A Semantics for Context-Oriented Programming with Layers

Dave Clarke and Ilya Sergey

Katholieke Universiteit Leuven
{Dave.Clarke, Ilya.Sergey}@cs.kuleuven.be

International workshop on Context-Oriented Programming at ECOOP’09
7 July 2009
Why yet another semantics for COP?
Context-oriented programming (COP) is a programming approach whereby the context in which expressions evaluate can be adapted as a program runs.
COP key features

- Context-dependent evaluation
- Explicit context
- Context manipulation
COP key features

- Context-dependent evaluation
- Explicit context
- Context manipulation
COP key features

- Context-dependent evaluation
- Explicit context
- Context manipulation
COP is implicitly (or explicitly) assumed in...

- Groovy
- LISP/Clojure
- Ruby
- Objective C
- ContextJ, ContextL
- Lasagne
- Ambience
- CaesarJ
Example: *with, without and proceed in action*

```java
class Actor {
    void act() {
        ...
        without(Logging) { stealth(); }
    }
}

layer Logging {
    class Actor {
        void act() {
            proceed();
            println("Acted");
        }
    }
}

with (Logging) { (new Actor()).act(); }
```
What is COP
Problem description
COP formalization
Possible pitfalls

Problems to solve

- What is the order of expression evaluation for COP language?
  - There is a big step semantics (Schippers et al.)

- How to ensure that all method invocations are resolved at runtime?

- Are statically-defined methods overridden correctly at runtime?
Problems to solve

- What is the order of expression evaluation for COP language?
  - There is a big step semantics (Schippers et al.)

- How to ensure that all method invocations are resolved at runtime?

- Are statically-defined methods overridden correctly at runtime?
Problems to solve

- What is the order of expression evaluation for COP language?
  - There is a big step semantics (Schippers et al.)

- How to ensure that all method invocations are resolved at runtime?

- Are statically-defined methods overridden correctly at runtime?
Problems to solve

- What is the order of expression evaluation for COP language?
  - There is a big step semantics (Schippers et al.)

- How to ensure that all method invocations are resolved at runtime?

- Are statically-defined methods overridden correctly at runtime?
Problems to solve

- What is the order of expression evaluation for COP language?
  - There is a big step semantics (Schippers et al.)

- How to ensure that all method invocations are resolved at runtime?

- Are statically-defined methods overridden correctly at runtime?
We need an **operational semantics** with the **sound type system**!
ContextFJ is a language to describe core features of the Context-Oriented programming

- Based on Featherweight Java
- Has layers as dedicated language constructs
- Includes proceed, with and without statements
- Has no inheritance and subtyping
### ContextFJ syntax: Terms and Contexts

#### Terms

\[
t ::= x \mid t.f \mid t.m_L(t) \mid \text{new } C(t) \\
    \mid \text{with}(l)t \mid \text{without}(l)t \mid \text{proceed}(t)
\]

#### Values

\[
v ::= \text{new } C(v)
\]

#### Evaluation Contexts

\[
E[ ] ::= [ ] \mid E[ ].f \mid E[ ].m_L(t) \\
    \mid E[v.m_L(v, [ ], t)] \mid E[\text{new } C(v, [ ], t)] \\
    \mid E[\text{with}(l)[ ]] \mid E[\text{without}(l)[ ]]
\]
ContextFJ syntax: layers and method bindings

Layer definition

\[ \mathcal{L} ::= \text{layer } l \{ B \} \]

Method bindings

\[ B ::= (m, C_0) \mapsto M \]
**Evaluation: bound methods**

**Bound methods**

\[
\begin{align*}
BM_L([ ]) &= \emptyset \\
BM_L(E[[].f]) &= BM_L(E) \\
BM_L(E[[].m(\bar{t})]) &= BM_L(E) \\
BM_L(E[v.m(\bar{v}, [ ], \bar{t})]) &= BM_L(E) \\
BM_L(E[new \ C(\bar{v}, [ ], \bar{t})]) &= BM_L(E) \\
BM_L(E[with(l)[ ]]) &= \begin{cases} 
BM_L(E), & \text{if } l \in L \\
BM_L(E) \cup \text{dom}(l), & \text{otherwise}
\end{cases} \\
BM_L(E[without(l)[ ]]) &= BM_L \cup \{l\}(E)
\end{align*}
\]
Evaluation: excluded layers

Excluded layers

\[ XL([ ]) = \emptyset \]
\[
\begin{align*}
XL(E[[].f])
\newline
XL(E[[].m(\bar{t})])
\newline
XL(E[v.m(\bar{v}, [ ], \bar{t})])
\newline
XL(E[\text{new } C(\bar{v}, [ ], \bar{t})])
\newline
XL(E[\text{with}(l)[ ]])
\end{align*}
\]

\[
\begin{align*}
XL(E[\text{without}(l)[ ]])
\end{align*}
\] = \{l\} \cup XL(E)
Reduction rules (1)

\[(E\text{-}\text{WITH})\]

\[E[\text{with}(l)v] \rightarrow E[v]\]

\[(E\text{-}\text{WITHOUT})\]

\[E[\text{without}(l)v] \rightarrow E[v]\]
Reduction rules (2)

(E-INVK_LAYER)

\[
\text{Imbody}(l, m, C') = (\bar{x}, t) \\
(m, C) \notin BM_L(E') \quad l \notin XL(E') \\
E[\text{with}(l)E'[(\text{new } C(\bar{v})).m_L(\bar{u})]] \rightarrow \\
E[\text{with}(l)E'[\{\bar{x} \mapsto \bar{u}, \text{proceed} \mapsto \text{this}.m_{L \cup \{l\}}, \text{this} \mapsto \text{new } C(\bar{v})}\} t]]
\]

(E-INVK_CLASS)

\[
(m, C) \notin BM_L(E) \quad \text{mbody}(m, C) = (\bar{x}, t) \\
E[\text{new } C(\bar{v})].m_L(\bar{u})] \rightarrow E[\{\bar{x} \mapsto \bar{u}, \text{this} \mapsto \text{new } C(\bar{v})\} t]
\]
Example: evaluation

layer $l_1$ {class $C$ { $D$ $m()$ {return proceed();} }}

layer $l_2$ {class $D$ { $C$ $n()$ {return new $C()$;}}} 

class $C$ {$D$ $m()$ {return new $D()$;}}

class $D$ {}

with($l_1$){with($l_2$){ new $C().m().n()$}}

→ with($l_1$){with($l_2$){ new C().$m_{l_1}()$.n()}}

→ with($l_1$){with($l_2$){ new $D().n()$ }}

→ with($l_1$){ with($l_2$){new $C()$}}

→ with($l_1$){new $C()$}

→ new $C()$
Some methods need other undefined methods of specific types to be evaluated - *requirements*

- Before invoke method we should satisfy its requirements
- Layers provide new methods and require some other ones
Some methods need other undefined methods of specific types to be evaluated - *requirements*

Before invoke method we should satisfy its requirements

Layers provide new methods and require some other ones
Type system outline

- Some methods need other undefined methods of specific types to be evaluated - *requirements*
- Before invoke method we should satisfy its requirements
- Layers provide new methods and require some other ones
**ContextFJ syntax again: method definitions**

**Method definition**

\[ M :::= C [\Psi] m(C \overline{x}) \{ \text{return } t; \} \]

**Method requirements**

\[ \Psi :::= \epsilon \mid MT, \Psi \]

**Method types**

\[ MT :::= (m, C_0) \mapsto [\Psi \overline{C}] \rightarrow C \cdot L \]

**Excluded layers**

\[ L :::= \text{a set of layer names} \mid \top \quad (\forall L. L \subseteq \top) \]
Method types demystified

\[(m, C_0) \mapsto [\Psi] \overline{\mathcal{C}} \rightarrow C \circ L\]

- \(m\) is a method’s name; \(C_0\) is a receiver class type; \(\overline{\mathcal{C}}\) are parameter types; and \(C\) is the result type;
- \(L\) is the set of excluded layers
- \(\Psi\) is a set of method requirements
Term typing

- Typing relation
  \[ \Psi; \Gamma \vdash t : C \]

- Term is well-typed.
  *Root* term \( t \) must be typed in the empty environment.
  \[ \exists C : \emptyset; \emptyset \vdash t : C \]
Key idea

- Method invocations add new requirement into the set $\Psi$
- Layer activations $\text{with}(l)$ removes provided methods from $\Psi$
- $\text{proceed}()$ and $\text{without}(l)$ statements modify method types in $\Psi$, excluding new layers
Method invocation typing

Method invocation is well-typed if
- it is defined in some class or in requirements $\Psi$;
- its requirements are satisfied by $\Psi$;
- its set of excluded layers $L$ is \textit{weaker} than a set $L'$ we suppose to exclude for method of this type.
Layer activation typing

\[(T\text{-}With)\]

\[
\begin{align*}
(Ψ, Φ) & ; Γ \vdash t : C \\
layer & l \{\overline{B}\} \\
\|Φ\| & \subseteq \text{provides}(l) \\
\text{requires}(l) & \preceq Ψ \\
∀((m, C_0) & \mapsto \overline{C} \rightarrow D \bullet L \in Φ) \cdot l \notin L & \\
Ψ & ; Γ \vdash \text{with}(l)t : C
\end{align*}
\]

- Using a layer \(l\) by \(\text{with}(l)\) statement allows us to exclude a part \(Φ\) of requirements from the environment

```
(FJ)
```

```
SYNTAX
```

```
DYNAMIC SEMANTICS
```

```
TYPE SYSTEM
```

```
Q&A
```

---

**Ilya Sergey**

**COP with Layers**
Benefits and limitations

- Caught errors
  - Unresolved method calls
  - Illegal method overriding in layers
  - `proceed()` calls without a higher method to proceed to

- System limitations
  - No inheritance
  - No class-based polymorphism
  - Too many annotations are required for analysis
Benefits and limitations

- **Caught errors**
  - Unresolved method calls
  - Illegal method overriding in layers
  - `proceed()` calls without a higher method to proceed to

- **System limitations**
  - No inheritance
  - No class-based polymorphism
  - Too many annotations are required for analysis
Future work

- Semantics for contexts and inheritance

\[ C <: D \]

layer \( l_1 \) \{class \( C \) \{ \[ F \ m(C \ x) \{ \text{return} \ t_1; \} \} \}

layer \( l_2 \) \{class \( D \) \{ \[ F \ m(C \ x) \{ \text{return} \ t_2; \} \} \}

with (l1) {
    with (l2) {
        new \( C() \).m(...);
    }
}
Thanks for your attention!