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

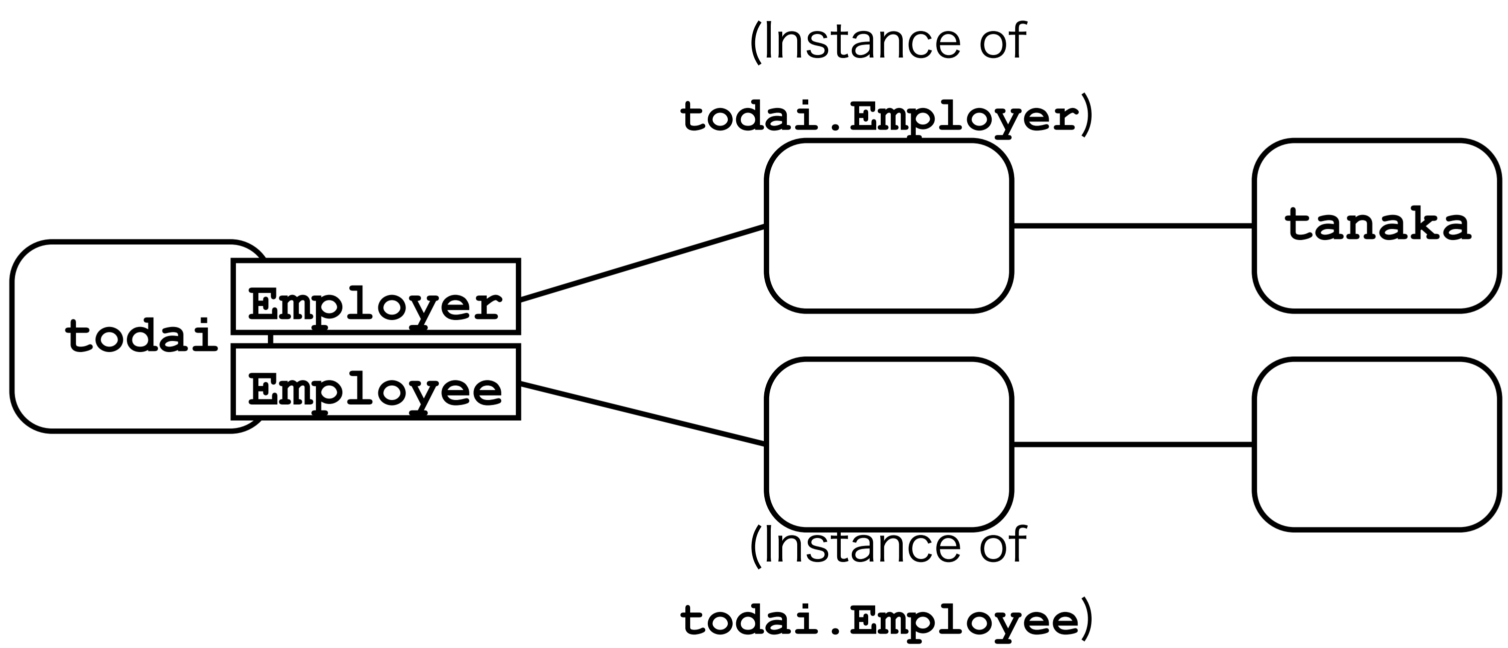
- Explicit treatment for context-specific behaviors
  - modularization of context-specific behaviors
  - composition/decomposition of context-specific behaviors

- 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

```
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

- Representative work: ContextJ, ContextL, ContextS (Hirschfeld et al., 2005, 2007, 2008)
- 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

```
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

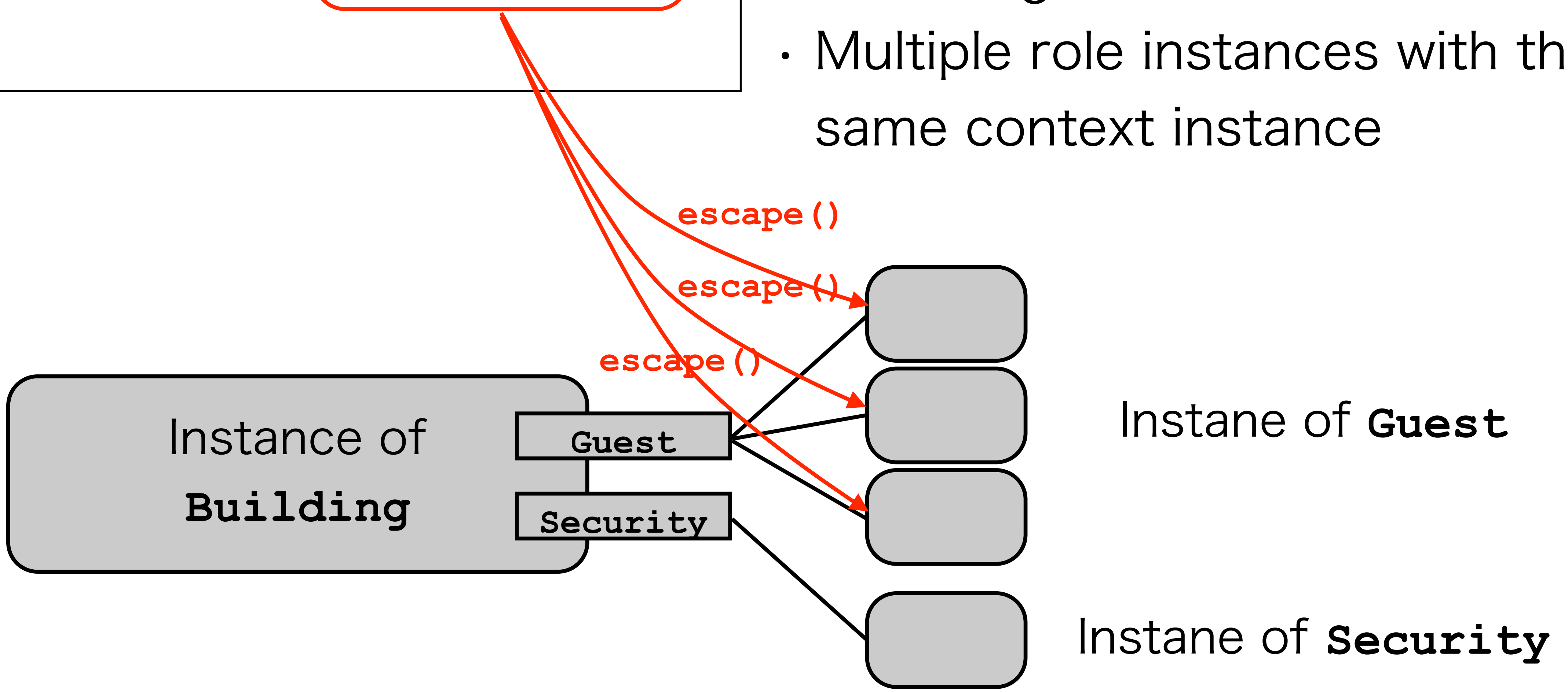
- Featuring two contexts: building and shop
    - building has roles
        - guest
        - administrator
        - security agent
        - owner
    - shop has roles
        - customer
        - shopkeeper
- Interactions among roles
    - A security agent notifies all the guests in the case of emergency
    - 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(); }
  }
}
```

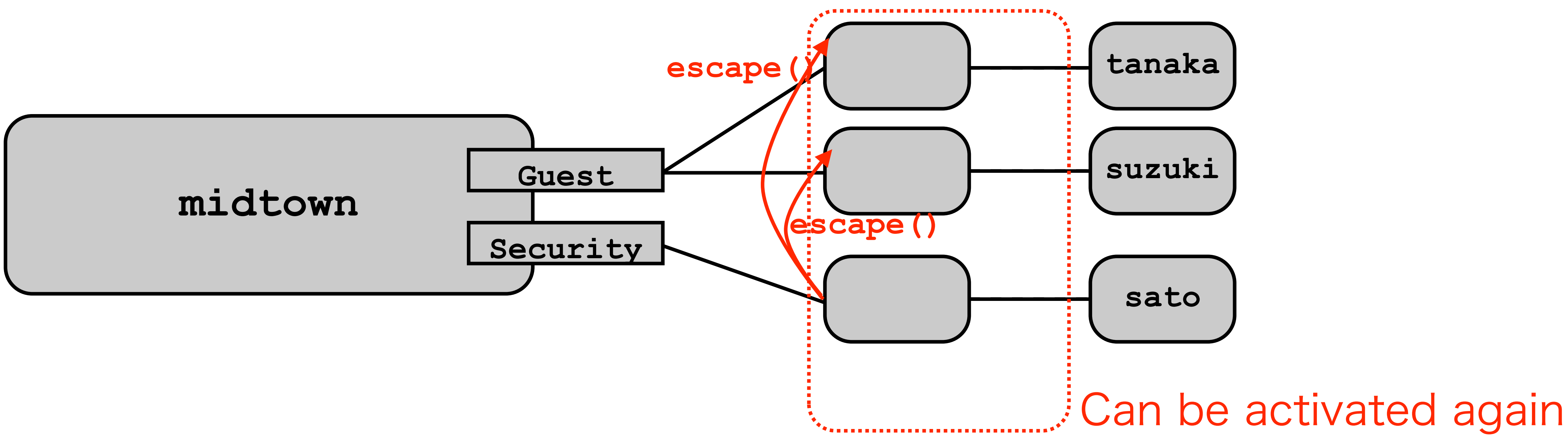
- A context is a set of roles
- Contexts and roles can be instantiated
- A role instance depends on its enclosing context instance
- Multiple role instances with the same context instance



Object adaptation and context activation

```
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();
}
```

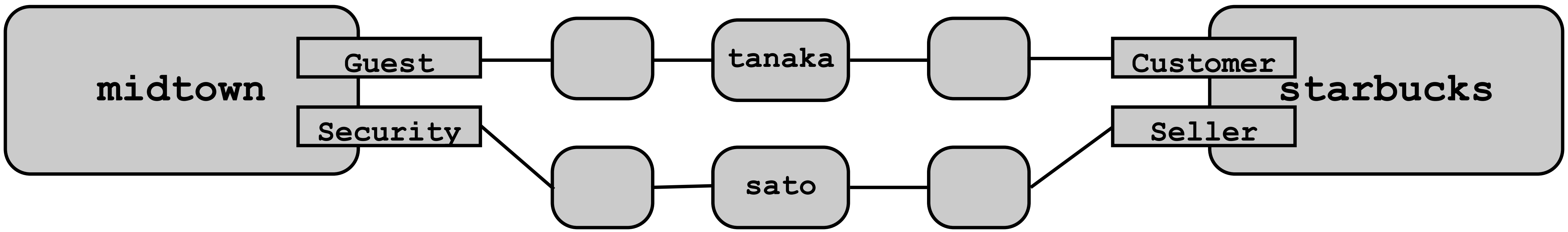
- Role instance is created in the **bind** sentence and composed with corresponding class instance
- Type of each class instance is changed to the mixin composition
- Roles can be deactivated and activated again



Multiple context activation

```
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

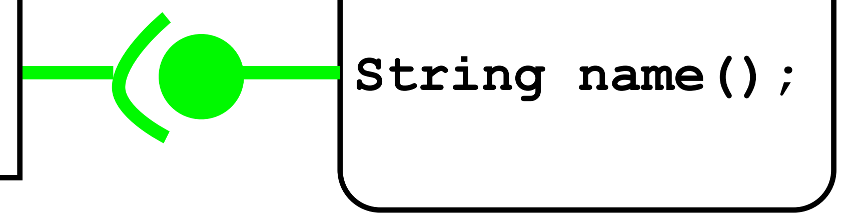
```
Person sato = new Person();
bind sato with midtown.Employee from tanaka {
  ...
}
```

role discarded by **`tanaka`**
and taken over by **`sato`**

Required interface

- Requiring the binding object to provide the implementation

```
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:

$$\texttt{new}\ \mathrm{C}(\overline{\mathrm{e}})\oplus\overline{\mathrm{r}}$$

- Run-time expression language

Syntax

- Named types

$$T ::= C.R \mid \overline{C}.\overline{R}::C$$

- Interface types

$$Ts ::= T \mid \{\ \overline{Mi}\ \} \qquad Mi = T\ m(\overline{T}\ \overline{x});$$

- Class and role declarations

$$L ::= \text{class } C\ \{\ \overline{T}\ \overline{f};\ \overline{M}\ \overline{A}\ \}$$

$$A ::= \textbf{role } R\ \textbf{requires}\ \{\ \overline{Mi}\ \}\ \{\ \overline{T}\ \overline{f};\ \overline{M}\ \}$$

- Expressions

$$e ::= x \mid e.f \mid e.m(\overline{e}) \mid \text{new } C(\overline{e}) \oplus \overline{r} \mid \textbf{bind}\ \overline{x}\ \textbf{with}\ \overline{r}\ \textbf{from}\ \overline{y}\ \{\ \overline{x}\overline{y}.e_0\ \}$$

Subtyping

- Reflexive and transitive closure induced by mixin composition

$$\mathrm{Ts} <: \mathrm{Ts} \qquad \frac{\mathrm{S} <: \mathrm{T} \qquad \mathrm{T} <: \mathrm{U}}{\mathrm{S} <: \mathrm{U}} \qquad \begin{array}{c}\mathrm{C.R::T} <: \mathrm{T} \\ \mathrm{C.R::T} <: \mathrm{C.R}\end{array}$$

- Structural subtyping b/w class and interface

$$\frac{\mathrm{T}\ \mathrm{m}(\overline{\mathrm{T}}\ \overline{\mathrm{x}}); \in \overline{\mathrm{M}}\mathrm{i} \;\Rightarrow\; \mathrm{mtype}(\mathrm{m}, \mathrm{C}) = \overline{\mathrm{T}} \to \mathrm{T}}{\mathrm{C} <: \{\ \overline{\mathrm{M}}\mathrm{i}\ \}}$$

Dynamic semantics (method invocation)

- Method invocation reduces the body of method declaration
- The method is not found in roles:
  - Substituting formal parameters and **`this`**

$$\frac{v = \texttt{new}\ C(\overline{v'}) \oplus \overline{r} \qquad mbody(m, \overline{r}) \text{ is undefined} \qquad mbody(m, \texttt{new}\ C(\overline{v'})) = \overline{x}.e}{v.m(\overline{v}) \rightarrow [\overline{v}/\overline{x}, \texttt{new}\ C(\overline{v'})/\texttt{this}]e}$$

- The method is found in roles:
  - Substituting formal parameters, **`this`**, and **`super`**

$$\frac{v = \texttt{new}\ C(\overline{v'}) \oplus \overline{r} \qquad r = \overline{r_1}, w.R(\overline{e}), \overline{r_2} \qquad mbody(m, \texttt{new}\ C(\overline{v'})) = x.e, w.R(\overline{e}) \qquad cp(v) = new\ C(\overline{v'}) \oplus \overline{r_2}}{v.m(\overline{v}) \rightarrow [\overline{v}/\overline{x}, \texttt{new}\ C(\overline{v'})/\texttt{this}, cp(v)/\texttt{super}]e}$$

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`**

$$\texttt{\textbf{bind}}\ \overline{v}\ \texttt{\textbf{with}}\ \overline{r}\ \texttt{\textbf{from}}\ \overline{w}\ \{\ \overline{x}\overline{y}.e\ \} \rightarrow [(\overline{v}\oplus\overline{r})/\overline{x},(\overline{w}\text{-}\overline{r})/\overline{y}]e$$

Expression typing

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

$$\Gamma \vdash x : \Gamma(x) \qquad \frac{\Gamma \vdash e_0 : S \qquad ftype(f, S) = T}{\Gamma \vdash e_0.f : T} \qquad \frac{\Gamma \vdash e_0 : S \qquad \Gamma \vdash \overline{e} : \overline{S} \qquad mtype(m, Ts) = \overline{T} \rightarrow T \qquad \overline{S} <: \overline{T}}{\Gamma \vdash e_0.m(\overline{e}) : T}$$

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

$$\frac{fields(C) = \overline{T}\ \overline{f} \qquad \Gamma \vdash \overline{e} : \overline{S} \qquad \overline{S} <: \overline{T} \qquad r_i = d_i.R_i(\overline{c_i}) \qquad \Gamma \vdash d_i : U_i \qquad U_i <: C_i \qquad \Gamma \vdash roleOK(C_i, R_i, \overline{c_i}, C)}{\Gamma \vdash \mathbf{new}\ C(\overline{e}) \oplus \overline{r} : \overline{C}.\overline{R}::C}$$

Expression typing (bind expression)

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

$$
\frac{\begin{array}{c}
\Gamma(\overline{x}:\overline{C}.\overline{R}::\Gamma(\overline{x}),\ \overline{y}:\Gamma(\overline{y})/\overline{C}.\overline{R}) \vdash e_0 : T \\
r_i = d_i.R_i(\overline{c_i}) \qquad \Gamma \vdash \overline{x}:\overline{S} \\
\Gamma \vdash \overline{d}:\overline{U} \qquad \overline{U} <: \overline{C} \qquad \Gamma \vdash \mathrm{roleOK}(C_i, R_i, \overline{c_i}, S_i) \\
\Gamma \vdash \overline{y}:\overline{V} \qquad \Gamma \vdash \mathrm{unbindAllowed}(V_i, \overline{C}.\overline{R})
\end{array}}{\Gamma \vdash \mathbf{bind}\ \overline{x}\ \mathbf{with}\ \overline{r}\ \mathbf{from}\ \overline{y}\ \{\ \overline{x}\overline{y}.e_0\ \} : T}
$$

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!