A new model for Context-Oriented Programs

Mohamed A. El-Zawawy, Eisa A. Aleisa

1College of Computer and Information Sciences, Al Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh, Kingdom of Saudi Arabia
2Department of Mathematics, Faculty of Science, Cairo University, 12613, Giza, Egypt

Abstract: Context-oriented programming (COP) is a new technique for programming that allows changing the context in which commands execute as a program executes. Compared to object-oriented programming (aspect-oriented programming), COP is more flexible (modular and structured). This paper presents a precise syntax-directed operational semantics for context-oriented programming with layers, as realized by COP languages like ContextJ* and ContextL. Our language model is built on Java enriched with layer concepts and activation and deactivation of layer scopes. The paper also presents a static type system that guarantees that typed programs do not get stuck. Using the means of the proposed semantics, the mathematical correctness of the type system is presented in the paper.

Keywords: Context-oriented programming; operational semantics; type systems; layers activation and deactivation

1. Introduction

Modularity of performance alterations relies on the dynamic environment of program executions. Context-oriented programming (COP) (Hirschfeld, Costanza & Nierstrasz, 2008) emerged as a programming technique to enhance this modularity. Classically these performance alterations are distributed among program modules and usually complex engineering is necessary to back dynamic combination of the modules. Smalltalk (Golubski & Lippe, 1995), Java (Campione, Walrath & Huml, 2000), JavaScript (Flanagan, 2012), and Common Lisp (Costanza, Herzeel & D’Hondt, 2009) are examples of languages on which COP were established. The base languages for COP are typical object-oriented languages. Main features of COP include (a) layers of variant procedures for introducing and classifying performance alterations and (b) an instrument for layer activation to endorsement and composition. A variant procedure is a procedure that can be executed around, after, or before the same (variant) procedure defined in a different part (class or layer) of the program. A layer is a set of variant procedures. A layer can be (de)activated in main function. Layers are meant to determine the specific semantics of objects for adaption with different applications.

In this paper, we present a new model for COP. The proposed model has basic language features. The model has the advantage of extending directly over well-studied Java features. The model is in-complex yet articulates enough to include more language features. Besides typical Java features, the model provides overriding (i.e., around-type) variant procedures, layers activation and deactivation, and a call mechanism for proceed and super. This paper also presents an operation semantics that directly (without mapping to non-COP) models the meanings of basic COP constructs. For the core of COP languages, the proposed semantics can be used to provide precise specifications. The paper also presents a type system for COP. Typically; a type system statically ensures the absence of run-time errors such as procedure-not-found and field-not-found errors. Noticeably, establishing the type system is not an easy task because in COP the existence of a procedure definition in a class may well rely upon whether a specific layer is activated. The paper also provides a mathematical proof for the soundness of the type system based on the proposed operational semantics.

Example

Figure 1 provides a COP example. Class Cube defines three variables of type integer (length, width, and height) with a constructor for initialization. The class also includes the modify() procedure to modify different variables.

The first definition of modify() is the main one and modifies and shows length. This definition is included in the main layer which is effectual for all objects of Cube. The second definition of modify() is a refinement and is included in the layer Second_dim. This refinement modifies width and appends its new value (the second dimension of the cube) that might be needed for further calculations. This refinement is effective only when its layer is activated. The third definition of modify() is yet another refinement and is included in the layer Third_dim.

In the example of Figure 1, the refinements of modify() runs the command proceed(). This special command invokes all refinements of modify() included...
in layers already activated ahead of the activation of the Second_dim or Third_dim layer. This command also invokes the version of modify() included in the main layer. On the other hand, the super command included in our language model (Figure 2) starts the lookup for procedures from the super-class of the class containing the current procedure.

![Figure 1: A COP program](http://www.lifesciencesite.com)

The with and without constructs are used in COP for layer activation and deactivation, respectively. We show their use on the following object of the class Cube.

```java
Cube c(1, 1, 1);
```

While no layers are activated, the following standard command invokes the version of the main layer of modify() that modifies and returns only the length of the cube c.

```java
System.out.println(c);
=> "Length: 4"
```

However, the following example activates Second_dim layer (via with). In this case, the printing command invokes first the version of modify() included in Second_dim and then invokes the version of the main layer of modify().

```java
with Second_dim (System.out.println(c));
=> "Length: 4; width: 5;"
```

Another example is the following:

```java
without Second_dim (with Second_dim (System.out.println(c));
=> "Length: 4;"
```

**Contributions**

Contributions of this paper are the following:

1. A precise operational semantics for a rich model of context-oriented programming languages.
2. A static type system that is mathematically sound for context-oriented programming languages.

**Organization**

The organization of the rest of the paper is as follows. Section 2 presents the language model and the operational semantics of the language. The type system together with its mathematical soundness proof is presented in Section 3. Related and future work is discussed in Section 4.

**2. Syntax and Operational Semantics**

This section presents the model of our programming language together with an operational semantics for the language. Most basic object-oriented aspects as subtyping and inheritance are included in the language (dubbed J-COP) that we use in this paper. For the sake of readability, we followed the Java syntax for corresponding constructs. The syntax of J-COP is shown in Figure 2.

`Bool` and `int` are our primitive types. We assume that `C` is a set of class names with typical element `C`. The set of types (Types) includes `bool`, `int`, and `C`. Moreover "Types" has reference and function types. We let `τ` be a typical element of the set of types. We let `LVar` denotes the set of local variables. Local variables are contained in procedures and are active as long as their hosting procedures are active. Local variables also serve as parameters for procedures. The set of instance variables of a class `C` is denoted by `Var_C`. The internal state of a class is stored via its instance variables. Typical elements of `IVar` and `IVar_C` are `o` and `v`, respectively. The sets of procedure and layer names are denoted by `FunNames` (typical element is `f`) and `LayerNames` (typical element is `l`), respectively. A layer expression is a sequence of layer activation/deactivation. A typical element of the set of layer expressions, denoted by `LayerExpr`, is denoted by `le`.

A program in J-COP consists of a set of classes and a main procedure triggering the program execution. A class contains definitions for a set of procedures and a set of layers each of which contains the definition of a procedure. A parameter, a statement, and an expression are the components of a procedure where the expression denotes the value returned by the procedure.

We use a state representation and a subtype relation to define an operational semantics for the language J-COP. We let `τ_1 ≤ τ_2` denotes that `τ_1` is a subtype of `τ_2`. The class definitions of a given program are used to build the relation `≤` which is introduced in Definition 1.

**Definition 1**

1. `Types = {bool, int, C, ref τ, τ_1 → τ_2}`.
2. A class `C` is a subclass of a class `D` (denoted by `C < D`) if `C` inherits `D` by definition of `C`. The relation `≤_C` on the set of classes is the reflexive transitive closure of `<`. A class `D` is a superclass of `C`, if `C` is a subclass of `D`.

http://www.lifesciencesite.com 2516  lifesciencej@gmail.com
3. The order \( \leq \) on the set of types is defined as:
\[
\leq_{\mathcal{C}}(\tau \leq \tau \mid \tau \in \{\text{int}, \text{bool}, \text{ref}, \tau_1 \to \tau_2\})
\]

\[\tau \in \text{Types} ::= \text{int} | \text{bool} | \text{C} | \text{ref} | \tau_1 \to \tau_2\]
\[e \in \text{Exprs} ::= n | (\text{C})e | \text{this} | o | e . e' | e_1 i_{\text{op}} e_2\]
\[b \in \text{LayerExprs} ::= \text{true} | \text{false} | e_1 c_{\text{op}} e_2 | b_1 b_{\text{op}} b_2\]
\[le \in \text{Stmts} ::= e_1 . v := e_2 | o_1 := le o_2 . f(e) | o_1 := o_2 . f(e) | o_1 := \text{super} . f(e) | o_1 := \text{proceed} o_2 . f(e) | o_1 := \text{new} C | S_1; S_2 | \text{if} b \text{ then } S_1 \text{ else } S_2 | \text{while } b \text{ do } S_1\]
\[\text{fun} \in \text{Funs} ::= f(p)(\{S; \text{return}(e);\})\]
\[\text{layer} \in \text{Layers} ::= \text{Layer} \{\text{fun}\}\]
\[\text{inhrt} \in \text{Inherits} ::= e | \text{inherits } C\]
\[\text{class} \in \text{Classes}: = \text{class } C \text{ inhrt } \{\text{fun}* \text{layer}^*\}\]
\[\text{prog} \in \text{Progs} ::= \text{class}'\text{main}(\{S\})\]

**Figure 2**: The programming language J-COP

Definition 2 introduces necessary components towards introducing the states of the operational semantics. The symbol \( \mathcal{A} \) denotes an infinite set of memory addresses with \( a \) as a typical element of \( \mathcal{A} \).

**Definition 2**

1. For a class \( C \), \( IVa\)r\( _C \) and \( \text{Fun}_C \)-denote the set of instance variables and the set of functions of \( C \), respectively. The set of layer names of a class \( C \) is denoted by \( \text{Layer}_C \).
2. \( \emptyset = \mathbb{Z} \cup \mathcal{A} \cup \{L\} \).
3. \( \text{Stacks} = \{s \mid s : \text{LVar} \to \emptyset\} \).
4. \( \text{ObjectContents} = \{I_{(C,n)} \mid I_{(C,n)} : \text{IVar}_C \to \emptyset, C \in \mathcal{C}, n \in \mathbb{N}\} \).
5. \( \text{Heaps} = \{h \mid h : \mathcal{A} \to \{I_{(C,n)} \mid C \in \mathcal{C}, n \in \mathbb{N}\}\} \).
6. \( \text{States} = \{(s, h, L^s) \mid s \in \text{stacks}, h \in \text{Heaps}, L^s \subseteq \text{LayerNames}\} \).

Model values are elements of the set \( \emptyset \). A semantic state is a triple of a stack, a heap, and a set of layer names that are active at that program point (state). The set of local variables includes the special variable \( \text{this} \) which points to the current active object. For an address \( a \in \text{dom}(h) \), \( h_a(\alpha) \) denotes the \( i \)-th component of the triple \( h(\alpha) \), where \( i = 1, 2, 3 \).

Definition 3 introduces the notations \( F_C \) and \( L_C \). For a class \( C \), \( F_C \) maps each procedure name in \( C \) to the triple consisting of the parameter variable of the procedure, procedure body, and returned expression of procedure. For a class \( C \), \( L_C \) maps each layer name in \( C \) to the components of its procedure.

**Definition 3**

1. \( \text{FunBodies} = \{F_C | F_C : \text{Fun}_C \to \text{LVar} \times \text{Stmt} \times \text{Expr}; f \mapsto (p_f, S_f, e_f)\} \).
2. \( \text{Layers} = \{L_C | L_C : \text{Layer}_C \to \text{Fun}_C \times \text{LVar} \times \text{Stmt} \times \text{Expr}; f \mapsto (I, p_f, S_f, e_f)\} \).

Figure 3 presents inference rules of four procedures that are used in the inference rules of the operational semantics.

For a given list of layer names \( Ls \) and a layer expression \( le \), Figure 3 presents the procedure layer which adds the layers activated by \( le \) to \( Ls \) and removes the layers deactivated by \( le \) from \( Ls \). The definition of the class procedure is presented in Figure 3. This procedure finds whether a given variable belongs to a given class or to any of its ancestor classes. The procedure super, which for a function name and a class name searches for the first ancestor of the class that contains a definition for the function, is outlined in the same figure which as well presents the definition of the procedure clslyrs. This procedure determines which members of a given list of active layers (\( L \)) contain a definition for a given procedure, \( f \).

The semantics of the J-COP expressions is presented in Figure 4. Some comments on the figure are in order. The variable \( v \) of the class pointed-to by \( e \) is denoted by \( e.v \). We assume that the set of variables in a class does not intersect with the set of the variables of any of the class’s ancestors. We also assume that for a class \( C \), the domain of \( \text{IVar}_C \) includes all the variables of \( C \) and its ancestors. Hence the rule \( \text{inst}^s_1 \) ensures that \( v \) is a member of the class pointed-to by \( e \) or is a member of any of the class’s ancestors (via calling the class procedure). The semantic of \( e \) is the address of the triple in memory representing the meant class object. The third component of this triple is denoted by \( I \) (which is a map representing the values of the object’s variables). The rule \( \text{cast}^s_1 \) says that the cast of the expression \( e \) in the form of a class \( C \) aborts only if \( e \) points to a triple in the memory that represents a class \( D \) that is not a descendant of \( C \).

Definition 4 formalizes the case when a statement aborts execution.

**Definition 4**

A statement \( S \) aborts at a state \( (s, h, L^s) \), denoted by \( S : (s, h, L^s) \rightarrow \text{abort} \), if it not possible (provided that \( S \) is not stuck in an infinite loop) to find a state \( (s', h', L'^s) \) such that \( S : (s, h, L^s) \rightarrow (s', h', L'^s) \) according to inference rules of Figure 5.

The semantics of the statements of the J-COP language is shown in Figure 5. Some comments on the rules are as follows. The rule \( (=^s_2) \) modifies the variable \( v \) of the object referenced by \( e_1 \). This is done via updating the third component of \( h(\{e_1\}(s, h)) \) and keeping the first two components \( h_2(\{e_1\}(s, h)) \) and
The preconditional of the rule \((C,f')\) requires that the body \(S\) of the procedure \(f\) to be well formed. The preconditional also requires the existence of a common type that covers any overloading for \(f\). The first part of the preconditional of the rule \((\equiv_{\text{lo},f}^l)\) requires that all procedures named \(f\) inside layers of the class \(C\) to have an upper bound type. Among others requirements, the preconditional of this rule also ensures that the set \(L^1\) is in line with the expression \(\leq (\text{layer}(le,L^1) = L^1')\). The rule \((\text{pro}^l)\) uses the rule \((\equiv_{\text{lo},f}^l)\) to determine types for all instances of \(f\) in layers of the class \(C\). In line with expectation of the rules for non-atomic statements like \((\text{if}^l)\), \((\text{while}^l)\), and \((\text{ser}^l)\), these rules require their sub-statements to be well formed.

Definition 6 presents the condition when a state respects a context denoted by \((s,h,L^1) \sim (\Gamma,L^1)\).

Definition 6.
1. \((s,h,L^1) \sim (\Gamma,L^1) \equiv \text{def} (a)\ L^1 \subseteq L^1', (b)\forall o \in \text{dom}(\Gamma).\Gamma(o) = \text{int} \Rightarrow s(a) \in \mathbb{Z}, \ (c)\forall o \in \text{dom}(\Gamma).\Gamma(o) = \text{ref} C \Rightarrow h(s(o)) = (C,n, l_{(C,n)}), \text{and } (d)\forall a \in \mathcal{A}.a \in \text{dom}(h) \Rightarrow h_3(a) \sim (s,h)\Gamma.

(Definition 6.2)
2. \(l_{(C,n)} \sim (s,h)\Gamma \equiv \text{def} \forall D.\text{if } C \subseteq D, \text{then } (a)\Gamma(D,v) = \text{int} \Rightarrow l_{(E,m)}(v) \in \mathbb{Z}, \text{and } (b)\Gamma(D,v) = \text{ref} E \Rightarrow h(l_{(C,n)}(v)) = (E,m,l_{(E,m)}) \text{ and } l_{(E,m)} \sim (s,h)\Gamma.

Now we prove the soundness of the type system.

Lemma 1
Typed expressions of the language \(J\text{-COP}\) do not abort (go wrong). Moreover:

(a) If \(\Gamma \vdash e : \text{int} \text{ and } (s,h,L^1) \sim (\Gamma,L^1)\), then \([e](s,h) \in \mathbb{Z} .
(b) If \(\Gamma \vdash e : \text{ref} C \text{ and } (s,h,L^1) \sim (\Gamma,L^1)\), then \(h_1([e](s,h)) = D \text{ and } D \subseteq C .

Proof
Suppose that \(e\) is an expression of the language \(J\text{-COP}\) such that \(\Gamma \vdash e : \tau \text{ and } (s,h,L^1) \sim (\Gamma,L^1)\). We show that \([e](s,h) \not\in \bot\) and we show \((a)\) and \((b)\) above. This is shown by induction on \(\Gamma \vdash e : \tau\) with case analysis on the last inference rule applied. Main cases are only shown below:

Case (\(\alpha'\)):
In this case \(\Gamma(o) = \tau\). We have two subcases.
In the first sub-case \(\Gamma(o) = \text{int}\) which implies \(s(o) \in \mathbb{Z}\) because \((s,h,L^1) \sim (\Gamma,L^1)\). In the second sub-case \(\Gamma(o) = \text{ref} C\) which implies \(s(o) \in \text{dom}(h)\) because \((s,h,L^1) \sim (\Gamma,L^1)\). Hence in both subcases \([e](s,h) \not\in \bot\) and clearly \((a)\) and \((b)\) are satisfied.

Case (\(\text{cast}_1\)):
In this case \(e = (C)e', \Gamma \vdash e' : \text{int}, \Gamma \vdash\)
We also have \( L[s] = \emptyset \). Hence by induction hypothesis, \( h_1([e'](s,h)) = E \) and \( E \leq C \). Hence \( E \leq D \) because \( C \leq D \). We also have \( l = h_2([e'](s,h)) \) and \( v \in dom(l) \) because \( class(C,v) = D \). Hence by \( \text{inst}_1 \), \([e'].v\](s,h) = \( I(v) \neq \bot\). Now \( l \sim_{(s,h)} \Gamma \) because \( (s,h,L') \sim (\Gamma,L') \). Hence \( \Gamma'(D,v) = \text{int} \Rightarrow I(v) \in \Gamma \) and \( \Gamma'(D,v) = \text{ref} E \Rightarrow h(I(v)) = (E,m,l_{(e,m)}) \) and \( l_{(e,m)} \sim_{(s,h)} \Gamma \). This completes the proof for this case.

**Lemma 2**

Typed Boolean expressions of the language J-COP do not abort (go wrong).

![Figure 3. Inference rules of necessary functions for semantics](attachment:image.png)

![Figure 4. Semantics of J-COP expressions](attachment:image.png)
\[
\text{class}(h_1([e_1](s,h)), v) = D \\
l = h_3([e_3](s,h)) \\
l' = l[v \mapsto [e_2](s,h)] \\
\]

\[
F_{h_2}([e_2](s,h))(f) = (p_f, S_f, e_f) \\
S_f(s) \iff S_f([e_2], p_f \mapsto [e](s,h), h, L') \rightarrow (s', h', L'') \\
o_1 = o_2.f(e)(s, h, L') \rightarrow (s', [o_1 \mapsto [e_f](s', h'), h', L'']) \\
---
\[
l_1 \ldots l_m \subseteq L' \text{ such that } \forall 1 \leq i \leq m \left( L_{h_1([o_2], (s,h))(l_i)} = (f, S_i, e_i, p_i) \right) \\
S_i(s) \iff S_i([e_2], p_i \mapsto [e](s_i, h_i), h_i, L_i') \rightarrow (s_2, h_2, L_2') \\
o_1 = \text{le } o_2.f(e)(s, h, L') \rightarrow (s_{m+1}, [o_1 \mapsto [e_f](s', h'), h', L'']) \\
---
\[
E = \text{is the direct superclass of } h_1(s(this)) \\
super(E, f) = D \\
F_p(f) = (p_f, S_f, e_f) \\
S_f(s) \iff S_f([e](s, h), h, L') \rightarrow (s', h', L'') \\
o_1 = \text{super.f(e)(s, h, L') \rightarrow (s', [o_1 \mapsto [e_f](s', h'), h', L''])} \\
as \in A \setminus \text{dom(h)} \quad \text{n is fresh} \\
o = \text{new C}(s, h, L') \rightarrow (s[o \mapsto a], h[a \mapsto (C, n, \{(v, l) \mid v \in IVar_C\}], L') \\
\]

\[
s_i, h_i, L_i = (s, h, L') \\
\text{clsInv}(h_1(s(o_2), f, L')) = [l_1 \ldots l_m] \\
S_i(s) \iff S_i([e](s_i, h_i), h_i, L_i') \rightarrow (s_2, h_2, L_2') \\
S_i(s) \iff S_i([e](s_i, h_i), h_i, L_i') \rightarrow (s_{i+1}, h_{i+1}, L_{i+1}') \\
o_1 = \text{proceed o_2.f(e)(s, h, L') \rightarrow (s_{m+1}, [o_1 \mapsto [e_m](s_{m+1}, h_{m+1})), h_{m+1}, L_{m+1}')} \\
---
\[
\forall b \ldots [b](s, h) = \text{true } \land S_f(s, h, L') \rightarrow (s', h', L'') \\
\text{if } b \text{ then } S_f(s, h, L') \rightarrow (s', h', L'') \\
\text{while } b \text{ do } S_f(s, h, L') \rightarrow (s', h', L'') \\
\forall b \ldots [b](s, h) = \text{false } \land S_f(s, h, L') \rightarrow (s', h', L'') \\
\text{while } b \text{ do } S_f(s, h, L') \rightarrow (s', h', L'') \\
\forall b \ldots [b](s, h) = \text{true } \land S_f(s, h, L') \rightarrow (s', h', L'') \\
\text{while } b \text{ do } S_f(s, h, L') \rightarrow (s', h', L'') \\
---
\]

**Figure 5.** Inference rules of the operational semantics for J-COP constructs

**Theorem 1**
Well-formed statements of the language J-COP do not abort (go wrong).

**Proof**
Suppose that $S$ is a statement of the J-COP language. Suppose that the maps $F_C$ and $L_C$ and the relation $\leq$ describing the classes used in $S$ are given along with $S$. Suppose also that $(\Gamma', L') \models S: \text{WF}$ and $(s, h, L') \sim (\Gamma', L')$. We show that if $S$ does not contain infinite loop then $\neg(S; (s, h, L') \rightarrow \text{abort})$, i.e.

\[
\exists \Gamma' \models S: \text{WF} \text{ with case analysis on the last type rule applied.} \\
\text{Outlines of main cases are shown below.} \\
\text{Case } (\Rightarrow) \\
\]

http://www.lifesciencesite.com 2520 lifesciencej@gmail.com
In this case:

a. $S = e_1, v := e_2$.

b. $\Gamma \vdash e_1, v : \tau_1, \Gamma \vdash e_2 : \tau_2$, and $\tau_2 \leq \tau_1$.

c. By the rule $(e, v')$, $\Gamma \vdash e_1, v : \tau_1 \Rightarrow \Gamma \vdash e_1 : \text{ref } C$, $\text{class}(C, v) = E$, and $\Gamma ((E, v)) = \tau_1$.

By Lemma 1, $[e_1] (s, h) \in \text{dom}(h)$ and $h_1 ([e_1] (s, h)) \leq C$. Hence there is a class $D$ such that

$\text{class}(h_1 ([e_1] (s, h)), v) = D$ because there is a class $E$ such that $\text{class}(C, v) = E$. Also by Lemma 1, $[e_2] (s, h) \neq \perp$. Hence the state $(s', h', l') = (s, h, [e_1] (s, h)) \Rightarrow (h_1 ([e_1] (s, h)), h_2 ([e_1] (s, h)), l)$, $L'$ is defined and the statement does not abort. Clearly

$(s, h, l') \sim (\Gamma', L')$.

Figure 6. Type system for J-COP constructs.
a. $S = \alpha_1 \Rightarrow \alpha_2.f(e)$.
b. $\Gamma \vdash \alpha_2 : \text{ref} \ C, \Gamma \vdash \alpha_1 : \tau_2, \Gamma \vdash e : \tau_1, \tau_1 \leq \tau_1$, and $\tau_2 \leq \tau_2$.
c. $(f) = (p, S, e')$ and $\Gamma \vdash p : \tau_1$.
d. $(\Gamma[p \rightarrow \tau_1, this \rightarrow \text{ref} \ C], \Gamma'') = S:WF$ and $\Gamma[p \rightarrow \tau_1, this \rightarrow \text{ref} \ C] = e'.\tau_2$.
e. $\forall D, (C \ll D)$ it is true that:

$$((\Gamma', \Gamma') = (D, D): \tau_1 \rightarrow \tau_2) \Rightarrow (\tau_1 = \tau_1$ and $\tau_2 \leq \tau_2$$.

By Lemma 1, $[\alpha_2]\langle s, h \rangle \in \text{dom}(h)$ and $h([\alpha_2]\langle s, h \rangle) \leq C$. Since $F_c(f) = (p, S, e')$, $F_{\text{ref}}[\alpha_2]\langle s, h \rangle(f) = (p, S, e')$. Now we notice that

$$\langle s \text{this} \rightarrow s[\alpha_2], p \rightarrow [e] \langle s, h \rangle, h, \tau' \rangle \sim (\Gamma[p \rightarrow \tau_1, this \rightarrow \text{ref} \ C], \Gamma'')$$.

This is so because

1. $\mathcal{h}(s[\alpha_2]) = h(s[\alpha_2]) = (C_n, \iota(C_n))$ because $\Gamma \vdash \alpha_2 : \text{ref} \ C$, and
2. $\mathcal{s}(p) = \mathcal{s}([e] \langle s, h \rangle)$ and $\Gamma \vdash e : \tau_1 \leq \tau_1$.

Therefore by induction hypothesis there is a state $(s', h', \tau')$ such that $(s', h', \tau') \sim (\Gamma[p \rightarrow \tau_1, this \rightarrow \text{ref} \ C], \Gamma')$ and

$$\mathcal{s} : (s \text{this} \rightarrow s[\alpha_2], p \rightarrow [e] \langle s, h \rangle, h, \tau') \rightarrow (s', h', \tau')$$

because

$$(\Gamma[p \rightarrow \tau_1, this \rightarrow \text{ref} \ C], \Gamma') = S:WF.$$ This implies $[e] \langle s', h' \rangle \neq \bot$ because

$$\Gamma[p \rightarrow \tau_1, this \rightarrow \text{ref} \ C] \equiv e'.\tau_2.$$ Hence the state $(s[\alpha_1] \rightarrow [e] \langle s', h' \rangle, h, \tau')$ is defined and satisfies $(s[\alpha_1] \rightarrow [e] \langle s', h' \rangle, h, \tau') \sim (\Gamma', \Gamma')$. This completes the proof of this case.

Case ($=^\iota$)

In this case

a. $S = \alpha_1 \Rightarrow \text{le} \alpha_2.f(e)$.
b. layer(le, $L^c$) = $L^c$ and $(s_1, h_1, L^c_1) = (s, h, L)$.
c. $[l_1 \ldots l_m] \subseteq L^c_1$ such that

$$\forall 1 \leq i \leq m \{l_1 \ldots l_m \text{\langle s_1, h_1 \rangle}\}(l) = (f, p_i, S_1, e_i)$$.

d. $\exists \tau_1, \tau_2$ such that for all $l \in L'$ if $L_c(l) = (f, p_i, S_1, e_i)$, then

$$\Gamma \vdash p_i : \tau_1, \Gamma[p \rightarrow \tau_1, this \rightarrow \text{ref} \ C] = S_i:WF,$$

and $\Gamma[p \rightarrow \tau_1, this \rightarrow \text{ref} \ C] \equiv e_i.\tau_1$.
e. $\Gamma \vdash \alpha_2 : \text{ref} \ C, \Gamma \vdash e : \tau_1, \Gamma \vdash p_i : \tau_2, \tau_1 \leq \tau_1$, and $\tau_2 \leq \tau_2'$.

By Lemma 1, $[\alpha_2]\langle s, h \rangle \in \text{dom}(h)$ and

$$h([\alpha_2]\langle s, h \rangle) \leq C.$$ If the list $[l_1 \ldots l_m]$ is empty then the statement $S$ does not abort and $(s', h', \tau') = (s, h, L')$. For $l_1$, we have $L_{h([\alpha_2]\langle s, h \rangle)}(l_1) = (f, p_i, S_1, e_1)$. Similarly to the previous case, we conclude:

$$(s_1[\text{this} \rightarrow s_1[\alpha_2], p_1 \rightarrow [e] \langle s_1, h_1 \rangle, h_1, L_1'] \sim (\Gamma[p_1 \rightarrow \tau_1, this \rightarrow \text{ref} \ C], \Gamma')$$.

Then by induction hypothesis there exists a state $(s_2, h_2, L_2')$ such that $(s_2, h_2, L_2') \sim (\Gamma', \Gamma')$ and

$$S_1 : (s_1[\text{this} \rightarrow s_1[\alpha_2], p_1 \rightarrow [e] \langle s_1, h_1 \rangle, h_1, L_1') \rightarrow (s_2, h_2, L_2')$$.

Now clearly,

$$\langle s_2[\alpha_1] \rightarrow [e] \langle s_2, h_2 \rangle, h_2, L_2' \rangle \sim (\Gamma[p_1 \rightarrow \tau_1, this \rightarrow \text{ref} \ C], \Gamma')$$.

Then by induction hypothesis there exists a state $(s_3, h_3, L_3')$ such that $(s_3, h_3, L_3') \sim (\Gamma', \Gamma')$ and

$$S_2 : (s_2[\alpha_1] \rightarrow [e] \langle s_2, h_2 \rangle, h_2, L_2') \rightarrow (s_3, h_3, L_3')$$.

Therefore a simple induction on $m$ can prove that for all $i$ there exists state $(s_{i+1}, h_{i+1}, L_{i+1}'$) such that

$$(s_{i+1}, h_{i+1}, L_{i+1}') \sim (\Gamma', \Gamma')$$

and

$$\forall i : (s_2[\alpha_1] \rightarrow [e] \langle s_2, h_2 \rangle, h_2, L_2) \rightarrow (s_{i+1}, h_{i+1}, L_{i+1}')$$.

Hence the state $(s_{n+1}, h_{n+1}, L_{n+1}')$ is defined and satisfies $(s_{n+1}, h_{n+1}, L_{n+1}') \sim (\Gamma', \Gamma')$. Now by Lemma 1, $[e_1] \langle s_{n+1}, h_{n+1} \rangle \neq \bot$ and hence

$$(s_{n+1}[\alpha_1] \rightarrow [e_1] \langle s_{n+1}, h_{n+1} \rangle, h_{n+1}, L_{n+1}')$$

is defined and satisfies

$$(s_{n+1}[\alpha_1] \rightarrow [e_1] \langle s_{n+1}, h_{n+1} \rangle, h_{n+1}, L_{n+1}') \sim (\Gamma', \Gamma')$$

which completes the proof of this case.

Case ($\supset$):

This case is similar to the case of ($=^\iota$).

Case ($\propto$):

This case is similar to the case of ($=^\iota$).

4. Discussions

**Related work:** an operational semantics and a type system for modeling and checking context-oriented constructs are presented in (Hirschfeld, Igarashi & Masursha, 2011). While the language model studied in (Hirschfeld et al., 2011) is functional, our model is structural. The type system presented in the current paper and that in (Hirschfeld et al., 2011) stop the command proceed from executing faulty procedures.

An operational semantics, that is based on delegation based calculus, is presented in (Schippers, Janssens, Haupt & Hirschfeld, 2008) for the language $c$, a context-oriented programming language. The research in (Clarke & Sergey, 2009) presents a syntax-based semantics for COP concepts as implemented by ContextL, ContextI*, and other examples. This paper also introduces a type system that prevents program from getting stuck. The semantics presented of most related work uses general calculi to represent context-dependent behavior of COP programs. Our semantics, on the other hand, is built directly on an accurate memory model which adds to the clarity and soundness of our semantics and type system.

Aiming at describing behavioral variations, delta modules and layers are used by delta-oriented programming (DOP) (Schaefer, Bettini, Bono,
Damiani & Tanzarella, 2010) and feature-oriented programming (FOP) (Batory, Sarvela & Rauschmayer, 2004), respectively. In these manners, the static composition of classes with layers creates many similar software artifacts. Transitional semantics was presented for DOP in (Schaefer, Bettini & Damiani, 2011) and for FOP in (Delaware, Cook & Batory, 2009). Noticeably, in these approaches new procedures can be added by layers. This fact sophisticates any accurate semantics and type system for DOP and FOP.

Future work: it is intersecting to extend the language of the current paper to allow layer inheritance and layer dependency. This enables one layer to require the presence of another layer. It also enables expressing the condition that two layers cannot be active simultaneously. Another direction for a future work is to extend the language to associate candidate procedures of the command proceed() with priorities for execution.

Acknowledgements:
Foundation item: Al Imam University (IMSIU) Project (No.: 330917). Authors are grateful to Al Imam University (IMSIU), KSA for financial support to carry out this work.

Corresponding Author:
Dr. Mohamed A. El-Zawawy
College of Computer and Information Sciences, Al Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh, Kingdom of Saudi Arabia.
E-mail: maelzawawy@cu.edu.eg

References
[1] Batory, D. S., Sarvela, J. N. & Rauschmayer, A., ‘Scaling step-wise refinement’, IEEE Tran., Software En, 2004; 30(6), 355–371.
[2] Campione, M., Walrath, K. & Huml, A., The Java Tutorial: A Short Course on the Basics, 3rd edn, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 2000.
[3] Clarke, D. & Sergey, I., ‘Semantics for context-oriented programming with layers, in ‘International Workshop on Context-Oriented Programming’, COP ’09, ACM, New York, NY, USA, 2009, pp. 10:1–10:6.
[4] Costanza, P., Herzeel, C. & D’Hondt, T., Context-oriented software transactional memory in common lisp, in J. Noble, ed., ‘DLS’, ACM, 2009, pp. 59–68.
[5] Delaware, B., Cook, W. R. & Batory, D. S., A machine-checked model of safe composition, in M. S. ’udholt, ed., ‘FOAL’, ACM, 2009, pp. 31–35.
[6] Flanagan, D., JavaScript - Pocket Reference: Activate Your Web Pages, Third Edition, O’Reilly.
[7] Golubski, W. & Lippe, W.-M. (1995), ‘A complete semantics for smalltalk-80’, Comput. Lang. 2012; 21(2), 67–79.
[8] Hirschfeld, R., Costanza, P. & Nierstrasz, O., ‘Context-oriented programming’, Journal of Object Technology, 2008; 7(3), 125–151.
[9] Hirschfeld, R., Igarashi, A. & Masuhara, H., Contextfj: a minimal core calculus for context-oriented programming, in H. Rajan, ed., ‘FOAL’, ACM, 2011, pp. 19–23.
[10] Schaefer, I., Bettini, L., Bono, V., Damiani, F. & Tanzarella, N., Delta-oriented programming of software product lines, in J. Bosch & J. Lee, eds, ‘SPLC’, Vol. 6287 of Lecture Notes in Computer Science, Springer, 2010, pp. 77–91.
[11] Schippers, H., Janssen, D., Haupt, M. & Hirschfeld, R., Delegation-based semantics for modularizing crosscutting concerns, in G. E. Harris, ed., ‘OOPSLA’, ACM, 2008, pp. 525–542.