A Smooth Combination of Role-based Languages and Context Activation

Tetsuo Kamina and Tetsuo Tamai
University of Tokyo
{kamina,tamai}@acm.org
Purpose

- Language constructs for context-awareness
  - Primary concept for many applications
    - Adaptive UI based on user’s profile
    - Location-aware information services
  - Important for recent application areas

- Explicit treatment for context-specific behaviors
  - Modularization of context-specific behaviors
  - Composition/decomposition of context-specific behaviors

- Simple theoretical framework for “context-awareness” in languages
Role-based languages

- EpsilonJ: An adaptive role model based language (Tamai, 2005)
  - Context is modeled as a collaboration field between roles
  - Context can be instantiated
  - Context instance can be dynamically composed with class instance

```java
context Company {
    role Employer {
        void pay() {
            Employer.getPaid();
        }
    }
    role Employee {
        void getPaid() { ... }
    }
}

Company todai = new Company();
Person tanaka = new Person();
todai.Employer.newBind(tanaka);
((todai.Employer)tanaka).pay();
```

Context activation by downcast
- No control of scoping
- Not type-safe
Context-oriented programming

- Layers
  - Modularization concept orthogonal to classes
  - Contain partial method definitions
  - Can be activated/deactivated dynamically at run-time
- Scope of context activation is explicitly controlled

```java
Person tanaka = new Person();
with (Company) {
    System.out.println(tanaka); // printing the Company specific info.
}
```

- COP focuses on behavioral variations of the same method
  - Composition of unrelated behaviors is not considered in ContextJ
- Context-dependent behavior is class based
Our proposal: NextEJ

- Extension of EpsilonJ with the features of COP (Kamina, 09)
  - Taking both advantages of EpsilonJ and COP

- Formalization
An example

- Featuring two contexts: building and shop
  - building has roles
    - guest
    - administrator
    - security agent
    - owner
  - shop has roles
    - customer
    - shopkeeper
- Interactions among roles
  - A security agent notifies all the guests in the case of emergency
  - A shopkeeper sells the customer an item
- Shops may be inside a building
Context and role declarations

The same structure with EpsilonJ

class Building {
    role Guest {
        void escape() { ... }
    }
    role Security {
        void notify() { Guest.escape(); }
    }
}

- A context is a set of roles
- Contexts and roles can be instantiated
- A role instance depends on its enclosing context instance
- Multiple role instances with the same context instance
Object adaptation and context activation

- Role instance is created in the `bind` sentence and composed with corresponding class instance
- Type of each class instance is changed to the mixin composition
- Roles can be deactivated and activated again

```java
Building midtown = new Building();
Person tanaka = new Person();
Person suzuki = new Person();
Person sato = new Person();
bind tanaka with midtown.Guest(),
    suzuki with midtown.Guest(),
    sato with midtown.Security() {
    ...
    sato.notify();
}
```
Multiple context activation

```java
Building midtown = new Building();
Person tanaka = new Person();
Person sato = new Person();
bind tanaka with midtown.Guest(),
    sato with midtown.Guest() {
    ...
    Shop starbucks = new Shop();
    bind tanaka with starbucks.Customer(),
        sato with starbucks.Shopkeeper() {
        tanaka.buy(caffeMocha);
    }
}
```

- **bind** can be nested
- **tanaka**, a guest of **midtown** is also a customer of **starbucks**
Swapping roles

- Context is deactivated outside the `bind` sentences

- Decomposition of deactivated context is allowed in NextEJ
  - Another object can assume the decomposed role of context

```java
Person sato = new Person();
bind sato with midtown.Employee from tanaka {
    ...
}
```

role discarded by `tanaka` and taken over by `sato`
Required interface

- Requiring the binding object to provide the implementation

```java
context Building {
    role Guest requires {String name();} {
        void foo() { ... name(); ... }
        ... }
}
```

- `name()` is imported to `Guest`
- The imported method may be overridden
- Structural subtyping between role and class
FEJ: the core calculus

- Purely functional core of NextEJ based on FJ (Igarashi, 2001)
  - FJ + dynamic composition and activation of contexts

- An object is followed by a sequence of role instances:
  \[ \text{new } C(e) \oplus r \]

- Run-time expression language
Syntax

- Named types
  \[ T ::= \text{C.R} \mid \overline{\text{C.R}}::\text{C} \]

- Interface types
  \[ Ts ::= T \mid \{ \overline{\text{Mi}} \} \quad \text{Mi} = T \text{m}(\overline{T \overline{x}}); \]

- Class and role declarations
  \[ L ::= \text{class C} \{ \overline{T \overline{f}}; \overline{M \overline{A}} \} \]
  \[ A ::= \text{role R requires} \{ \overline{\text{Mi}} \} \{ \overline{T \overline{f}}; \overline{M} \} \]

- Expressions
  \[ e ::= x \mid e.f \mid e.m(\overline{e}) \mid \text{new C}(\overline{e})\oplus\overline{r} \mid \text{bind } \overline{x} \text{ with } \overline{r} \text{ from } \overline{y} \{ \overline{xy}.e_0 \} \]
Subtyping

- Reflexive and transitive closure induced by mixin composition

\[
\begin{align*}
Ts <: Ts & \quad S <: T \quad T <: U \\
& \quad S <: U & \quad C.R::T <: T \\
& \quad C.R::T <: C.R
\end{align*}
\]

- Structural subtyping b/w class and interface

\[
T \ m(T \ x); \in \overline{Mi} \Rightarrow \ mtype(m, C) = \overline{T} \rightarrow \overline{T}
\]

\[
C <: \{ \overline{Mi} \}
\]
Dynamic semantics (method invocation)

- Method invocation reduces the body of method declaration
- The method is not found in roles:
  - Substituting formal parameters and this
    \[ v = \text{new } C(\overline{v'}) \oplus r \]
    \[ \text{mbody}(m, \overline{r}) \text{ is undefined} \]
    \[ \text{mbody}(m, \text{new } C(\overline{v'})) = x.e \]
    \[ v.m(\overline{v}) \rightarrow [v/x, \text{new } C(\overline{v'})/\text{this}]e \]
- The method is found in roles:
  - Substituting formal parameters, this, and super
    \[ v = \text{new } C(\overline{v'}) \oplus r \]
    \[ r = r_1, w.R(\overline{e}), r_2 \]
    \[ \text{mbody}(m, \text{new } C(\overline{v'})) = x.e, w.R(\overline{e}) \]
    \[ \text{cp}(v) = \text{new } C(\overline{v'}) \oplus r_2 \]
    \[ v.m(\overline{v}) \rightarrow [v/x, \text{new } C(\overline{v'})/\text{this}, \text{cp}(v)/\text{super}]e \]
Dynamic semantics (bind expression)

- Bind expression reduces its body
  - Substituting free variables with values appearing in `bind` and `from`
  - Role instances appearing in `with` are composed with values from `bind` and decomposed with values from `from`

\[
\text{bind } \overline{\nu} \text{ with } \overline{\tau} \text{ from } \overline{w} \{ \overline{x}\overline{y}.e \} \rightarrow [\overline{\nu} \oplus \overline{\tau}]/\overline{x},(\overline{w}-\overline{\tau})/\overline{y}]e
\]
Expression typing

- Field access and method invocation are the same as those of FJ

$$
\Gamma \vdash x : \Gamma(x) \quad \frac{\Gamma \vdash e_0 : S \quad \text{ftype}(f, S) = T}{\Gamma \vdash e_0.f : T}
$$

$$
\Gamma \vdash e_0 : S \quad \frac{\Gamma \vdash e : S}{\text{mtype}(m, Ts) = T \rightarrow T \quad \bar{S} <: \bar{T}}
\quad \Gamma \vdash e_0.m(\bar{e}) : T
$$

- Typing rule for `new` checks that all the role instances are wellformed

$$
\text{fields}(C) = \bar{T} \bar{f} \quad \Gamma \vdash \bar{e} : \bar{S} \quad \bar{S} <: \bar{T}

\frac{r_i = d_i.R_i(\bar{c}_i) \quad \Gamma \vdash d_i : U_i}{U_i <: C_i \quad \Gamma \vdash \text{roleOK}(C_i, R_i, \bar{c}_i, C)}
\quad \Gamma \vdash \text{new } C(\bar{e}) \oplus \bar{r} : C.R : C
$$
Expression typing (bind expression)

- Environment $\Gamma$ is updated in the first hypothesis
  - In environment where variables $\overline{x}$ from $\textbf{bind}$ are mixin compositions and variables $\overline{y}$ from $\textbf{from}$ are mixin decomposition, the body is well-typed
- All the role instances are well-typed

\[
\Gamma(x:C.R::\Gamma(x), y:\Gamma(y)/C.R) \vdash e_0:T
\]
\[
\quad \quad r_i = d_i.R_i(c_i) \quad \Gamma \models \overline{x}:S
\]
\[
\Gamma \models \overline{d}:U \quad U <: C \quad \Gamma \models \text{roleOK}(C_i, R_i, c_i, S_i)
\]
\[
\Gamma \models \overline{y}:V \quad \Gamma \models \text{unbindAllowed}(V_i, C.R)
\]
\[
\Gamma \models \textbf{bind} \overline{x} \textbf{ with } \overline{r} \textbf{ from } \overline{y} \{ \overline{xy}e_0 \} : T
\]
Properties

• Subject reduction: If $\Gamma \vdash e : T$ and $e \rightarrow e'$, then $\Gamma \vdash e' : S$ for some $S <: T$

• Progress: If $\Gamma \vdash e : T$ and there exist no $e'$ such that $e \rightarrow e'$, then $e$ is a value

• Type soundness: If $\phi \vdash e : T$ and $e \rightarrow^* e'$ with $e'$ a normal form, then $e'$ is a value $v$ with $\phi \vdash v : S$ and $S <: T$
Related work

- ObjectTeams (Hermann, 2003, 2007)
  - Supporting context-dependent behavior
    - lowering
    - lifting
  - Grouping of context-dependent behavior
  - Binding is class-based denoted by the name of class
- CaesarJ (Mezini, 2002)
  - Deploying and undeploying aspects at any time
  - CaesarJ: binding is specified in the binding classes
  - NextEJ: binding is specified at the time of binding
Conclusion

- NextEJ: a smooth combination of EpsilonJ and COP
  - Solving the typing problem of EpsilonJ
  - Integrating context activation and composition of (possibly unrelated) behaviors

- FEJ: the core calculus of NextEJ
  - Ensuring type soundness
Thanks!