ContextFJ: A Minimal Core Calculus for Context-oriented Programming

Robert Hirschfeld
Hasso-Plattner-Institut Potsdam
hirschfeld@hpi.uni-potsdam.de

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

Hidehiko Masuhara
The University of Tokyo
masuhara@acm.org

Abstract

We develop a minimal core calculus called ContextFJ to model language mechanisms for context-oriented programming (COP). Unlike other formal models of COP, ContextFJ has a direct operational semantics that can serve as a precise description of the core of COP languages. We also discuss a simple type system that helps to prevent undefined methods from being accessed via `proceed`.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Language, Theory

Keywords Context-oriented programming, operational semantics

1. Introduction

Context-oriented programming (COP) is an approach to improving modularity of behavioral variations that depend on the dynamic context of the execution environment [7]. In traditional programming paradigms, such behavioral variations tend to be scattered over several modules, and system architectures that support their dynamic composition are often complicated.

Many COP extensions including those designed on top of Java [2], Smalltalk [6], Common Lisp [4] and JavaScript [10], are based on object-oriented programming languages and introduce *layers* of *partial methods* for defining and organizing behavioral variations, and *layer activation mechanisms* for layer selection and composition. A partial method in a layer is a method that can run before, after, or around the same (partial) method defined in a different layer or a class. A layer groups related partial methods and can be (de)activated at run-time. It so contributes to the specific behavior of a set of objects in response to messages sent and received.

In this paper, we report on our ongoing work on a formal model of core language features of COP. We present a small calculus called ContextFJ that is an extension of Featherweight Java (FJ) [8]. As the first step, we severely limit the supported language features in ContextFJ so as to make the calculus simple yet expressive enough to add more interesting features in future. In addition to the usual features of FJ, it supports around-type (i.e., overriding) partial methods, dynamic activation and deactivation of layers, and `proceed` and `super` calls. Aside from the Java features FJ already omits, ContextFJ does not (yet) support first-class layers that can be passed around via arguments or variables, stateful layers that allow to share state between partial methods or associated objects, and before and after methods.

We give a small-step reduction semantics to model the behavior of COP programs *directly* without using translation to a language without COP features. As far as we know, this is a first direct semantics of COP features. Such direct semantics can serve as a precise specification of the core of COP languages.

We also discuss a type system for ContextFJ. As usual, the task of a type system is to statically guarantee the absence of run-time field-not-found and method-not-found errors. However, since in COP the presence of a method definition in a given class may depend on whether a particular layer is activated or not, this task is much harder. As a starting point, we develop a simple but restrictive type system, which allows partial methods only for existing methods in classes. We state that this simple type system is sound; a full version of the paper, available at `http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/papers/`, includes proofs.

The rest of the paper is organized as follows. We first start with reviewing the language mechanisms for COP in Section 2. Section 3 defines the syntax and operational semantics of ContextFJ and Section 4 defines a simple type system. We discuss related and future work in Section 5.

2. Language Constructs for COP

2.1 Partial Method Definitions and Layers

We briefly review basic constructs along with their usage. In our example, behavioral variations are expressed as partial method definitions and related method definitions are grouped in layers.

```
class Person {
  String name, residence, employer;
  Person(String _name, String _residence,
         String _employer) {
    name = _name; residence = _residence;
    employer = _employer;
  }
  String toString() { return "Name: " + name; }
  layer Contact {
    String toString() {
      return
        proceed() + "; Residence: " + residence;
  }}
  layer Employment {
    String toString() {
      return proceed() + "; Affil.: " + employer;
}}}
```


FOAL'11, March 21, 2011, Pernambuco, Brazil.

Class Person defines three fields name, residence, and employer (all of type String) that will be initialized during object creation. It also defines the toString() method to show object-specific information referred to in its fields.

The base definition incorporates the name field in the textual representation. It belongs to the so-called base layer and with that is effective for all instances of Person (and its subclasses).

The one refinement is implemented as a partial definition of the same method in class Person and associated with the Contact layer. (COP layers are usually used to group more than one partial method definition, but as an illustrating example for ContextFJ, having one layer holding on to one partial method definition will suffice.) Our partial definition of toString() appends information from the residence field that might be useful for further correspondence. It belongs to the Contact layer and is only effective if Contact is active.

The other refinement is associated with the Employment layer and differs from the second refinement in the field, now employer, that it deals with.

In our example, the partial definitions of toString() call the special method proceed() to invoke other partial definitions of toString() contributed by layers that were already active before the activation of the Contact or the Employment layer, or to invoke the base-level implementation of this method (here provided by class Person).

proceed(...) is similar to super as it allows for calling other behavior previously defined in the composition path: Whereas super changes the starting point of the method lookup to the superclass of the class the (partial) method was defined in, proceed(...) will try to find the next partial or base-level definition of the same method. If proceed(...) cannot find such a partial method in the current receiver class or the active layers associated with it, lookup continues in the superclass of the current lookup class. Lookup is statically guaranteed to succeed as we will see in Section 4.

2.2 Layer Activation and Deactivation

Layers are explicitly activated or deactivated using the with and without constructs respectively. In the following transcript, we show the application of theses constructs to an instance of class Person.

```
Person atsushi =
  new Person("Atsushi", "Kyoto", "Kyodai");
```

Printing our object to the standard output stream via println(...) with *no layers activated* leads to directly calling our base-level implementation that returns a description covering only the name of the object.

```
System.out.println(atsushi);
=> "Name: Atsushi"
```

However, if we put a with statement activating the Contact layer around this code, the same attempt to print out our object leads to first calling our partial definition of toString() contributed by the Contact layer which is responsible for providing contact information from the residence field, and then calling our base-level implementation providing the person's name.

```
with (Contact) { System.out.println(atsushi); }
=> "Name: Atsushi; Residence: Kyoto"
```

The nesting of with (or without) statements leads to nested layer activations, where "inner" layers gain precedence over "outer" ones.

```
with (Employment) {
  with (Contact) { System.out.println(atsushi); }
}
=> "Name: Atsushi; Residence: Kyoto; Affil.: Kyodai"
```

With that, the change of the order of with or without statements corresponds directly to the partial method definitions obtained by the method lookup.

```
with (Contact) {
  with (Employment) { System.out.println(atsushi); }
}
=> "Name: Atsushi; Affil.: Kyodai; Residence: Kyoto"
```

Previously activated layers using with can be deactivated by without and vice versa.

```
with (Contact) {
  without (Contact) { System.out.println(atsushi); }
}
=> "Name: Atsushi"
```

An attempt to deactivate a layer that is not active will not affect the current layer composition.

```
without (Contact) { System.out.println(atsushi); }
=> "Name: Atsushi"
```

As in most COP language extensions and also in ours, layer compositions are effective for the *dynamic extent* of the execution of the code block enclosed by their corresponding with or without statements[1].

3. Syntax and Semantics of ContextFJ

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; and x and y over variables, which contain a special variable this. The abstract syntax of ContextFJ is given as follows:

CL	::=	class C ◁ C { $\overline{\texttt{C}}$ $\overline{\texttt{f}}$; K $\overline{\texttt{M}}$ }	(*classes*)
K	::=		(*constructors*)
		C($\overline{\texttt{C}}$ $\overline{\texttt{f}}$){ super($\overline{\texttt{f}}$); this.$\overline{\texttt{f}}$ = $\overline{\texttt{f}}$; }	
M	::=	C m($\overline{\texttt{C}}$ $\overline{\texttt{x}}$){ return e; }	(*methods*)
e, d	::=	x | e.f | e.m($\overline{\texttt{e}}$) | new C($\overline{\texttt{e}}$)	(*expressions*)
		| with L e | without L e	
		| proceed($\overline{\texttt{e}}$) | super.m($\overline{\texttt{e}}$)	
		| new C($\overline{\texttt{v}}$)<C,$\overline{\texttt{L}}$,$\overline{\texttt{L}}$>.m($\overline{\texttt{e}}$)	
v, w	::=	new C($\overline{\texttt{v}}$)	(*values*)

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 $\overline{\texttt{C}}$, $\overline{\texttt{x}}$, $\overline{\texttt{e}}$, and so on. The empty sequence is denoted by •. We also abbreviate pairs of sequences, writing "$\overline{\texttt{C}}$ $\overline{\texttt{f}}$" for "$\texttt{C}_1$ $\texttt{f}_1, \cdots, \texttt{C}_n$ $\texttt{f}_n$", where n is the length of $\overline{\texttt{C}}$ and $\overline{\texttt{f}}$, and similarly "$\overline{\texttt{C}}$ $\overline{\texttt{f}}$;" as shorthand for the sequence of declarations "$\texttt{C}_1$ $\texttt{f}_1$;...$\texttt{C}_n$ $\texttt{f}_n$;" and "this.$\overline{\texttt{f}}$=$\overline{\texttt{f}}$;" for "this.$\texttt{f}_1$=$\texttt{f}_1$;...;this.$\texttt{f}_n$=$\texttt{f}_n$;". We use commas and semicolons for concatenations. Sequences of field declarations, parameter names, layer names, and method declarations are assumed to contain no duplicate names.

A class definition CL consists of its name, its superclass name, field declarations $\overline{\texttt{C}}$ $\overline{\texttt{f}}$, a constructor K, and method definitions $\overline{\texttt{M}}$. A constructor K is a trivial one that takes initial values of all fields and sets them to the corresponding fields. Unlike the examples in

[1] Variants of COP languages allow to manage layer compositions on a per-instance basis [9, 10], which is left as future work in the paper.

the last section, we do not provide syntax for layers; partial methods are registered in a partial method table, explained below. A method M takes $\overline{\mathtt{x}}$ as arguments and returns the value of expression e. As ContextFJ is a functional calculus like FJ, the method body consists of a single return statement and all constructs including with and without return values. An expression e can be a variable, field access, method invocation, object instantiation, layer activation/deactivation, proceed/super call, or a special expression new C($\overline{\mathtt{v}}$)<C,$\overline{\mathtt{L}}$,$\overline{\mathtt{L}}$>.m($\overline{\mathtt{e}}$), which will be explained shortly. A value is an object of the form new C($\overline{\mathtt{v}}$).

The expression new C($\overline{\mathtt{v}}$)<D,$\overline{\mathtt{L}}'$,$\overline{\mathtt{L}}$>.m($\overline{\mathtt{e}}$), where $\overline{\mathtt{L}}'$ is assumed to be a prefix of $\overline{\mathtt{L}}$, is a special run-time expression and not supposed to appear in classes. It basically means that m is going to be invoked on new C($\overline{\mathtt{v}}$). The annotation <D,$\overline{\mathtt{L}}'$,$\overline{\mathtt{L}}$>, which is used to model super and proceed, indicates where method lookup should start. More concretely, the triple <D,$(\mathtt{L}_1;\cdots;\mathtt{L}_i)$,$(\mathtt{L}_1;\cdots;\mathtt{L}_n)$> ($i \leq n$) means that the search for the method definition will start from class D of layer $\mathtt{L}_i$. So, for example, the usual method invocation new C($\overline{\mathtt{v}}$).m($\overline{\mathtt{e}}$) (without annotation) is semantically equivalent to new C($\overline{\mathtt{v}}$)<C,$\overline{\mathtt{L}}$,$\overline{\mathtt{L}}$>.m($\overline{\mathtt{e}}$), where $\overline{\mathtt{L}}$ is the active layers when this invocation is to be executed. This triple also plays the role of a "cursor" in the method lookup procedure and proceeds as follows

$$
\begin{array}{ll}
 & \texttt{<D,}(\mathtt{L}_1;\cdots;\mathtt{L}_i)\texttt{,}(\mathtt{L}_1;\cdots;\mathtt{L}_n)\texttt{>} \\
\Rightarrow & \texttt{<D,}(\mathtt{L}_1;\cdots;\mathtt{L}_{i-1})\texttt{,}(\mathtt{L}_1;\cdots;\mathtt{L}_n)\texttt{>} \quad \Rightarrow \quad \cdots \\
\Rightarrow & \texttt{<D, •,}(\mathtt{L}_1;\cdots;\mathtt{L}_n)\texttt{>} \\
\Rightarrow & \texttt{<E,}(\mathtt{L}_1;\cdots;\mathtt{L}_n)\texttt{,}(\mathtt{L}_1;\cdots;\mathtt{L}_n)\texttt{>} \quad \text{(E is a direct superclass of D)} \\
\Rightarrow & \texttt{<E,}(\mathtt{L}_1;\cdots;\mathtt{L}_{n-1})\texttt{,}(\mathtt{L}_1;\cdots;\mathtt{L}_n)\texttt{>} \quad \Rightarrow \quad \cdots
\end{array}
$$

until the method definition is found. Notice that the third element is needed when the method is not found in D in any layer including the base: the search continues to layer $\mathtt{L}_n$ of D's direct superclass.

With the help of this form, we can give a semantics of super and proceed by simple substitution-based reduction. For example, consider method invocation new C().m($\overline{\mathtt{v}}$). As in FJ, this expression reduces to the method body where parameters and this are replaced with arguments $\overline{\mathtt{v}}$ and the receiver new C(), respectively. Now, what happens to super in the method body? It cannot be replaced with the receiver new C() since it would confuse this and super. Method lookup for super is different from usual (virtual) method lookup in that it has to start from the direct superclass of *the class in which* super *appears*. So, if the method body containing super.n() is found in class D, then the search for n has to start from the direct superclass of D. To express this fact, we replace super with new C()<E,...> where E is the direct superclass of D. We can deal with proceed similarly. Suppose the method body is found in layer $\mathtt{L}_i$ in D. Then, proceed($\overline{\mathtt{e}}$) is replaced with new C()<D,$(\mathtt{L}_1;\cdots;\mathtt{L}_{i-1})$,$\overline{\mathtt{L}}$>.m($\overline{\mathtt{e}}$), where $\mathtt{L}_1;\cdots;\mathtt{L}_{i-1}$ are layers activated before $\mathtt{L}_i$.

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

1. $CT(\mathtt{C}) = \texttt{class C ...}$ for any $\mathtt{C} \in dom(CT)$.
2. $\texttt{Object} \notin dom(CT)$.
3. For every class name C (except Object) appearing anywhere in CT, we have $\mathtt{C} \in dom(CT)$;
4. There are no cycles in the transitive closure of the extends clauses.
5. $PT(\mathtt{m},\mathtt{C},\mathtt{L}) = \texttt{... m(...)\{...\}}$ for any $(\mathtt{m},\mathtt{C},\mathtt{L}) \in dom(PT)$.

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

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

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

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

$$\frac{\texttt{class C} \lhd \texttt{D \{ ... C}_0\ \texttt{m(}\overline{\mathtt{C}}\ \overline{\mathtt{x}}\texttt{)\{ return e; \} ...\}}}{\mathit{mbody}(\mathtt{m},\mathtt{C},\bullet,\overline{\mathtt{L}}) = \overline{\mathtt{x}}\texttt{.e in C},\bullet}$$

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

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

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

Figure 1. ContextFJ: Lookup functions.

Lookup functions. As in FJ, we define a few auxiliary functions to look up field and method definitions. They are defined by the rules in Figure 1. The function $\mathit{fields}(\mathtt{C})$ returns a sequence $\overline{\mathtt{C}}\ \overline{\mathtt{f}}$ of pairs of a field name and its type by collecting all field declarations from C and its superclasses. The function $\mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}_1,\overline{\mathtt{L}}_2)$ returns the parameters and body $\overline{\mathtt{x}}\texttt{.e}$ of method m in class C when the search starts from $\overline{\mathtt{L}}_1$; the other layer names $\overline{\mathtt{L}}_2$ keep track of the layers that are activated when the search initially started. It also returns the information on where the method has been found—the information will be used in reduction rules to deal with proceed and super. As we mentioned already, the method definition is searched for in class C in all activated layers and the base definition and, if there is none, then the search continues to C's superclass. By reading the rules in a bottom-up manner, we can read off the recursive search procedure. The first rule means that m is found in the base class definition C (notice the third argument is •) and the second that m is found in layer $\mathtt{L}_0$. The third rule, which deals with the situation where m is not found in a base class (expressed by the condition $\mathtt{m} \notin \overline{\mathtt{M}}$), motivates the fourth argument of *mbody*. The search goes on to C's superclass D and has to take all activated layers into account; so, $\overline{\mathtt{L}}$ is copied to the third argument in the premise. The fourth rule means that, if C of $\mathtt{L}_0$ does not have m, then the search goes on to the next layer (in $\overline{\mathtt{L}}'$) leaving the class name unchanged.

3.2 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 $\overline{\mathtt{L}}$". Here, $\overline{\mathtt{L}}$ do not contain duplicate names, as we noted earlier. The main rules are shown in Figure 2.

The first four rules are the main computation rules for field access and method invocation. The first rule for field access is straightforward: *fields* tells which argument to new C(..) corresponds to $\mathtt{f}_i$. The next three rules are for method invocation. The second rule is for method invocation where the cursor of the method lookup procedure has not been "initialized"; the cursor is set to be at the receiver's class and the currently activated layers. In the third rule, the receiver is new C($\overline{\mathtt{v}}$) and <C$'$,$\overline{\mathtt{L}}'$,$\overline{\mathtt{L}}$> is the location of the cursor. When the method body is found in the base-layer class

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\overline{\mathtt{L}} \vdash \mathtt{new\ C(\overline{v}).f}_i \longrightarrow \mathtt{v}_i} \qquad \frac{\overline{\mathtt{L}} \vdash \mathtt{new\ C(\overline{v})\langle C,\overline{L},\overline{L}\rangle.m(\overline{w})} \longrightarrow \mathtt{e}}{\overline{\mathtt{L}} \vdash \mathtt{new\ C(\overline{v}).m(\overline{w})} \longrightarrow \mathtt{e}}$$

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

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

$$\frac{\mathit{remove}(\mathtt{L}, \overline{\mathtt{L}}) = \overline{\mathtt{L}}' \qquad \overline{\mathtt{L}}'; \mathtt{L} \vdash \mathtt{e} \longrightarrow \mathtt{e}'}{\overline{\mathtt{L}} \vdash \mathtt{with\ L\ e} \longrightarrow \mathtt{with\ L\ e}'}$$

$$\frac{\mathit{remove}(\mathtt{L}, \overline{\mathtt{L}}) = \overline{\mathtt{L}}' \qquad \overline{\mathtt{L}}' \vdash \mathtt{e} \longrightarrow \mathtt{e}'}{\overline{\mathtt{L}} \vdash \mathtt{without\ L\ e} \longrightarrow \mathtt{without\ L\ e}'}$$

$$\frac{}{\overline{\mathtt{L}} \vdash \mathtt{with\ L\ v} \longrightarrow \mathtt{v}} \qquad \frac{}{\overline{\mathtt{L}} \vdash \mathtt{without\ L\ v} \longrightarrow \mathtt{v}}$$

Figure 2. ContextFJ: Reduction rules.

$\mathtt{C}''$ (denoted by "$\mathtt{in\ C}'', \bullet$"), the whole expression reduces to the method body where the formal parameters $\overline{\mathtt{x}}$ and `this` are replaced by the actual arguments $\overline{\mathtt{w}}$ and the receiver, respectively. Furthermore, `super` is replaced by the receiver with the cursor pointing to the superclass of $\mathtt{C}''$. The fourth rule, which is similar to the third, deals with the case where the method body is found in layer $\mathtt{L}_0$ in class $\mathtt{C}''$. In this case, `proceed` in the method body is replaced with the invocation of the same method, where the receiver's cursor points to the next layer $\overline{\mathtt{L}}''$ (dropping $\mathtt{L}_0$). Since the meaning of the annotated invocation is not affected by the layers in the context (note that $\overline{\mathtt{L}}'''$ are not significant in these rules), the substitution for `super` and `proceed` also means that their meaning is the same throughout a given method body, even when they appear inside `with` and `without`. Note that, unlike FJ, reduction in ContextFJ is call-by-value, requiring receivers and arguments to be values. This evaluation strategy reflects the fact that arguments should be evaluated under the caller-side context.

The following rules are related to context manipulation. The fifth rule means that `e` in `with L e` should be executing by activating `L`. The auxiliary function $\mathit{remove}(\mathtt{L}, \overline{\mathtt{L}})$, which removes `L` from $\overline{\mathtt{L}}$ (or returns $\overline{\mathtt{L}}$ if `L` is not in $\overline{\mathtt{L}}$), is used to avoid duplication of `L`. The next rule is similar: `e` is evaluated under the context where `L` is absent. The last two rules mean that, once the evaluation of the body of `with`/`without` is finished, it returns the value of the body.

There are other trivial congruence rules to allow subexpressions to reduce, but we omit them for brevity.

4. Type System

As usual, the role of a type system is to guarantee type soundness, namely, to prevent statically field-not-found and method-not-found errors from happening at run-time. In ContextFJ, it also means that a type system should ensure that every `proceed()` or `super()` call succeeds. However, it is not trivial to ensure this property, due to the dynamic nature of layer activation—the existence of a method definition in a given class may depend on whether a particular layer is activated.

Here, we give a simple type system, which is mostly a straightforward extension of FJ's type system but prohibits layers from introducing new methods that do not exist in the base-layer class—in other words, every partial method has to override a method of the same name in the base-layer class. As a result, the function *mtype* to retrieve a method type is the same as FJ's: it takes a method name and a class name as arguments and returns a pair, written $\overline{\mathtt{C}}\to\mathtt{C}_0$, of a sequence of the argument types $\overline{\mathtt{C}}$ and the return type $\mathtt{C}_0$. Its definition is given by the following rules.

$$\frac{\mathtt{class\ C}\lhd\ \mathtt{D\ \{...\ C}_0\ \mathtt{m(\overline{C}\ \overline{x})\{\ return\ e;\ \}\ ...\}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{C}}\to\mathtt{C}_0}$$

$$\frac{\mathtt{class\ C}\lhd\ \mathtt{D\ \{...\ \overline{M}\ \}} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{C}}\to\mathtt{C}_0}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{C}}\to\mathtt{C}_0}$$

Subtyping. The subtyping relation $\mathtt{C} <: \mathtt{D}$ is defined as the reflexive and transitive closure of the `extends` clauses.

$$\frac{}{\mathtt{C} <: \mathtt{C}} \qquad \frac{\mathtt{C} <: \mathtt{D} \qquad \mathtt{D} <: \mathtt{E}}{\mathtt{C} <: \mathtt{E}} \qquad \frac{\mathtt{class\ C}\lhd\ \mathtt{D\ \{...\}}}{\mathtt{C} <: \mathtt{D}}$$

Typing. The type judgment for expressions is of the form $\mathcal{L}; \Gamma \vdash \mathtt{e} : \mathtt{D}$, read "`e`, which appears in $\mathcal{L}$, is given type `D` under Γ". Here, Γ denotes a type environment, which assigns types to variables—more formally, it is a finite mapping from variables to class names. $\mathcal{L}$, which stands for the location where `e` appears, is either $\bullet$, which means the top-level (i.e., under execution), `C.m`, which means method `m` in base class `C`, or `L.C.m`, which means `m` in class `C` in layer `L`. It is used in the typing rules for `proceed()` and `super()` calls. The type judgment for methods is of the form either `M ok in C`, read "method `M` is well-formed in base-layer class `C`", or `M ok in L.C`, read "partial method `M` is well-formed in layer `L` of class `C`." Finally, the type judgment for classes is of the form `CL ok`, read "class `CL` is well-formed." The typing rules are given in Figure 3.

The typing rules for expressions are straightforward. The first four rules for variables, field access, method invocation, and object instantiation are the same as those in FJ (except $\mathcal{L}$). The fifth and sixth rules for `with` and `without`, respectively, mean that a layer (de)activation is well typed if its body is well typed. The next rule means that $\mathtt{super.m}'(\overline{\mathtt{e}})$ has to appear in a method definition in some class `C` (not at the top level) and the type of $\mathtt{m}'$ is retrieved from `C`'s superclass `E`. Otherwise, it is similar to the rule for method invocations. The rule for `proceed(`$\overline{\mathtt{e}}$`)` is similar. The expression has to appear in a partial method definition, hence the location should be `L.C.m`. The final rule combines the rules for object instantiation and method invocation. Although the run-time type of the receiver is $\mathtt{C}_0$, the current cursor is at class `D`, which is a superclass of $\mathtt{C}_0$. So, the type of `m` is retrieved from `D`.

The typing rules for method definitions are straightforward also. Both rules check that the method body is well typed under the assumption that formal parameters $\overline{\mathtt{x}}$ are given declared types $\overline{\mathtt{C}}$ and `this` is given the name of the class name where the method appears. The type of the method body has to be a subtype of the declared return type. One notable difference in these rules is in the last premise. The first rule for base-layer methods means that the method may or may not be overriding; if it is overriding, the usual overriding condition is checked. Note that we allow covariant overriding of the return type. On the other hand, the second rule for a partial method means that it *has to override* the base-layer method *with exactly the same type*. We cannot allow covariant overriding because the order of layer composition vary at run-time.

A program $(\mathit{CT}, \mathit{PT}, \mathtt{e})$ is well-formed if $\mathit{CT}(\mathtt{C})$ `ok` for any $\mathtt{C} \in \mathit{dom}(\mathit{CT})$ and $\mathit{PT}(\mathtt{m}, \mathtt{C}, \mathtt{L})$ `ok in L.C` for any $(\mathtt{m}, \mathtt{C}, \mathtt{L}) \in$

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

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

$$\frac{\mathcal{L};\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{mtype}(\mathtt{m},\mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{D}_0 \qquad \mathcal{L};\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} \qquad \overline{\mathtt{E}} <: \overline{\mathtt{D}}}{\mathcal{L};\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{D}_0}$$

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

$$\frac{\mathcal{L};\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0}{\mathcal{L};\Gamma \vdash \mathtt{with}\ \mathtt{L}\ \mathtt{e}_0 : \mathtt{C}_0} \qquad \frac{\mathcal{L};\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0}{\mathcal{L};\Gamma \vdash \mathtt{without}\ \mathtt{L}\ \mathtt{e}_0 : \mathtt{C}_0}$$

$$\frac{\mathcal{L} = \mathtt{C.m} \text{ or } \mathtt{L.C.m} \qquad \mathtt{class\ C} \lhd \mathtt{E}\ \{\ldots\} \qquad \mathit{mtype}(\mathtt{m}',\mathtt{E}) = \overline{\mathtt{D}} \to \mathtt{D}_0 \qquad \mathcal{L};\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} \qquad \overline{\mathtt{E}} <: \overline{\mathtt{D}}}{\mathcal{L};\Gamma \vdash \mathtt{super}.\mathtt{m}'(\overline{\mathtt{e}}) : \mathtt{D}_0}$$

$$\frac{\mathcal{L} = \mathtt{L.C.m} \qquad \mathit{mtype}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{D}} \to \mathtt{D}_0 \qquad \mathcal{L};\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} \qquad \overline{\mathtt{E}} <: \overline{\mathtt{D}}}{\mathcal{L};\Gamma \vdash \mathtt{proceed}(\overline{\mathtt{e}}) : \mathtt{D}_0}$$

$$\frac{\mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \qquad \mathcal{L};\Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}} \qquad \mathtt{C}_0 <: \mathtt{D} \qquad \mathit{mtype}(\mathtt{m},\mathtt{D}) = \overline{\mathtt{F}} \to \mathtt{F}_0 \qquad \mathcal{L};\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{F}} \qquad \overline{\mathtt{E}} <: \overline{\mathtt{F}}}{\mathcal{L};\Gamma \vdash \mathtt{new}\ \mathtt{C}_0\mathtt{<D,}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}''\mathtt{>}(\overline{\mathtt{v}})\mathtt{.m}(\overline{\mathtt{e}}) : \mathtt{F}_0}$$

Method/class typing: $\boxed{\mathtt{M\ ok\ in\ C}}$ $\boxed{\mathtt{M\ ok\ in\ L.C}}$ $\boxed{\mathtt{CL\ ok}}$

$$\frac{\begin{array}{c}\mathtt{C.m};\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{D}_0 \\ \mathtt{D}_0 <: \mathtt{C}_0 \qquad \mathtt{class\ C} \lhd \mathtt{D}\ \{\ldots\} \\ \text{if } \mathit{mtype}(\mathtt{m},\mathtt{D}) = \overline{\mathtt{E}} \to \mathtt{E}_0, \text{ then } \overline{\mathtt{E}} = \overline{\mathtt{C}} \text{ and } \mathtt{C}_0 <: \mathtt{E}_0\end{array}}{\mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return\ e}_0\mathtt{;}\ \}\ \mathtt{ok\ in\ C}}$$

$$\frac{\begin{array}{c}\mathtt{L.C.m};\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{D}_0 \\ \mathtt{D}_0 <: \mathtt{C}_0 \qquad \mathit{mtype}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{C}} \to \mathtt{C}_0\end{array}}{\mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return\ e}_0\mathtt{;}\ \}\ \mathtt{ok\ in\ L.C}}$$

$$\frac{\mathtt{K} = \mathtt{C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}},\ \overline{\mathtt{C}}\ \overline{\mathtt{f}})\{\ \mathtt{super}(\overline{\mathtt{g}})\mathtt{;}\ \mathtt{this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}\mathtt{;}\ \} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{ok\ in\ C}}{\mathtt{class\ C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}\ \mathtt{K}\ \overline{\mathtt{M}}\ \}\ \mathtt{ok}}$$

Figure 3. ContextFJ: Typing rules.

$\mathit{dom}(PT)$ and $\bullet;\emptyset \vdash \mathtt{e} : \mathtt{C}$ for some $\mathtt{C}$, where $\emptyset$ is the empty type environment.

This type system is sound with respect to the operational semantics given in the last section:

THEOREM 1 (Subject Reduction). *Suppose given class and partial method tables are well-formed. If* $\bullet;\Gamma \vdash \mathtt{e} : \mathtt{C}$ *and* $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$*, then* $\bullet;\Gamma \vdash \mathtt{e}' : \mathtt{D}$ *for some* $\mathtt{D}$ *such that* $\mathtt{D} <: \mathtt{C}$*.*

THEOREM 2 (Progress). *Suppose given class and partial method tables are well-formed. If* $\bullet;\emptyset \vdash \mathtt{e} : \mathtt{C}$*, then either* $\mathtt{e}$ *is a value or* $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$ *for some* $\mathtt{e}'$*.*

5. Discussion

Related Work The operational semantics of *cj*, a context-oriented extension to the *j* language family, is expressed using a delegation-based calculus [14]. Another approach to providing an operational semantics of COP layer constructs and their application is based on graph transformations [11]. Both approaches to representing context-dependent behavior *encode* COP programs into more general calculi. Our semantics, on the other hand, *directly* expresses context-dependent behavior.

Feature-oriented programming (FOP) [3] and delta-oriented programming (DOP) [12] also advocate the use of layers or delta modules respectively to describe behavioral variations. In both approaches, various similar software artifacts are obtained by *statically* composing layers with base-level classes. Thus, formal models of FOP [1, 5] and DOP [13] typically give translational semantics. Since they usually allow layers to add new methods, type systems that guarantee the translated program to be well typed with respect to the base language's type system are more sophisticated than ours.

Future Work The present type system may be too restrictive since it does not allow layers to introduce new methods. We are currently working on a more sophisticated type system that does not prevent method introduction by exploring some ideas from type systems for FOP.

References

[1] Sven Apel, Christian Kästner, and Christian Lengauer. Feature Featherweight Java: a calculus for feature-oriented programming and stepwise refinement. In *GPCE*, 2008. doi:10.1145/1449913.1449931.

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

[3] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling stepwise refinement. *TSE*, 2004. doi:10.1109/TSE.2004.23.

[4] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming - an overview of ContextL. In *DLS*, 2005. doi:10.1145/1146841.1146842.

[5] Benjamin Delaware, William Cook, and Don Batory. A machine-checked model of safe composition. In *FOAL*, 2009. doi:10.1145/1509837.1509846.

[6] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An introduction to context-oriented programming with ContextS. In *GTTSE*, 2008.

[7] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *JOT*, 2008.

[8] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLAS*, 2001. doi:10.1145/503502.503505.

[9] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: A context-oriented programming language with declarative event-based context transition. In *Proc. of AOSD*, 2011. (to appear).

[10] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in ContextJS. *SCP*, 2010. doi:10.1016/j.scico.2010.11.013.

[11] Tim Molderez, Hans Schippers, Dirk Janssens, Michael Haupt, and Robert Hirschfeld. A platform for experimenting with language constructs for modularizing crosscutting concerns. In *WASDeTT*, 2010.

[12] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *SPLC*, 2010.

[13] Ina Schaefer, Lorenzo Bettini, and Ferruccio Damiani. Compositional type-checking for delta-oriented programming. In *AOSD*, 2011. (to appear).

[14] Hans Schippers, Dirk Janssens, Michael Haupt, and Robert Hirschfeld. Delegation-based semantics for modularizing crosscutting concerns. In *OOPSLA*, 2008. doi:10.1145/1449764.1449806.