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?

COP definition

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

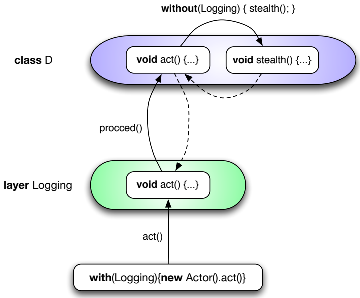
Example: with, without and proceed in action

Enhanced ContextJ

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

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


with (Logging) { (new Actor()).act(); }
```



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**!

What is *ContextFJ*

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

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

Values

$$v \quad ::= \quad \text{new } C(\overline{v})$$

Evaluation Contexts

$$E[\,] \quad ::= \quad [\,] \mid E[[\,].f] \mid E[[\,].m_L(\bar{t})] \mid E[v.m_L(\overline{v}, [\,], \bar{t})] \mid E[\text{new } C(\overline{v}, [\,], \bar{t})] \mid E[\text{with}(l)[\,]] \mid E[\text{without}(l)[\,]]$$

ContextFJ syntax: layers and method bindings

Layer definition

$$\mathcal{L} \quad ::= \quad \text{layer } l \ \{\overline{B}\}$$

Method bindings

$$B \quad ::= \quad (m, C_0) \mapsto M$$

Evaluation: bound methods

Bound methods

$$\mathsf{BM}_L([\,]) = \emptyset$$

$$\left.\begin{array}{l}\mathsf{BM}_L(E[[\,].f]) \\ \mathsf{BM}_L(E[[\,].m(\overline{t})]) \\ \mathsf{BM}_L(E[v.m(\overline{v},[\,],\overline{t})]) \\ \mathsf{BM}_L(E[\mathsf{new}\ C(\overline{v},[\,],\overline{t})])\end{array}\right\} = \mathsf{BM}_L(E)$$

$$\mathsf{BM}_L(E[\mathsf{with}(l)[\,]]) = \begin{cases}\mathsf{BM}_L(E), & \text{if } l \in L \\ \mathsf{BM}_L(E) \cup \mathrm{dom}(l), & \text{otherwise}\end{cases}$$

$$\mathsf{BM}_L(E[\mathsf{without}(l)[\,]]) = \mathsf{BM}_{L\cup\{l\}}(E)$$

Evaluation: excluded layers

Excluded layers

$$\begin{array}{rcl} \mathsf{XL}([\,]) & = & \emptyset \\ \left.\begin{array}{l} \mathsf{XL}(E[[\,].f]) \\ \mathsf{XL}(E[[\,].m(\overline{t})]) \\ \mathsf{XL}(E[v.m(\overline{v},[\,],\overline{t})]) \\ \mathsf{XL}(E[\mathsf{new}\ C(\overline{v},[\,],\overline{t})]) \\ \mathsf{XL}(E[\mathsf{with}(l)[\,]]) \end{array}\right\} & = & \mathsf{XL}(E) \\ \mathsf{XL}(E[\mathsf{without}(l)[\,]]) & = & \{l\} \cup \mathsf{XL}(E) \end{array}$$

Reduction rules (1)

$$\text{(E-WITH)}$$
$$E[\mathsf{with}(l)v] \longrightarrow E[v]$$

$$\text{(E-WITHOUT)}$$
$$E[\mathsf{without}(l)v] \longrightarrow E[v]$$

Reduction rules (2)

$$\frac{\text{lmbody}(l, m, C) = (\overline{x}, t) \qquad (m, C) \notin \text{BM}_L(E') \qquad l \notin \text{XL}(E')}{E[\text{with}(l)E'[(\text{new } C(\overline{v})).m_L(\overline{u})]] \longrightarrow E[\text{with}(l)E'[\ \{\overline{x} \mapsto \overline{u}, \text{proceed} \mapsto \text{this}.m_{L \cup \{l\}}, \text{this} \mapsto \text{new } C(\overline{v})\}\ t]]} \quad (\text{E-InvkLayer})$$

$$\frac{(m, C) \notin \text{BM}_L(E) \qquad \text{mbody}(m, C) = (\overline{x}, t)}{E[\text{new } C(\overline{v})).m_L(\overline{u})] \longrightarrow E[\{\overline{x} \mapsto \overline{u}, \text{this} \mapsto \text{new } C(\overline{v})\}t]} \quad (\text{E-InvkClass})$$

Example: evaluation

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

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

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

$$\text{class } D \ \{\}$$

$$\begin{aligned}
&\text{with}(l_1)\{\text{with}(l_2)\{\text{new } C().m().n()\}\} \\
&\quad \longrightarrow \ \text{with}(l_1)\{\text{with}(l_2)\{\text{new } C().m_{\{l_1\}}().n()\}\} \\
&\quad \longrightarrow \ \text{with}(l_1)\{\text{with}(l_2)\{\text{new } D().n()\}\} \\
&\quad \longrightarrow \ \text{with}(l_1)\{\text{with}(l_2)\{\text{new } C()\}\} \\
&\quad \longrightarrow \ \text{with}(l_1)\{\text{new } C()\} \\
&\quad \longrightarrow \ \text{new } C()
\end{aligned}$$

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

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

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

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

Method requirements

$$\Psi \quad ::= \quad \epsilon \ \mid\ MT, \Psi$$

Method types

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

Excluded layers

$$L \quad ::= \quad \textit{a set of layer names} \ \mid\ \top \quad (\forall L. L \subseteq \top)$$

Method types demystified

$$(m, C_0) \mapsto [\Psi]\overline{C} \to C \bullet L$$

- m is a method's name; C_0 is a receiver class type; $\overline{C}$ are parameter types; and C is the result type;
- L is the *set of excluded layers*
- Ψ 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.

$$\exists C : \emptyset; \emptyset \vdash t : C$$

Expression typing

Key idea

- Method invocations add new requirement into the set Ψ
- Layer activations with(l) removes provided methods from Ψ
- proceed() and without(l) statements modify method types in Ψ, excluding new layers

Method invocation typing

(T-Invk)

$$\frac{\Psi;\Gamma \vdash t_0 : C \qquad \Psi;\Gamma \vdash \bar{t} : \overline{C} \qquad \mathsf{mtype}(m, C, \Psi) = [\Phi]\overline{C} \to D \bullet L' \qquad \Phi \preceq \Psi \qquad L \subseteq L'}{\Psi;\Gamma \vdash t_0.m_L(\bar{t}) : D}$$

Method invocation is well-typed if

- it is defined in some class or in requirements Ψ;
- its requirements are satisfied by Ψ;
- its set of excluded layers L is *weaker* than a set L' we suppose to exclude for method of this type.

Layer activation typing

$$\text{(T-WITH)}\quad \frac{(\Psi, \Phi);\Gamma \vdash t : C \quad \text{layer } l\ \{\overline{B}\} \quad \|\Phi\| \subseteq \text{provides}(l) \quad \text{requires}(l) \preceq \Psi \quad \forall((m, C_0) \mapsto \overline{C} \to D \bullet L \in \Phi) \cdot l \notin L}{\Psi;\Gamma \vdash \text{with}(l)t : C}$$

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

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

$$C <: D$$

$$\textbf{layer } l_1\{\textbf{class } C \ \{ \ F \ m(\overline{C} \ \overline{x})\{\textbf{return } t_1;\} \ \}\}$$

$$\textbf{layer } l_2\{\textbf{class } D \ \{ \ F \ m(\overline{C} \ \overline{x})\{\textbf{return } t_2;\} \ \}\}$$

```
with (l1) {
  with (l2) {
    new C().m(...);
  }
}
```

Which one of $m()$s should we pick?

Q&A

Thanks for your attention!