] Re-typechecking inherited methods under the new `requires` clause would be another way to relax this condition but this is against modular checking.

The *activation_layer* is an expression whose static type must be a sublayer of *deactivation_layer_type*, which in turn has to be swappable. It deactivates *all* instances of *deactivation_layer_type* (and its sublayers), and activates the *activation_layer*.

Let's consider `Difficulty` once again. We could define `Difficulty` as a `swappable` layer and use `swap` to switch to another mode temporarily.

```
swappable layer Difficulty {...}
...
Difficulty dif = someCondition() ? new Easy() : new Hard();
with(dif){
  ...
  swap(new Hard(), Difficulty){
    // Enforce hard mode
  }
}
```

As discussed in the previous work [14], the layer swapping mechanism also requires no sublayers of a swappable layer to be `required` by other layers.

3 ContextFJ$_{<:}$

In this section, we formalize a core functional subset of Safe JCop as ContextFJ$_{<:}$, give its syntax, operational semantics and type system, and show type soundness. ContextFJ$_{<:}$, a descendant of Featherweight Java (FJ) [13], extends ContextFJ [10, 11] with layer inheritance, layer subtyping, first-class layers, and swappable layers. Note that we omit some rules (especially when they are similar to those in ContextFJ [11]). The whole calculus is in the full version of this paper. We also recommend that readers consult [11].

3.1 Syntax

Let metavariables `C`, `D` and `E` range over class names; `L` over layer names; `f` and `g` over field names; `m` over method names; `x` and `y` over variables, which contains special variable `this`. The abstract syntax of ContextFJ$_{<:}$ is given in Fig. 1.

Following FJ, we use overlines to denote sequences: so, $\overline{\texttt{f}}$ stands for a possibly empty sequence $\texttt{f}_1, \cdots, \texttt{f}_n$ and similarly for $\overline{\texttt{T}}$, $\overline{\texttt{x}}$, $\overline{\texttt{e}}$, and so on. The empty sequence is denoted by $\bullet$. Concatenation of sequences is often denoted by a comma except for layer names, for which we use a semicolon. We also abbreviate pairs of sequences, writing "$\overline{\texttt{T}}\ \overline{\texttt{f}}$" for "$\texttt{T}_1\ \texttt{f}_1, \cdots, \texttt{T}_n\ \texttt{f}_n$", where n is the length of $\overline{\texttt{T}}$ and $\overline{\texttt{f}}$, and similarly "$\overline{\texttt{T}\ \texttt{f};}$" as shorthand for the sequence of declarations "$\texttt{T}_1\ \texttt{f}_1;\ldots\texttt{T}_n\ \texttt{f}_n;$" and "$\texttt{this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;}$" for "$\texttt{this.f}_1\texttt{=f}_1\texttt{;}\ldots\texttt{;this.f}_n\texttt{=f}_n\texttt{;}$". Sequences of field declarations, parameter names, layer names, and method declarations are assumed to contain no duplicate names.

We briefly explain the syntax, focusing on COP-related constructs. A layer definition `LA` consists of optional modifier `swappable`, its name, its superlayer name, layers that it `requires`, and partial methods. A partial method is similar

$$
\begin{array}{lll}
\texttt{T},\texttt{S} & ::= \texttt{C} \mid \texttt{L} & (\textit{types}) \\
\texttt{CL} & ::= \texttt{class C} \triangleleft \texttt{C \{ } \overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}} \texttt{ \}} & (\textit{classes}) \\
\texttt{LA} & ::= [\texttt{swappable}]\ \texttt{layer L} \triangleleft \texttt{L req } \overline{\texttt{L}} \texttt{ \{ } \overline{\texttt{PM}} \texttt{ \}} & (\textit{layers}) \\
\texttt{K} & ::= \texttt{C(}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{)\{ super(}\overline{\texttt{f}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{; \}} & (\textit{constructors}) \\
\texttt{M} & ::= \texttt{T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}} & (\textit{methods}) \\
\texttt{PM} & ::= \texttt{T C.m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}} & (\textit{partial methods}) \\
\texttt{e},\texttt{d} & ::= \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m(}\overline{\texttt{e}}\texttt{)} \mid \texttt{new T(}\overline{\texttt{e}}\texttt{)} \mid \texttt{with e e} \mid \texttt{swap (e,L) e} & (\textit{expressions}) \\
 & \quad \mid \texttt{proceed(}\overline{\texttt{e}}\texttt{)} \mid \texttt{super.m(}\overline{\texttt{e}}\texttt{)} \mid \texttt{new C(}\overline{\texttt{v}}\texttt{)<C,}\overline{\texttt{L}}\texttt{,}\overline{\texttt{L}}\texttt{>.m(}\overline{\texttt{e}}\texttt{)} & \\
\texttt{v},\texttt{w} & ::= \texttt{new C(}\overline{\texttt{v}}\texttt{)} \mid \texttt{new L()} & (\textit{values})
\end{array}
$$

Fig. 1. Syntax of ContextFJ$_{<:}$. A phrase enclosed by [] is optional.

to a method but the former specifies which `m` to modify by qualifying the simple method name with its class `C`. Instantiation can be a layer instance `new L()`. Note that arguments to the constructor are always empty because a layer has no fields. In the expression `with` $\texttt{e}_1$ $\texttt{e}_2$, $\texttt{e}_1$ stands for the layer to be activated and $\texttt{e}_2$ the body of `with`. In the expression `swap` $(\texttt{e}_1,\ \texttt{L})\ \texttt{e}_2$, $\texttt{e}_1$ means the layer to be activated, `L` layers to deactivate, $\texttt{e}_2$ the body. All instances of `L` and its subclasses are deactivated. The expression $\texttt{new C(}\overline{\texttt{v}}\texttt{)<D,}\overline{\texttt{L}}'\texttt{,}\overline{\texttt{L}}\texttt{>.m(}\overline{\texttt{e}}\texttt{)}$ is a special run-time expression that is related to method invocation mechanism of COP, and not supposed to appear in classes and layers. It basically means that `m` is going to be invoked on $\texttt{new C(}\overline{\texttt{v}}\texttt{)}$. The annotation $\texttt{<D,}\overline{\texttt{L}}'\texttt{,}\overline{\texttt{L}}\texttt{>}$ is used to model `super` and `proceed`. $\overline{\texttt{L}}$ means activated layers in the method lookup and `D` and $\overline{\texttt{L}}'$ (which is assumed to be a prefix of $\overline{\texttt{L}}$) stand for the location of a "cursor" where the method lookup starts from.

Program. A ContextFJ$_{<:}$ program $(CT, LT, \texttt{e})$ consists of a class table CT, a layer table LT and an expression `e`, which stands for the body of the `main` function. CT maps a class name to a class definition and LT a layer name to a layer definition. A layer definition can be regarded as a function that maps a partial method name `C.m` to a partial method definition. So, we can view LT as a Curried function, so we often write $LT(\texttt{L})(\texttt{C.m})$ for the partial method `C.m` in `L` in a program. We assume that the domains of CT and LT are finite. Precisely speaking, the semantics and type system are parameterized over CT and LT but, to lighten the notation, we assume them to be fixed and omit from judgments.

Given CT and LT, `extends` and `requires` clauses are considered relations, written $\triangleleft$ and `req`, respectively, over class/layer names. As usual, we write $\mathcal{R}^+$ for the transitive closure of relation $\mathcal{R}$; similarly for $\mathcal{R}^*$ for the reflexive transitive closure of $\mathcal{R}$. We write `L swappable` if $LT(\texttt{L})$ is defined with the `swappable` modifier.

We assume the following sanity conditions are satisfied by a given program:

1. $CT(\texttt{C}) = \texttt{class C ...}$ for any $\texttt{C} \in dom(CT)$.
2. $\texttt{Object} \notin dom(CT)$.
3. For every class name `C` (except `Object`) appearing anywhere in CT, $\texttt{C} \in dom(CT)$.

4. $LT(\mathtt{L}) = \mathtt{...\ layer\ L\ ...}$ for any $\mathtt{L} \in dom(LT)$.
5. $\mathtt{Base} \notin dom(LT)$.
6. For every layer name `L` (except `Base`) appearing anywhere in LT, $\mathtt{L} \in dom(LT)$.
7. Both for classes and layers, there are no cycles in the transitive closure of the `extends` clauses.
8. $LT(\mathtt{L})(\mathtt{C.m}) = \mathtt{...\ C.m(...)\{...\}}$ for any $\mathtt{L} \in dom(LT)$ and $(\mathtt{C.m}) \in dom(LT(\mathtt{L}))$.
9. Relation $\mathcal{R}$ is defined by: $\mathtt{L}_1\mathcal{R}\mathtt{L}_2$ iff $\mathtt{L}_1\ \mathtt{req\hat{}+}\ \mathtt{L}_2$. $\mathcal{R}$ has no cycles.
10. A layer cannot `require` any superlayer of it, that is, $\mathtt{L}_1\ \triangleleft\mathtt{\hat{}+}\ \mathtt{L}_2 \to \neg(\mathtt{L}_1\mathcal{R}\mathtt{L}_2)$.
11. $\mathtt{L}_1\ \triangleleft^{+}\ \mathtt{L}_2 \wedge \mathtt{L}_2\ \mathtt{swappable} \to \neg(\exists \mathtt{L}_3.\mathtt{L}_3\mathcal{R}\mathtt{L}_1)$

In the condition 6, like `Object` of classes, `Base` layer is defined as the root of layer sub-typing tree. Conditions 7, 9, 10 and are very important for our formal system, because they are used to ensure that `proceed` and `super` calls will not fail. The final condition means that no sublayers of a swappable layer can be `require`d by other layers, as we mentioned earlier.

3.2 Operational Semantics

Lookup Functions. We need a few auxiliary lookup functions to define operational semantics. The function $\mathit{fields}(\mathtt{C})$ (whose definition is omitted) returns a sequence $\overline{\mathtt{T}}\ \overline{\mathtt{f}}$ of pairs of a field name and its type by collecting all field declarations from `C` and its superclasses. Other lookup functions are defined in Fig. 2. The function $\mathit{pmbody}(\mathtt{m},\mathtt{C},\mathtt{L})$ returns the parameters and body $\overline{\mathtt{x}}.\mathtt{e}$ of the partial method `C.m` defined in layer `L`. If `C.m` is not found in `L`, the superlayer of `L` is searched and so on. The function $\mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}_1,\overline{\mathtt{L}}_2)$ returns the parameters and body $\overline{\mathtt{x}}.\mathtt{e}$ of method `m` in class `C` when the search starts from $\overline{\mathtt{L}}_1$; the other sequence $\overline{\mathtt{L}}_2$ keeps track of the layers that are activated when the search initially started. It also returns `D` and $\overline{\mathtt{L}}''$ (which will be a prefix of $\overline{\mathtt{L}}_2$), information on where the method has been found. For example, since the rule MB-Layer means that the method is found in class `C` and layer $\mathtt{L}_0$ (or its superlayers), which is the rightmost layer of $\overline{\mathtt{L}}_1 = (\overline{\mathtt{L}}';\mathtt{L}_0)$, *mbody* returns `C` and $(\overline{\mathtt{L}}';\mathtt{L}_0)$. Such information will be used in reduction rules to deal with `proceed` and `super`. Readers familiar with ContextFJ will notice that the rules for *mbody* are mostly the same as those in ContextFJ, except that $\mathit{pmbody}(\mathtt{m},\mathtt{C},\mathtt{L})$ is substituted for $PT(\mathtt{m},\mathtt{C},\mathtt{L})$ to take layer inheritance into account.

Operational Semantics. The operational semantics of ContextFJ$_{<:}$ is given by a reduction relation of the form $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$, read "expression `e` reduces to $\mathtt{e}'$ under the activated layers $\overline{\mathtt{L}}$". The sequence $\overline{\mathtt{L}}$ of layer names stands for nesting of `with` and the rightmost name stands for the most recently activated layer. $\overline{\mathtt{L}}$ do not contain duplicate names. Note that we put a sequence of layer names $\overline{\mathtt{L}}$ rather than layer instances because layer instances have no fields and `new L()` and `L` can be identified. If we modelled fields in layer instances, we would have to put instances for layer names.

$\boxed{\mathit{pmbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e}}$

$$\frac{\mathit{LT}(\mathtt{L})(\mathtt{C.m}) = \mathtt{T}_0\ \mathtt{C.m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}}}{\mathit{pmbody}(\mathtt{m},\mathtt{C},\mathtt{L}) = \overline{\mathtt{x}}\mathtt{.e}} \quad \text{(PMB-Layer)}$$

$$\frac{\mathit{LT}(\mathtt{L})(\mathtt{C.m})\ \text{undefined} \qquad \mathtt{L} \lhd \mathtt{L}_S \qquad \mathit{pmbody}(\mathtt{m},\mathtt{C},\mathtt{L}_s) = \overline{\mathtt{x}}\mathtt{.e}}{\mathit{pmbody}(\mathtt{m},\mathtt{C},\mathtt{L}) = \overline{\mathtt{x}}\mathtt{.e}} \quad \text{(PMB-Super)}$$

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

$$\frac{\mathtt{class\ C} \lhd \mathtt{D\ \{\ ...\ T_0\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}\ ...\}}}{\mathit{mbody}(\mathtt{m},\mathtt{C},\bullet,\overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e}\ \mathtt{in}\ \mathtt{C},\bullet} \quad \text{(MB-Class)}$$

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

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

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

Fig. 2. ContextFJ$_{<:}$: Lookup functions.

Before giving reduction rules, we have to define two auxiliary functions, $\mathit{with}(\mathtt{L},\overline{\mathtt{L}})$ and $\mathit{swap}(\mathtt{L},\mathtt{L}_{sw},\overline{\mathtt{L}})$ to manipulate activated layers.

$$\mathit{with}(\mathtt{L},\overline{\mathtt{L}}) = (\overline{\mathtt{L}} \setminus \{\mathtt{L}\});\mathtt{L} \qquad \mathit{swap}(\mathtt{L},\mathtt{L}_{sw},\overline{\mathtt{L}}) = (\overline{\mathtt{L}} \setminus \{\mathtt{L}' \mid \mathtt{L}' \lhd^* \mathtt{L}_{sw}\});\mathtt{L}$$

The function *with* removes `L` (if exists) from layer sequence $\overline{\mathtt{L}}$ and adds `L` to the end of $\overline{\mathtt{L}}$ and *swap* removes $\mathtt{L}_{sw}$ and all sublayers of $\mathtt{L}_{sw}$ from $\overline{\mathtt{L}}$, and adds `L` to the end of $\overline{\mathtt{L}}$.

Main reduction rules are found in Fig. 3. The rules R-Invk and R-InvkP for method invocation are essentially the same as ones in ContextFJ. R-Invk initializes the cursor according to the currently activated layers $\overline{\mathtt{L}}$ and R-InvkP represents invocation of a partial method (the rule for base method invocation is omitted). Note how `this`, `proceed` and `super` are replaced with the receiver with different cursor locations. For `proceed`, the cursor moves one layer to the left and, for `super`, the cursor moves one level up. The rules RC-With and RC-Swap are related to layer activation and swapping, respectively. The rule RC-With means that `with (new L()) e` executes `e` with `L` activated (as the first layer). The rule RC-Swap is similar; it means that `swap (new L(),` $\mathtt{L}_{sw}$`) e` executes by deactivating all sublayers of $\mathtt{L}_{sw}$ and activating a layer `L`.

$\boxed{\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'}$

$$\frac{\overline{\mathtt{L}} \vdash \mathtt{new\ C(\overline{v})<C,\overline{L},\overline{L}>.m(\overline{w})} \longrightarrow \mathtt{e}'}{\overline{\mathtt{L}} \vdash \mathtt{new\ C(\overline{v}).m(\overline{w})} \longrightarrow \mathtt{e}'} \qquad \text{(R-INVK)}$$

$$\frac{\begin{array}{c} \mathit{mbody}(\mathtt{m}, \mathtt{C}', \overline{\mathtt{L}}'', \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0 \ \mathtt{in}\ \mathtt{C}'', (\overline{\mathtt{L}}'''; \mathtt{L}_0) \\ \mathtt{class\ C''} \triangleleft \mathtt{\ D\{...\}} \end{array}}{\begin{array}{l} \overline{\mathtt{L}} \vdash \mathtt{new\ C(\overline{v})<C',\overline{L}'',\overline{L}'>.m(\overline{w})} \longrightarrow \\ \qquad \begin{bmatrix} \mathtt{new\ C(\overline{v})} & \mathtt{/this}, \\ \overline{\mathtt{w}} & /\overline{\mathtt{x}}, \\ \mathtt{new\ C(\overline{v})<C'',\overline{L}''',\overline{L}'>.m/proceed}, \\ \mathtt{new\ C(\overline{v})<D,\overline{L}',\overline{L}'>} & \mathtt{/super} \end{bmatrix} \mathtt{e}_0 \end{array}} \qquad \text{(R-INVKP)}$$

$$\frac{\mathit{with}(\mathtt{L}, \overline{\mathtt{L}}) = \overline{\mathtt{L}}' \qquad \overline{\mathtt{L}}' \vdash \mathtt{e} \longrightarrow \mathtt{e}'}{\overline{\mathtt{L}} \vdash \mathtt{with\ new\ L()\ e} \longrightarrow \mathtt{with\ new\ L()\ e}'} \qquad \text{(RC-WITH)}$$

$$\frac{\mathit{swap}(\mathtt{L}, \mathtt{L}_{sw}, \overline{\mathtt{L}}) = \overline{\mathtt{L}}' \qquad \overline{\mathtt{L}}' \vdash \mathtt{e} \longrightarrow \mathtt{e}'}{\overline{\mathtt{L}} \vdash \mathtt{swap\ (new\ L(),L}_{sw}\mathtt{)\ e} \longrightarrow \mathtt{swap\ (new\ L(),L}_{sw}\mathtt{)\ e}'} \qquad \text{(RC-SWAP)}$$

Fig. 3. ContextFJ$_{<:}$: Reduction Rules.

3.3 Type System

As usual, the role of a type system is to ensure the absence of a certain class of run-time errors. Here, they are "field-not-found" and "method-not-found" errors, including the failure of `proceed` or `super` calls.

As discussed in the last section, the type system takes information on activated layers at every program point into account. We approximate such information by a set Λ of layer names, which mean that, for any element in Λ, an instance of one of its sublayers has to be activated at run time. This set gives underapproximation in the sense that other layers might be activated. Activated layers are approximated by sets rather than sequences because the type system is mainly concerned about access to fields and methods and the order of activated layers does not influence which fields and methods are accessible.

In our type system, a type judgment for an expression is of the form $\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e} : \mathtt{T}$, where Γ is a type environment, which records types of variables, and $\mathcal{L}$ stands for where `e` appears, namely, a method in a class or a partial method in a layer. For example, the body of the partial method `World.getWeatherInfo()` in layer `Thunder` is typed as follows:

$$\begin{array}{l} \mathtt{Thunder.World.getWeatherInfo}; \{\mathtt{Weather}, \mathtt{Thunder}\}; \bullet \\ \qquad\qquad \vdash \mathtt{"thunder\ and\ "\ +\ proceed()} : \mathtt{String} \end{array}$$

where $\bullet$ stands for the empty type environment. The layer name set $\{\mathtt{Weather}, \mathtt{Thunder}\}$ comes from the fact that `Thunder requires Weather`. `Thunder` is also included because `Thunder` is obviously activated when a partial method defined in this very layer is executed.

We start with the definitions of two kinds of layer subtyping discussed in the last section and proceed to functions to look up method types and typing rules.

Subtyping. We define subtyping $\mathtt{C} \mathrel{\mathtt{<:}} \mathtt{D}$ for class types, weak subtyping $\mathtt{L}_1 \mathrel{\mathtt{<:}_w} \mathtt{L}_2$ and normal subtyping $\mathtt{L}_1 \mathrel{\mathtt{<:}} \mathtt{L}_2$ for layer types by the rules in Fig. 4. Class subtyping $\mathtt{C} \mathrel{\mathtt{<:}} \mathtt{D}$ (whose rules are omitted) is defined as the reflexive and transitive closure of $\lhd$, just as FJ. Weak layer subtyping is also the reflexive and transitive closure of $\lhd$. We extend it to the relation $\Lambda_1 \mathrel{\mathtt{<:}_w} \Lambda_2$ between layer name sets by LSS-Intro. It is used to check activated layers Λ_1 satisfy requirement Λ_2 given by a `requires` clause in typechecking a layer activation. So, for every element in Λ_2, there must exist a sublayer of it in Λ_1. Normal subtyping is almost the reflexive and transitive closure of $\lhd$ but there is one additional condition: for $\mathtt{L}_1$ to be a normal subtype of $\mathtt{L}_2$, the layers they `require` must be the same (LS-Extends). The notation $\mathtt{L}\ \mathtt{req}\ \Lambda$ means that $\mathtt{L}\ \mathtt{req}\ \mathtt{L}'$ for any $\mathtt{L}' \in \Lambda$.

$\boxed{\text{layer subtyping } \mathtt{<:}}$

$$\frac{}{\mathtt{L} \mathrel{\mathtt{<:}} \mathtt{L}} \quad \text{(LS-Refl)}$$

$$\frac{\mathtt{L}_1 \mathrel{\mathtt{<:}} \mathtt{L}_2 \qquad \mathtt{L}_2 \mathrel{\mathtt{<:}} \mathtt{L}_3}{\mathtt{L}_1 \mathrel{\mathtt{<:}} \mathtt{L}_3} \quad \text{(LS-Trans)}$$

$$\frac{\mathtt{L} \lhd \mathtt{Base} \qquad \mathtt{L}\ \mathtt{req}\ \emptyset}{\mathtt{L} \mathrel{\mathtt{<:}} \mathtt{Base}} \quad \text{(LS-Base)}$$

$$\frac{\mathtt{L}_1 \lhd \mathtt{L}_2 \qquad \mathtt{L}_1\ \mathtt{req}\ \Lambda \qquad \mathtt{L}_2\ \mathtt{req}\ \Lambda}{\mathtt{L}_1 \mathrel{\mathtt{<:}} \mathtt{L}_2} \quad \text{(LS-Extends)}$$

$\boxed{\text{weak layer subtyping } \mathtt{<:}_w}$

$$\frac{}{\mathtt{L} \mathrel{\mathtt{<:}_w} \mathtt{L}} \quad \text{(LSw-Refl)}$$

$$\frac{\mathtt{L}_1 \mathrel{\mathtt{<:}_w} \mathtt{L}_2 \qquad \mathtt{L}_2 \mathrel{\mathtt{<:}_w} \mathtt{L}_3}{\mathtt{L}_1 \mathrel{\mathtt{<:}_w} \mathtt{L}_3} \quad \text{(LSw-Trans)}$$

$$\frac{\mathtt{L}_1 \lhd \mathtt{L}_2}{\mathtt{L}_1 \mathrel{\mathtt{<:}_w} \mathtt{L}_2} \quad \text{(LSw-Extends)}$$

$\boxed{\text{layer set subtyping}}$

$$\frac{\forall \mathtt{L}_0 \in \Lambda_0 . \exists \mathtt{L}_1 \in \Lambda_1 \text{ s.t. } \mathtt{L}_1 \mathrel{\mathtt{<:}_w} \mathtt{L}_0}{\Lambda_1 \mathrel{\mathtt{<:}_w} \Lambda_0} \quad \text{(LSS-Intro)}$$

Fig. 4. ContextFJ$_{<:}$: Subtyping Relations.

Method type lookup. Similarly to *pmbody* and *mbody*, we define two auxiliary functions *pmtype* and *mtype* to look up the signature $\overline{\mathtt{T}} \rightarrow \mathtt{T}_0$ (consisting of argument type $\overline{\mathtt{T}}$ and a return type $\mathtt{T}_0$) of a (partial) method. $\mathit{pmtype}(\mathtt{m}, \mathtt{C}, \mathtt{L})$ returns the signature of `C.m` in `L` (or one of its superlayers); we omit its definition, which is similar to *pmbody*. $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_1, \Lambda_2)$, whose definition is essentially the same (save layer inheritance) as that in ContextFJ but shown in Fig. 5, returns the type of `m` in `C` under the assumption that Λ_1 is activated. The other layer set Λ_2 ($\supseteq \Lambda_1$) is used when the lookup goes to a superclass. If Λ_1 and Λ_2 are the same, which is mostly the case, we write $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_1)$.

These rules by themselves do not define *mtype* as a function, because different layers may contain partial methods of the same name with different signatures.

$$\boxed{\mathit{mtype}(\mathtt{m},\mathtt{C},\Lambda_1,\Lambda_2)=\overline{\mathtt{T}}\rightarrow\mathtt{T}_0}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D\ \{...\ T_0\ m(\overline{T}\ \overline{x})\{\ return\ e;\ \}\ ...\}}}{\mathit{mtype}(\mathtt{m},\mathtt{C},\Lambda_1,\Lambda_2)=\overline{\mathtt{T}}\rightarrow\mathtt{T}_0}\quad\text{(MT-Class)}$$

$$\frac{\mathtt{L}\in\Lambda_1 \qquad \mathit{pmtype}(\mathtt{m},\mathtt{C},\mathtt{L})=\overline{\mathtt{T}}\rightarrow\mathtt{T}_0}{\mathit{mtype}(\mathtt{m},\mathtt{C},\Lambda_1,\Lambda_2)=\overline{\mathtt{T}}\rightarrow\mathtt{T}_0}\quad\text{(MT-PMethod)}$$

$$\frac{\begin{array}{c}\mathtt{class\ C} \lhd \mathtt{D\ \{...\ \overline{M}\ \}} \qquad \mathtt{m}\notin\overline{\mathtt{M}}\\ \forall\mathtt{L}\in\Lambda_1.\mathit{pmtype}(\mathtt{m},\mathtt{C},\mathtt{L})\text{ undefined} \qquad \mathit{mtype}(\mathtt{m},\mathtt{D},\Lambda_2,\Lambda_2)=\overline{\mathtt{T}}\rightarrow\mathtt{T}_0\end{array}}{\mathit{mtype}(\mathtt{m},\mathtt{C},\Lambda_1,\Lambda_2)=\overline{\mathtt{T}}\rightarrow\mathtt{T}_0}\quad\text{(MT-Super)}$$

Fig. 5. ContextFJ$_{<:}$: Method Type Lookup functions.

$$\boxed{\mathcal{L};\Lambda;\Gamma\vdash\mathtt{e}:\mathtt{T}}$$

$$\frac{\mathcal{L};\Lambda;\Gamma\vdash\mathtt{e}_0:\mathtt{C}_0 \qquad \mathit{mtype}(\mathtt{m},\mathtt{C}_0,\Lambda)=\overline{\mathtt{T}}\rightarrow\mathtt{T}_0 \qquad \mathcal{L};\Lambda;\Gamma\vdash\overline{\mathtt{e}}:\overline{\mathtt{S}} \qquad \overline{\mathtt{S}}\mathrel{<:}\overline{\mathtt{T}}}{\mathcal{L};\Lambda;\Gamma\vdash\mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)}:\mathtt{T}_0}\quad\text{(T-Invk)}$$

$$\frac{\mathcal{L};\Lambda;\Gamma\vdash\mathtt{e}_l:\mathtt{L} \qquad \mathtt{L\ req}\ \Lambda' \qquad \Lambda\mathrel{<:_w}\Lambda' \qquad \mathcal{L};\Lambda\cup\{\mathtt{L}\};\Gamma\vdash\mathtt{e}_0:\mathtt{T}_0}{\mathcal{L};\Lambda;\Gamma\vdash\mathtt{with}\ \mathtt{e}_l\ \mathtt{e}_0:\mathtt{T}_0}\quad\text{(T-With)}$$

$$\frac{\begin{array}{c}\mathcal{L};\Lambda;\Gamma\vdash\mathtt{e}_l:\mathtt{L} \qquad \mathtt{L}_{sw}\ \mathtt{swappable} \qquad \mathtt{L}\mathrel{<:_w}\mathtt{L}_{sw} \qquad \mathtt{L\ req}\ \Lambda'\\ \Lambda_{rm}=\Lambda\setminus\{\mathtt{L}'\mid\mathtt{L}'\mathrel{<:_w}\mathtt{L}_{sw}\} \qquad \Lambda_{rm}\mathrel{<:_w}\Lambda' \qquad \mathcal{L};\Lambda_{rm}\cup\{\mathtt{L}\};\Gamma\vdash\mathtt{e}_0:\mathtt{T}_0\end{array}}{\mathcal{L};\Lambda;\Gamma\vdash\mathtt{swap}\ (\mathtt{e}_l,\mathtt{L}_{sw})\mathtt{e}_0:\mathtt{T}_0}\quad\text{(T-Swap)}$$

$$\frac{\mathtt{L\ req}\ \Lambda' \qquad \mathit{mtype}(\mathtt{m},\mathtt{C},\Lambda',\Lambda'\cup\{\mathtt{L}\})=\overline{\mathtt{T}}\rightarrow\mathtt{T}_0 \qquad \mathtt{L.C.m};\Lambda;\Gamma\vdash\overline{\mathtt{e}}:\overline{\mathtt{S}} \qquad \overline{\mathtt{S}}\mathrel{<:}\overline{\mathtt{T}}}{\mathtt{L.C.m};\Lambda;\Gamma\vdash\mathtt{proceed(}\overline{\mathtt{e}}\mathtt{)}:\mathtt{T}_0}$$
(T-Proceed)

Fig. 6. ContextFJ$_{<:}$: Expression typing.

So, precisely speaking, it should rather be understood as a relation; in a well-typed program, it will behave as a function, though.

Expression Typing. As mentioned already, the type judgment for expressions is of the form $\mathcal{L};\Lambda;\Gamma\vdash\mathtt{e}:\mathtt{T}$, read "`e` is given type `T` under context Γ, location $\mathcal{L}$ and layer set Λ". In addition to `C.m` and `L.C.m`, $\mathcal{L}$ can be $\bullet$, which means the top-level (i.e., under execution). Main typing rules are given in Fig. 6.

The rule T-Invk is straightforward: for the method signature $\overline{\mathtt{T}}\rightarrow\mathtt{T}_0$, retrieved from the receiver type $\mathtt{C}_0$, the types of the actual arguments must be their subtypes. The whole expression is given the method return type. The rule T-Proceed is similar, but the activated layer set Λ' is taken from the `requires` clause of the layer `L` in which this expression appears. The last argument to *mtype* is $\Lambda\cup\{\mathtt{L}\}$ because a `proceed` call can proceed to a partial method `D.m` (where `D` is a superclass of `C`) defined in the same layer `L`. The rule T-With checks, by

Λ <:$_w$ Λ', that the layers `required` by `L`—the layer type to be activated—are already activated and that the body $\mathtt{e}_0$ is well typed under the assumption that `L` is additionally activated. T-Swap is similar; the set Λ_{rm} stands for the set of layers after deactivation and must be a weak subtype of the required set Λ'.

Other Typing Rules. For typing other entities, such as (partial) methods and layers, we use the following judgments:

`PM ok in L`	partial method `PM` is well formed in layer `L`
`M ok in C`	base method `M` is well formed in class `C`
`LA ok`	layer definition `LA` is well formed
`CL ok`	class definition `CL` is well formed
$\mathit{override}(CT, LT)$	method override is valid in CT and LT
$\vdash (CT, LT, \mathtt{e}) : \mathtt{T}$	program $(CT, LT, \mathtt{e})$ is given type `T`

Representative typing rules are given in Fig. 7. The rule T-PMethod for a partial method means that the method body `e` is typed under the layer set `required` by this layer. The rule T-Layer demands that the requires clause of the layer be *covariant* and all partial methods are well formed. A program is typed if all classes and layers in CT and LT are well formed, the main expression `e` is typed (at the top-level $\bullet$), and $\mathit{override}(CT, LT)$ holds.

The most involved is the rule to check valid method overriding. Note that, unlike Java, checking valid method overriding requires a whole program (except for the main expression) because a layer may add a new method to a base class, one of whose subclass may accidentally define a method of the same name without knowing of that layer. The first premise means that for two partial methods of the same (qualified) name must have the same signature. The second premise means that, for any partial method, the overridden method (base method in `C` or partial methods for `C`'s superclass) must have the same signature. Finally, the third premise means that a base method can override a (partial) method in its superclass (or layers modifying it) with a covariant return type.

3.4 Type Soundness

We prove type soundness of ContextFJ$_{<:}$ via subject reduction and progress [17]. To prove subject reduction, we have to give a typing rule for run-time expressions of the form `new C(`$\overline{\mathtt{v}}$`)<D,`$\overline{\mathtt{L}}'$`,`$\overline{\mathtt{L}}$`>.m(`$\overline{\mathtt{e}}$`)`, which are not supposed to appear in a class/layer table. The typing rule is given as follows:

$$\frac{\begin{array}{c}\forall \mathtt{L}_0 \in \Lambda_0 . \exists \mathtt{L}_1 \in \Lambda_1 . \mathtt{L}_1 \mathrel{<:_w} \mathtt{L}_0 \\ \text{or } \exists \mathtt{L}_2 \in \mathit{dom}(LT) . \mathtt{L}_2 \ \mathtt{swappable} \text{ and } \mathtt{L}_0 \mathrel{<:_w} \mathtt{L}_2 \text{ and } \mathtt{L}_1 \mathrel{<:_w} \mathtt{L}_2\end{array}}{\Lambda_1 \mathrel{<:_{sw}} \Lambda_0} \qquad \text{(LSSW-Intro)}$$

$$\frac{\begin{array}{c}\mathit{fields}(\mathtt{C_0}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \mathcal{L};\Lambda;\Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{S}} \qquad \overline{\mathtt{S}} \mathrel{<:} \overline{\mathtt{T}} \\ \mathtt{C_0} \mathrel{<:} \mathtt{D_0} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{D_0}, \{\overline{\mathtt{L}}'\}, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{T}}' \rightarrow \mathtt{T_0} \\ \mathcal{L};\Lambda;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}}' \qquad \overline{\mathtt{S}}' \mathrel{<:} \overline{\mathtt{T}}' \qquad \overline{\mathtt{L}}' \text{ is a prefix of } \overline{\mathtt{L}} \\ \{\overline{\mathtt{L}}\}\ \mathit{wf} \qquad \Lambda \mathrel{<:_{sw}} \{\overline{\mathtt{L}}\} \qquad \mathit{WP}(\mathtt{m}, \mathtt{D_0}, \overline{\mathtt{L}}', \overline{\mathtt{L}})\end{array}}{\mathcal{L};\Lambda;\Gamma \vdash \mathtt{new\ C_0(\overline{v})<D_0,\overline{L}',\overline{L}>.m(\overline{e})} : \mathtt{T_0}} \qquad \text{(T-InvkA)}$$

Basically, it combines the typing rules for `new` and method invocation with a few additional complications. The first three premises mean that the types of the values $\overline{\mathtt{v}}$ for fields $\overline{\mathtt{f}}$ are subtypes of the declared. The method signature is taken from the current location $\mathtt{<\overline{D}_0,\overline{L}',\overline{L}>}$ of the cursor and the types of the actual arguments $\overline{\mathtt{e}}$ have to be subtypes of the formal argument types. We detail the last two conditions below.

The following two predicates $\Lambda\ \mathit{wf}$ and $\mathit{WP}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}', \overline{\mathtt{L}})$ are crucial to ensure successful `proceed` and `super` calls in the presence of `with`.[2] The condition $\{\overline{\mathtt{L}}\}\ \mathit{wf}$, read "layer set $\{\overline{\mathtt{L}}\}$ is well formed," means that for every layer in the set, there are layers that it `requires` in the same set. Formally, it is defined by the following rule:

$$\frac{\begin{array}{c}\forall \mathtt{L} \in \Lambda, \forall \mathtt{L}' \text{ s.t, } \mathtt{L}\mathcal{R}\mathtt{L}', (\exists \mathtt{L}'' \in \Lambda \text{ s.t, } \mathtt{L}'' \mathrel{<:_w} \mathtt{L}') \\ \text{and } (\forall \mathtt{L}'' \mathrel{\lhd^*} \mathtt{L}', \forall \mathtt{L}''' \text{ s.t, } \mathtt{L}''\mathcal{R}\mathtt{L}''', \neg(\mathtt{L} \mathrel{\lhd^*} \mathtt{L}''')).\end{array}}{\Lambda\ \mathit{wf}} \qquad \text{(LS-wf)}$$

The last premise $\mathit{WP}(\mathtt{m}, \mathtt{D_0}, \overline{\mathtt{L}}', \overline{\mathtt{L}})$ intuitively means "a chain of `proceed` calls from the given cursor location eventually reaches a (partial) method that does *not* call `proceed`" and is defined by the following rules:

$$\frac{(\exists \mathtt{L_0} \in \overline{\mathtt{L}}_1 . \mathtt{proceed} \notin \mathit{pmbody}(\mathtt{m}, \mathtt{C}, \mathtt{L_0})) \text{ or } \mathtt{class\ C\ \{..\ C_0\ m(..)\{..\}\ ..\}}}{\mathit{WP}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}_1, (\overline{\mathtt{L}}_1 ; \overline{\mathtt{L}}_2))} \qquad \text{(WP-Layer)}$$

$$\frac{\mathtt{C} \lhd \mathtt{D} \qquad \mathit{WP}(\mathtt{m}, \mathtt{D}, (\overline{\mathtt{L}}_1 ; \overline{\mathtt{L}}_2), (\overline{\mathtt{L}}_1 ; \overline{\mathtt{L}}_2))}{\mathit{WP}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}_1, (\overline{\mathtt{L}}_1 ; \overline{\mathtt{L}}_2))} \qquad \text{(WP-Super)}$$

Given the typing rule for run-time expressions, we can state the type soundness theorem below.

Theorem 1 (Subject Reduction). *Suppose given CT and LT are well-formed. If* $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\{\overline{\mathtt{L}}\}$ *wf and* $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$*, then* $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}' : \mathtt{S}$ *for some* $\mathtt{S}$ *such that* $\mathtt{S} \mathrel{<:} \mathtt{T}$.

Theorem 2 (Progress). *Suppose given CT and LT are well-formed. If* $\bullet; \{\overline{\mathtt{L}}\}; \bullet \vdash \mathtt{e} : \mathtt{T}$ *and* $\{\overline{\mathtt{L}}\}$ *wf, then* $\mathtt{e}$ *is a value or* $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$ *for some* $\mathtt{e}'$.

[2] The previous type system for ContextFJ [11] deals with `ensure`, an activation mechanism with a semantics slightly different from `with`, and T-InvkA is simpler. Further discussions on making `proceed` and `with` typesafe can be found in [12].

Theorem 3 (Type Soundness). *If* $\vdash (CT, LT, \mathtt{e}) : \mathtt{T}$ *and* `e` *reduces to a normal form under the empty set of layers, then the normal form is* $\mathtt{new\ S(\overline{v})}$ *for some* $\overline{\mathtt{v}}$ *and* `S` *such that* `S <: T`.

4 Related Work

Our work is a direct descendant of Igarashi, Hirschfeld, and Masuhara [10, 11], where a tiny COP language ContextFJ is developed and its type system is proved to be sound. ContextFJ is not equipped with layer inheritance, layer subtyping, or first-class layers but allows baseless methods to be declared in the second type system [11], in which `requires` declarations are first introduced into COP.

There are many type systems proposed for advanced composition mechanisms such as mixins [4, 8], traits [16], open classes (a.k.a. inter-type declarations) [6], and revisers [5]. A common idea is to let programmers declare dependency between modules as required interfaces; our `requires` declarations basically follow it. In most work, however, composition is done at compile or link time unlike COP languages. So, it is interesting that the same idea works even for dynamic composition found in COP languages.

Kamina and Tamai [15] propose McJava, in which mixin-based composition can be deferred to object instantiation. In fact, `new` expressions can specify a class and mixins to instantiate an object. So, the type of an object also consists of a class name and a sequence of mixin names. Whereas composition is per-instance basis in McJava, it is global in ContextFJ$_{<:}$. However, in McJava, composition cannot be changed once an object is instantiated.

Drossopoulou et al. [7] proposed *Fickle*$_{\mathtt{II}}$, a class-based object-oriented language with *dynamic reclassification*, which allows an object to change its class at run time. Their idea of root classes, which serve as interface, is similar to our swappable layers; their restriction that state classes cannot be used as type for fields is similar to ours that a sublayer of a swappable cannot be `require`d by any other layer.

Bettini et al. [3] developed a type system for *dynamic trait replacement*, which allows methods in an object to be exchanged at run time. They introduce the notion of *replaceable* to describe the signatures of replaceable methods; a replaceable appears as part of the type of an object and the trait to replace methods of the object has to provide the methods in that replaceable. The roles of replaceables and traits are somewhat similar to those of swappable layers, which provide interfaces common to swapped layers, and sublayers of swappable.

5 Concluding Remarks

We have developed a formal type system for a small COP language with layer inheritance, layer subtyping, swappable layers, and first-class layers, and shown that the type system is sound with respect to the operational semantics. As in previous work, `requires` declarations are important to guarantee safety in the presence of baseless methods. Subtyping for first-class layers is subtle because there are two kinds of substitutability. We have introduced weak subtyping for checking whether a `requires` clause is satisfied and normal subtyping for usual substitutability.

We are working on implementing the type system on top of the existing JCop compiler but there are many other features that are not modelled in our calculus. We briefly discuss how our type system can be extended to these features.

In JCop, a layer definition can contain field and (ordinary) method declarations so that a layer instance can act just like an ordinary object. Typechecking accesses to these members of layer instances is the same as ordinary objects. If we model fields of layer instances, we will have to modify the reduction relation so that the sequence of activated layers consists of layer instances (with their field values) rather than layer names.

JCop provides special variable `thislayer`, which can be used in partial methods and is similar to `this` of classes. It represents the layer instance in which the invoked partial method is found at run time and can be used to access fields and methods of that layer instance. In operational semantics, the layer instance would be substituted for `thislayer`, similarly to `this`. Typing `thislayer` is also similar to `this` in the sense that it is given the name of the layer in which it appears but `thislayer` cannot be used for layer activation because, at run time, it may be bound to an instance of a *weak* subtype.

JCop also introduces `superproceed()` call, which can be used in a partial method and invokes a superlayer's partial method that is overridden by the partial method. Similarly to `super` calls in Java, the destination of `superproceed()` is known statically, so it is easy to typecheck.

We have not fully investigated the interaction between our type system with other features in Java, such as concurrency, generics, and lambda, although we expect most of them are orthogonal.

Acknowledgments. We thank Tomoyuki Aotani, Malte Appeltauer, Robert Hirschfeld, and Tetsuo Kamina for valuable discussions on the subject. We appreciate valuable comments and suggestions from the anonymous reviewers. This work was supported in part by Kyoto University Design School (Inoue) and MEXT KAKENHI Grant Number 23220001 (Igarashi).

References

1. M. Appeltauer and R. Hirschfeld. *The JCop language specification: Version 1.0, April 2012.* Number 59. Universitätsverlag Potsdam, 2012.
2. M. Appeltauer, R. Hirschfeld, and J. Lincke. Declarative layer composition with the JCop programming language. *Journal of Object Technology*, 12, 2013.
3. L. Bettini, S. Capecchi, and F. Damiani. On flexible dynamic trait replacement for Java-like languages. *Science of Computer Programming*, 78(7):907–932, 2013.
4. V. Bono, A. Patel, and V. Shmatikov. A core calculus of classes and mixins. In *ECOOP'99–Object-Oriented Programming*, pages 43–66. Springer, 1999.
5. S. Chiba, A. Igarashi, and S. Zakirov. Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In *Proc. of the ACM OOPSLA*, pages 539–554, 2010.

6. C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Prog. Lang. Syst.*, 28(3):517–575, 2006.
7. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object reclassification: Fickle$_{\text{II}}$. *ACM Trans. Prog. Lang. Syst.*, 24(2):153–191, 2002.
8. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. of the ACM POPL*, pages 171–183. ACM, 1998.
9. R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.
10. R. Hirschfeld, A. Igarashi, and H. Masuhara. ContextFJ: A minimal core calculus for context-oriented programming. In *Proc. of Foundations of Aspect-Oriented Languages (FOAL)*, Mar. 2011.
11. A. Igarashi, R. Hirschfeld, and H. Masuhara. A type system for dynamic layer composition. In *Proc. of FOOL*, Oct. 2012.
12. A. Igarashi, H. Inoue, R. Hirschfeld, and H. Masuhara. ContextFJ: A minimal calculus for context-oriented programming, 2015. In preparation for submission.
13. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
14. H. Inoue, A. Igarashi, M. Appeltauer, and R. Hirschfeld. Towards type-safe JCop: A type system for layer inheritance and first-class layers. In *Proc. of the Workshop on Context-Oriented Programming*, pages 7:1–7:6. ACM, 2014.
15. T. Kamina and T. Tamai. McJava–a design and implementation of Java with mixin-types. In *Proc. of APLAS*, pages 398–414, 2004.
16. L. Liquori and A. Spiwack. FeatherTrait: A modest extension of Featherweight Java. *ACM Trans. Prog. Lang. Syst.*, 30(2):11, 2008.
17. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

$\boxed{\mathtt{PM\ ok\ in\ L}}$

$$\frac{\mathtt{L\ req}\ \Lambda \qquad \mathtt{L.C.m}; \Lambda \cup \{\mathtt{L}\}; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{S}_0 \qquad \mathtt{S}_0 \mathrel{<:} \mathtt{T}_0}{\mathtt{T}_0\ \mathtt{C.m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\ \{\ return\ e}_0\mathtt{;\ \}\ ok\ in\ L}} \quad \text{(T-PMethod)}$$

$\boxed{\mathtt{LA\ ok}}$

$$\frac{\mathtt{L}'\ \mathtt{req}\ \Lambda' \qquad \{\overline{\mathtt{L}}\} \mathrel{<:}_w \Lambda' \qquad \overline{\mathtt{PM}}\ \mathtt{ok\ in\ L}}{\mathtt{layer\ L\ req}\ \overline{\mathtt{L}} \triangleleft \mathtt{L}'\ \{\ \overline{\mathtt{PM}}\ \}\ \mathtt{ok}} \quad \text{(T-Layer)}$$

$\boxed{\mathit{override}(\mathit{CT}, \mathit{LT})}$

$$\frac{\begin{array}{l}\forall \mathtt{m}, \mathtt{C}, \overline{\mathtt{T}}, \mathtt{T}_0, \overline{\mathtt{S}}, \mathtt{S}_0.\ \text{if } \mathit{LT}(\mathtt{L}_1)(\mathtt{C.m}) = \mathtt{T}_0\ \mathtt{m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{...\}} \\ \qquad\qquad\qquad \text{and } \mathit{LT}(\mathtt{L}_2)(\mathtt{C.m}) = \mathtt{S}_0\ \mathtt{m(}\overline{\mathtt{S}}\ \overline{\mathtt{y}}\mathtt{)\{...\}}, \text{then } \overline{\mathtt{T}}, \mathtt{T}_0 = \overline{\mathtt{S}}, \mathtt{S}_0\end{array}}{\mathit{noconflict}(\mathtt{L}_1, \mathtt{L}_2)}$$

$$\frac{\begin{array}{l}\forall \mathtt{m}, \overline{\mathtt{T}}, \mathtt{T}_0, \overline{\mathtt{S}}, \mathtt{S}_0, \overline{\mathtt{x}}.\ \text{if } \mathit{LT}(\mathtt{L})(\mathtt{C.m}) = \mathtt{S}_0\ \mathtt{m(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\{...\}} \\ \qquad\qquad\qquad \text{and } \mathit{mtype}(\mathtt{m}, \mathtt{C}, \emptyset, \mathit{dom}(\mathit{LT})) = \overline{\mathtt{T}} \rightarrow \mathtt{T}_0, \text{then } \overline{\mathtt{T}}, \mathtt{T}_0 = \overline{\mathtt{S}}, \mathtt{S}_0\end{array}}{\mathit{override}^h(\mathtt{L}, \mathtt{C})}$$

$$\frac{\begin{array}{l}\forall \mathtt{m}, \mathtt{D}, \overline{\mathtt{T}}, \mathtt{T}_0, \overline{\mathtt{S}}, \mathtt{S}_0. \text{if } \mathtt{class\ C} \triangleleft \mathtt{D\ \{...\ S_0\ m(}\overline{\mathtt{S}}\ \mathtt{x)\{...\}...\}} \\ \qquad\qquad\qquad \text{and } \mathit{mtype}(\mathtt{m}, \mathtt{D}, \mathit{dom}(\mathit{LT}), \mathit{dom}(\mathit{LT})) = \overline{\mathtt{T}} \rightarrow \mathtt{T}_0, \\ \qquad\quad \text{then } \overline{\mathtt{T}} = \overline{\mathtt{S}} \text{ and } \mathtt{S}_0 \mathrel{<:} \mathtt{T}_0\end{array}}{\mathit{override}^v(\mathtt{C})}$$

$\boxed{\vdash (\mathit{CT}, \mathit{LT}, \mathtt{e}) : \mathtt{T}}$

$$\frac{\begin{array}{c}\forall \mathtt{C} \in \mathit{dom}(\mathit{CT}).\mathit{CT}(\mathtt{C})\ \mathtt{ok} \qquad \forall \mathtt{L} \in \mathit{dom}(\mathit{LT}).\mathit{LT}(\mathtt{L})\ \mathtt{ok} \\ \bullet; \emptyset; \bullet \vdash \mathtt{e} : \mathtt{T} \qquad \mathit{noconflict}(\mathtt{L}_1, \mathtt{L}_2) \\ \forall \mathtt{C} \in \mathit{dom}(\mathit{CT}).\mathtt{L} \in \mathit{dom}(\mathit{LT}).\mathit{override}^h(\mathtt{L}, \mathtt{C}) \\ \forall \mathtt{C} \in \mathit{dom}(\mathit{CT}).\mathit{override}^v(\mathtt{C})\end{array}}{\vdash (\mathit{CT}, \mathit{LT}, \mathtt{e}) : \mathtt{T}} \quad \text{(T-Prog)}$$

Fig. 7. ContextFJ$_{<:}$: Method/Class/Layer/Program typing.