A Semantics for Context-oriented Programming with Layers

Dave Clarke Ilya Sergey
Katholieke Universiteit Leuven, Heverlee, Belgium
{dave.clarke,ilya.sergey}@cs.kuleuven.be

Abstract

Context-oriented programming (COP) is a new programming approach whereby the context in which expressions evaluate can be adapted as a program runs. COP provides a degree of flexibility beyond object-oriented programming, while arguably retaining more modularity and structure than aspect-oriented programming. Although many languages exploring the context-oriented approach exist, to our knowledge no formal type-sound dynamic semantics of these languages exists. We address this shortcoming by providing a concise syntax-based formal semantics for context-oriented programming *with layers*, as witnessed by ContextL, ContextJ*, and other languages. Our language is based on Featherweight Java extended with layers and scoped layer activation and deactivation. As layers may introduce methods not appearing in classes, we also give a static type system that ensures that no program gets stuck (i.e., there exists a binding for each dispatched method call).

1. Introduction

Context-oriented programming is a new programming approach whereby the context in which expressions evaluate can be adapted as a program runs. It can be characterized as possessing the following facets:

Context-dependent evaluation The interpretation of programming language statements (e.g., method dispatch) depends upon the context in which they are evaluated;

Explicit context The notion of context is an explicit concept in the programming language; and

Context manipulation Context may be explicitly manipulated during the execution of the program.

In the approaches to context-oriented programming based on layers, as witnessed by ContextL, ContextJ*, and other languages [6, 9, 16, 2], context is explicitly represented as named layers. During program evaluation these layers can be activated and deactivated using the $\mathsf{with}(l)$ and $\mathsf{without}(l)$ scoping constructs, respectively.

```
class Actor {
  void act() {
    ...
    without(Logging) { stealth(); }
  }
}
layer Logging {
  class Actor {
    void act() {
      proceed();
      println("Acted");
    }
  }
}
...
with (Logging) { (new Actor()).act(); }
```

Figure 1. Context-oriented Programming

The example in Figure 1 illustrates the ideas. Layer `Logging` alters the behaviour of the `Actor` method `act()`. It prints out a logging message and delegates to the initial `act()` method via the `proceed()` statement. The original `act()` method deactivates the `Logging` layer to avoid logging of the `stealth()` method call.

The paper makes the following contributions:

- A concise. syntax-directed dynamic semantics for a context-oriented programming language. The semantics are presented as an extension to Featherweight Java (without inheritance or subtyping), by adding layers, layer activation, layer deactivation, and layer delegation.
- A static type system which ensures that a well-typed binding exists for every dispatched method.

2. ContextFJ

ContextFJ (Figure 2) extends Featherweight Java [7] without inheritance with constructs for expressing layer-based context-oriented programming.

COP'09 July 7, 2009, Genova, Italy

Class Definition
$\mathcal{C} \quad ::= \quad \mathsf{class}\ C\{\overline{C}\ \overline{f}; K\ \overline{M}\}$

Constructor
$K \quad ::= \quad C\ (\overline{C}\ \overline{f})\{\mathsf{this}.\overline{f} = \overline{f};\}$

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

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

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

Excluded Layers
$L \quad ::= \quad$ *a set of layer names* $\mid \top \quad (\forall L.L \subseteq \top)$

Layer Definition
$\mathcal{L} \quad ::= \quad \mathsf{layer}\ l\ \{\overline{B}\}$

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

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

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

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

Figure 2. *ContextFJ* Syntax. Note that the term $\mathsf{proceed}(\overline{t})$ may only appear within the body of a method in a layer definition. In addition, the subscript $_L$ on method invocations $t.m_L(\overline{t})$ is usually used during reduction but may appear in the actual program text. Method assumptions Ψ only contain method types of the form $\overline{C} \to C \bullet L$.

Classes and constructors are as in Featherweight Java, although all inheritance related constructions have been removed. Methods have an additional annotation indicating which methods are invoked within the method and not defined in some appropriate class, and thus need to be satisfied by some layer. These will be explained in detail in a moment.

Next we define a collection of named *layers*. Each layer consists of a name and a map from tuples of a method name and a class name, (m, C), to a method definition. The layer name is used to activate and deactivate the layer.

Terms consist of variables and field lookup, as usual. Method invocation is generalized slightly to the form $t.m_L(\overline{t})$, where label L is a set of so-called *excluded* layers. When dispatching m, a binding from a layer in set L cannot be selected. Note that the programmer would write $t.m(\overline{t})$, as normal, which corresponds to $t.m_\emptyset(\overline{t})$ in the formalism.

Three new terms have been added to the language to deal with layer activation and deactivation, and for delegating to an earlier activated layer. The term $\mathsf{with}(l)t$ activates layer l for the duration of the evaluation of expression t. The term $\mathsf{without}(l)t$ deactivates layer l for t. The expression $\mathsf{proceed}(\overline{t})$ delegates the current call to a previously activated layer containing the current method, ignoring all activations of the layer, which it was invoked from. proceed may only appear in the body of methods appearing in a layer.

The most general form of the *method type* is as follows, though this form is not always used in its full generality:

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

where

- 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*, namely the layers in which this method cannot be dispatched to. That is, a binding for method (m, C_0) cannot be taken from some layer in L. These layers are the ones excluded by $\mathsf{without}(-)$ or already visited when doing delegation via proceed.
- Ψ is a *set of method assumptions*, namely the methods used by this method, excluding methods appearing in classes. These 'missing' methods must be provided by some active layer when the method is invoked. These method assumptions have the form: $(m, C_0) \mapsto \overline{C} \to C \bullet L$, which states that a method m of type $\overline{C} \to C$ from class C_0 is called within the given method. L indicates the layers excluded for the m method lookup. In other words, all $\mathsf{with}(l)$ statements, such that method m is defined for class C_0 in l will be ignored for $l \in L$ during such a lookup.

In a number of situations, a less general method type is used. *Methods defined in layers* will have types of the form $[\Psi]\overline{C} \to C$, thus with an empty set of *excluded layers*, which we omit for the sake of brevity. *Methods defined in classes* have types of the form $[\Psi]\overline{C} \to C \bullet \top$. The *top* element of the *excluded layers set* is a technical trick so that only one rule for method invocation is required. *Assumed methods*, such as those appearing in Ψ, are supposed to be resolved by the dynamic context. Their types are of the form: $\overline{C} \to C \bullet L$. Finally, types of *methods provided by a layer* have the form $\overline{C} \to C$.

3. Dynamic Semantics

We introduce the necessary auxiliary notions to define the dynamic semantics. The standard notion of *evaluation context* [14], which is an expression with a single hole in it, is used to define the reduction rules by capturing the notion of the 'next subterm to be reduced.' We write $E[t]$ for the ordinary term obtained by filling the hole in E with t.

The following collects the methods, (m, C) pairs, defined in a layer:

Definition 1 (Layer domain)**.** *Given* layer $l\ \{\overline{B}\}$. *Define:*

$$\mathrm{dom}(l) = \{(m, C_0) \mid (m, C_0) \mapsto M \in \overline{B}\}.$$

The following function computes the methods available in an evaluation context (at the hole), accounting for methods excluded by $\mathsf{without}(l)$ statements.

Definition 2 (Bound Methods)**.**

$$\begin{array}{rcl}
\mathsf{BM}_L([\,]) & = & \emptyset \\
\left.\begin{array}{l}\mathsf{BM}_L(E[[\,].f]) \\ \mathsf{BM}_L(E[[\,].m(\overline{t})]) \\ \mathsf{BM}_L(E[v.m(\overline{v},[\,],\overline{t})]) \\ \mathsf{BM}_L(E[\mathsf{new}\ C(\overline{v},[\,],\overline{t})])\end{array}\right\} & = & \mathsf{BM}_L(E) \\
\mathsf{BM}_L(E[\mathsf{with}(l)[\,]]) & = & \begin{cases}\mathsf{BM}_L(E), & \text{if } l \in L \\ \mathsf{BM}_L(E) \cup \mathrm{dom}(l), & \text{otherwise}\end{cases} \\
\mathsf{BM}_L(E[\mathsf{without}(l)[\,]]) & = & \mathsf{BM}_{L \cup \{l\}}(E)
\end{array}$$

The following function computes the layers excluded by $\mathsf{without}(-)$ in some evaluation context:

Definition 3 (Excluded Layers)**.**

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

It's important to note, that the evaluation semantics we present is syntax-directed, i.e., the order of layer activations is defined by the structure of the reduction context surrounding term to be reduced. The innermost $\mathsf{with}(-)$ or $\mathsf{without}(-)$ statement takes effect before outer ones. Any $\mathsf{with}(-)$ or $\mathsf{without}(-)$ statements occurring in method bodies are analyzed only after the body is substituted. So, at the moment of method invocation, the set of activated layers of the appropriate context is fixed. The lookup functions for methods in both layers and classes and for fields are given in Figure 3.

Figure 4 contains the reduction rules for *ContextFJ*. Rules `(E-With)` and `(E-Without)` discard a layer activation or deactivation when the wrapped expression finishes evaluating. Rule `(E-ProjNew)` is field access. Both rules `(E-InvkLayer)` and `(E-InvkClass)` handle method calls. Rule `(E-InvkLayer)` finds the first available layer, l, containing a binding for method m of class C, from the inside out, which is not excluded by a $\mathsf{without}(-)$ clause or the set L of already visited layers. In addition to substituting the arguments and this, the call to proceed within the method body t is replaced with a call to the same method, except that dispatching the method will skip the present layer l as well as layers in L. This is how proceed is modelled: it invokes the first layer not excluded; this layer is then excluded by subsequent calls to proceed. Rule `(E-InvkClass)` states that if the method is not found in a layer, then it is selected from the appropriate class body.

Method body lookup in layer

$$\frac{\mathsf{layer}\ l\ \{\overline{B}\} \qquad (m, C_0) \mapsto C\ [\Psi]\ m(\overline{C}\ \overline{x})\{\mathsf{return}\ t;\} \in \overline{B}}{\mathsf{lmbody}(l, m, C_0) = (\overline{x}, t)}$$

Method body lookup in class

$$\frac{\mathsf{class}\ C\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \qquad B\ [\Psi]\ m(\overline{B}\ \overline{x})\{\mathsf{return}\ t;\} \in \overline{M}}{\mathsf{mbody}(m, C) = (\overline{x}, t)}$$

Field lookup

$$\frac{\mathsf{class}\ C\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}}{\mathsf{fields}(C) = \overline{C}\ \overline{f}}$$

Figure 3. Lookup definitions

4. Static Semantics

Before presenting the typing rules, we present some auxiliary definitions in Figure 5. These definitions rely on the following notion of *assumption dominance*, which captures that if we are able to satisfy a 'greater' assumption set, then we are able to satisfy 'lesser' one also.

Definition 4 (Assumptions dominance ($\preceq$))**.**

$$\Phi \preceq \Psi \quad \Leftrightarrow \quad \begin{array}{l}\forall (m, C_0) \mapsto \overline{C} \to C \bullet L \in \Phi \cdot \\ \quad \exists (m, C_0) \mapsto \overline{C} \to C \bullet L' \in \Psi\ s.t\ L \subseteq L'\end{array}$$

The key novelty here is in the definitions of mtype. First lookup is performed in the class of the receiver of a method, and then in the assumption set.

In Figure 6 we describe typing rules for methods, layers and classes. All other typing rules, such as `(T-Var)`, `(T-Field)` and `(T-New)` are straightforward adaptions of Featherweight Java [7], and are thus omitted.

Rules `(T-Class)` and `(M OK in C)`, for methods appearing in classes, are standard. The rule `(T-Layer)` en-

$$\frac{}{E[\mathsf{with}(l)v] \to E[v]}\ (\text{E-With})$$

$$\frac{}{E[\mathsf{without}(l)v] \to E[v]}\ (\text{E-Without})$$

$$\frac{\mathsf{fields}(C) = \overline{C}\ \overline{f}}{E[(\mathsf{new}\ C(\overline{v})).f_i] \to E[v_i]}\ (\text{E-ProjNew})$$

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

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

Figure 4. Reduction rules

$$\frac{\mathsf{mtype}(m, C_0, \emptyset) = [\Psi]\overline{C} \to C \quad \mathsf{implies}\ \overline{C} = \overline{D},\ C = D\ \mathsf{and}\ \Phi \preceq \Psi}{\mathsf{override}(m, C_0, [\Phi]\overline{D} \to D)}\ (\text{override}(m, C_0, [\Phi]\overline{C} \to C))$$

$$\frac{CT(C) = \mathsf{class}\ C\ \{\overline{C}\ \overline{f}; K\ \overline{M}\} \quad E\ [\Phi]\ m(\overline{E}\ \overline{x})\{\ \mathsf{return}\ t;\ \} \in \overline{M}}{\mathsf{mtype}(m, C, \Psi) = [\Phi]\overline{E} \to E \bullet \top}\ (\text{mtype-class})$$

$$\frac{m\ \textit{is not defined in}\ C \quad (m, C) \mapsto \overline{E} \to E \bullet L \in \Psi}{\mathsf{mtype}(m, C, \Psi) = \overline{E} \to E \bullet L}\ (\text{mtype-assumptions})$$

Figure 5. Overriding and Method type lookup

sures that all methods are well-defined, and there are no contradictions among their *method assumption sets* with respect to method signatures. The rule `(M OK in l)` for typing a method in a layer performs a substitution of $\mathsf{this}.m_{\{l\}}$ for proceed in order to check the method. This amounts to saying the assumptions for this method will require a method of the same type to dispatch proceed to.

Expression typing is based on the standard typing environment Γ, which is a set of bindings from term variables to types, but also a set of *method assumptions* Ψ. The typing rules for expressions are listed in Figure 7. Rule `(T-Invk)` searches for the type of the method m in both Ψ and C_0. The methods required by this method need to be recorded in Ψ (second last premise). The final premise allows the excluded layer set for this method to grow, meaning the environment needs to work harder to find a suitable method (this is sound).

The rule for $\mathsf{with}(-)$ requires a few auxiliary notions. $\mathsf{provides}(l)$ gives the types of the methods defined in layer l and $\mathsf{requires}(l)$ gives the assumptions of all methods defined in l.

Definition 5. $\mathsf{provides}(l)$ *and* $\mathsf{requires}(l)$ *Given* layer $l\ \{\overline{B}\}$. *Define:*

$$\mathsf{provides}(l) = \left\{ \begin{array}{l} (m, C_0) \mapsto \overline{C} \to D \\ \quad \mid (m, C_0) \mapsto D\ m[\Psi](\overline{C}\ \overline{x})\{...\} \in \overline{B} \end{array} \right\}$$

$$\mathsf{requires}(l) = \biguplus\{\Psi \mid (m, C_0) \mapsto D\ m[\Psi](\overline{C}\ \overline{x})\{...\} \in \overline{B}\}$$

where

$$\begin{array}{rcl} \emptyset \uplus \Psi & = & \Psi \\ ((m, C) \mapsto \ldots \bullet L, \Phi) \uplus \Psi & = & \\ & & \left\{ \begin{array}{l} (m, C) \mapsto \ldots \bullet (L \cup L'), \\ \quad (\Phi \uplus (\Psi/(m, C) \mapsto \ldots) \\ \qquad \textit{if}\ (m, C) \mapsto \ldots \bullet L' \in \Psi \\ (m, C) \mapsto \ldots \bullet L, \Phi \uplus \Psi \\ \qquad \textit{otherwise} \end{array} \right. \end{array}$$

The auxiliary function $\| \cdot \|$ erases all assumptions and excluded layers from a method type:

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

Now in the rule `(T-With)` the wrapped expression t is typed in an environment where the provided methods of the layer, $\mathsf{provides}(l)$, are available (Φ). In addition, the required methods must appear in the context surrounding the entire expression (Ψ) which provides all of the required methods used by the layer *and* the unsatisfied assumptions of expression t.

The rule `(T-Without)` modifies the *assumption set* to exclude all methods occurring in the excluded layer l from Ψ. This is achieved using the following definition:

Definition 6. *Layer exclusion from assumptions*

$$\begin{array}{rcl} \epsilon \rceil l & = & \epsilon \\ ((m, C) \mapsto X \bullet L, \Psi) \rceil l & = & \\ & & \left\{ \begin{array}{l} (m, C) \mapsto X \bullet L \cup \{l\}, (\Psi \rceil l) \\ \qquad \textit{if}\ (m, C) \in \mathrm{dom}(l) \\ (m, C) \mapsto X \bullet L, (\Psi \rceil l) \\ \qquad \textit{otherwise} \end{array} \right. \end{array}$$

$$\frac{K = C(\overline{C}\ \overline{f})\{\ \mathsf{this}.\overline{f} = \overline{f};\ \}\quad \overline{M}\ \mathsf{OK\ in}\ C}{\mathsf{class}\ C\ \{\ \overline{C}\ \overline{f};\ K\ \overline{M}\ \}\ \mathsf{OK}}\ (\text{T-CLASS})$$

$$\frac{\begin{array}{c}\mathsf{layer}\ l\ \{\overline{B}\}\quad \overline{B}\ \mathsf{OK\ in}\ l \\ \forall \Phi_1, \Phi_2 \in \{\Phi \mid (m, C_0) \mapsto C\ m\ [\Phi](\overline{C}\ \overline{x})\{\ldots\} \in \overline{B}\} \\ ((n, D) \mapsto \overline{E} \to E \bullet L_1 \in \Phi_1\ \textit{and}\ (n, D) \mapsto \overline{F} \to F \bullet L_2 \in \Phi_2) \\ \textit{implies}\ \overline{E} = \overline{F}\ \textit{and}\ E = F\end{array}}{\mathsf{layer}\ l\ \{\overline{B}\}\ \ \mathsf{OK}}\ (\text{T-LAYER})$$

$$\frac{CT(C) = \mathsf{class}\ C\ \{\ \ldots\ \}\quad \Psi; (\overline{x} : \overline{C}, \mathsf{this} : C) \vdash t_0 : D}{D\ m[\Psi](\overline{C}\ \overline{x})\{\ \mathsf{return}\ t_0;\ \}\ \ \mathsf{OK\ in\ C}}\ (\text{M OK IN C})$$

$$\frac{\begin{array}{c}\mathsf{layer}\ l\ \{\ldots\} \\ \Psi \uplus \mathsf{provides}(l); (x : \overline{C}, \mathsf{this} : C_0) \vdash \{\mathsf{proceed} \mapsto \mathsf{this}.m_{\{l\}}\}t_0 : C \\ \mathsf{override}(m, C_0, [\Psi]\overline{C} \to C)\end{array}}{(m, C_0) \mapsto C\ m[\Psi](\overline{C}\ \overline{x})\{\ \mathsf{return}\ t_0;\ \}\ \ \mathsf{OK\ in}\ l}\ (\text{B OK IN } l)$$

Figure 6. Typing methods, layers and classes

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

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

$$\frac{\Psi; \Gamma \vdash t : C\quad \Psi' = \Psi \rceil l}{\Psi'; \Gamma \vdash \mathsf{without}(l)t : C}\ (\text{T-WITHOUT})$$

Figure 7. (Selected) Expression typing

5. Discussion

The present approach groups the assumptions/requirements of a layer at the granularity of a layer. This means that when a method from a layer is used, all assumptions of the layer need to be satisfied for the program to type check. Indeed, simply activating a layer without even using it imposes such a requirement. A more fine grained approach would involve collecting the method calls which are actually made within an expression and recording only those per method and only requiring that the methods actually used from an active layer need to have an appropriate binding.

The course-grained approach is more modular, in the sense that small changes to code do not necessitate changes to the interface of layers. The fine-grained approach is more precise and does not require activating layers which will not be used just to satisfy the type checker.

We expect that much of our annotations can be inferred, resulting in a significantly simpler system for a programmer to use.

6. Related Work

Our dynamic semantics draws heavily from the semantics of dynamic binding [8, 11]. These models do not have without$(-)$. Furthermore, their type system does not exclude failure of dynamic binding due to a missing binding. The semantics of proceed adapts ideas from Schippers et al [15], although our approach does not use the notions of *heap* and *stack*, and we provide a type system.

Our type system heavily drew inspiration from Contextual Modal Type Theory [12], which had neither proceed nor without$(-)$. In particular, our method types $[\Psi]\overline{C} \to C$ resemble contextual modal types.

The calculus of evolving objects [3] gives a foundational account of highly dynamic systems. Their setting is quite different from ours and our dynamic semantics is significantly simpler. Quite a few programming languages have concepts similar to layers and dynamic binding. These include Haskell's implicits [10], Clojure [5], Scala's implicits [13], and Groovy's [9] and Objective C's [4] categories. Due to space limitations we cannot give a detailed comparison.

7. Conclusion and Future Work

This paper presented the first type-sound semantic for a context-oriented language with layers. The semantics is based on Featherweight Java and the type system ensures that all dispatched methods find an appropriate binding either in some layer or in the original collection of classes.

Syntax

$$
\begin{array}{rcl}
\Delta & ::= & \overline{l = B} \\
B & ::= & \overline{p = e} \\
v & ::= & \lambda x.e \mid x \\
e & ::= & v \mid p \mid e\ e \mid \mathsf{with}(l)e \mid \mathsf{without}(l)e \\
E[\,] & ::= & [\,] \mid E[[\,]\ e] \mid E[v\ [\,]] \mid E[\mathsf{with}(l)[\,]] \\
 & \mid & E[\mathsf{without}(l)[\,]]
\end{array}
$$

Excluded layers

$$
\begin{array}{rcl}
\mathsf{XL}([\,]) & = & \emptyset \\
\left.\begin{array}{l}\mathsf{XL}(E[[\,]\ e]) \\ \mathsf{XL}(E[v\ [\,]]) \\ \mathsf{XL}(E[\mathsf{with}(l)[\,]])\end{array}\right\} & = & \mathsf{XL}(E) \\
\mathsf{XL}(E[\mathsf{without}(l)[\,]]) & = & \{l\} \cup \mathsf{XL}(E)
\end{array}
$$

Bound parameters

$$
\begin{array}{rcl}
\mathsf{BP}_L([\,]) & = & \emptyset \\
\mathsf{BP}_L(E[[\,]\ e]) & = & \mathsf{BP}_L(E) \\
\mathsf{BP}_L(E[v\ [\,]]) & = & \mathsf{BP}_L(E) \\
\mathsf{BP}_L(E[\mathsf{with}(l)[\,]]) & = & \begin{cases} \mathsf{BP}_L(E) \text{ if } l \in L \\ \mathsf{BP}_L(E) \cup \mathrm{dom}(\Delta(l)) \\ \quad \text{otherwise} \end{cases} \\
\mathsf{BP}_L(E[\mathsf{without}(l)[\,]]) & = & \mathsf{BP}_{L \cup \{l\}}(E)
\end{array}
$$

Reduction rules

$$
\begin{array}{rcl}
E[(\lambda x.e)\ v] & \rightarrow & E[e[v/x]] \\
E[\mathsf{with}(l)v] & \rightarrow & E[v] \\
E[\mathsf{with}(l)E'[p]] & \rightarrow & E[\mathsf{with}(l)E'[e]] \\
 & & \text{if} \begin{cases} p \notin \mathsf{BP}_\emptyset(E'[\,]) \\ l \notin \mathsf{XL}(E') \\ e = \Delta(l)(p) \end{cases} \\
E[\mathsf{without}(l)v] & \rightarrow & E[v]
\end{array}
$$

Figure 8. Contextλ

Directions for future work include extending the core language to incorporate inheritance and subtyping, including inheritance between layers, and considering adding dependencies between the layers, such as that one layer requires another (kind of) layer to be present, or that two layers can never be active at the same time.

One of the main problems to address is ambiguity, which occurs because there may be multiple candidate methods available: does a less specific method in a closer layer take precedence over a more specific method?

A. Contextλ

For reference and comparison, Figure 8 presents the dynamic semantics for context-oriented λ-calculus *without* proceed. The combination of proceed and closures causes problems, and is addressed by Clarke et al. [1].

References

[1] Dave Clarke, Pascal Costanza, and Éric Tanter. How should context-escaping closures proceed? Submitted to COP2009, 2009.

[2] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: an overview of contextl. In *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*, pages 1–10, New York, NY, USA, 2005. ACM.

[3] Mariangiola Dezani-Ciancaglini, Paola Giannini, and Oscar Nierstrasz. A Calculus of Evolving Objects. In *MPOOL'08*, 2008. http://www.di.unito.it/ dezani/papers/dgn.pdf.

[4] Andrew Duncan. *Objective- C Pocket Reference*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[5] Rich Hickey. The Clojure programming language. In *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, pages 1–1, New York, NY, USA, 2008. ACM.

[6] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, 7(3):125–151, 2008.

[7] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.

[8] Oleg Kiselyov, Chung chieh Shan, and Amr Sabry. Delimited dynamic binding. *SIGPLAN Not.*, 41(9):26–37, 2006.

[9] Guillaume Laforge. Groovy: An agile dynamic language for the Java Platform, 2008.

[10] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark Shields. Implicit parameters: Dynamic scoping with static types. In *POPL*, pages 108–118, 2000.

[11] Luc Moreau. A syntactic theory of dynamic binding. *Higher-Order and Symbolic Computation*, 11:727–741, 1998.

[12] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Logic*, 9(3):1–49, 2008.

[13] Martin Odersky. The Scala Language Specification. Available from http://www.scala-lang.org/, 2004.

[14] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[15] Hans Schippers, Dirk Janssens, Michael Haupt, and Robert Hirschfeld. Delegation-based semantics for modularizing crosscutting concerns. *SIGPLAN Not.*, 43(10):525–542, 2008.

[16] Eddy Truyen. *Dynamic and Context-Sensitive Composition in Dis uted Systems*. PhD thesis, K.U.Leuven, Belgium, November 2004.