Flexible Object Adaptation for Java-like Languages

Tetsuo Kamina and Tetsuo Tamai

The University of Tokyo
{kamina,tamai}@acm.org

Abstract. To realize object adaptability with a clear conceptual framework, a role model *Epsilon* was proposed. The novelty of Epsilon model is its ability to change object's behavior dynamically. However, such kind of flexibility also easily brings type-unsafety and other unreliabilities. This paper proposes a small core language ϵ that formalizes some key concepts of object adaptation, which is informally described in the Epsilon model. In ϵ, three kinds of objects, context instances, role instances, and class instances exist at run-time. A role instance is a member of a context instance where how role instances are collaborating with each other is encapsulated. A class instance can dynamically assume a role by binding itself with a role instance, and can also throw the role away. Their relationship can change during computation, and ϵ's type system assures that the computation does not go wrong, even though some exceptional cases concerning downcasting exist. This formalization clarifies the essential features of the object adaptation incorporated in Epsilon model and provides a solid base for program analysis and language processor implementation.

1 Introduction

Considerable research efforts have been devoted to make objects in object-oriented systems more flexible and adaptable. The recent interest in self-managed (or autonomic, self-healing, adaptive) systems/computing indicates renewed attention on this target.

Objects in the conventional object-oriented languages are created from fixed templates defined as classes and once they are created, it is hard to change their structures and behaviors dynamically. One way of enabling dynamic changes to objects is to fully employ the mechanism of meta-programming or reflection and allow free transformation of objects at run-time. Some languages such as Ruby[27] provide dynamic object structure change capability as their innate feature. The obvious problem with such a feature and meta-programming in general is performance decline. But even if performance is somehow ensured at a certain level, taking advantage of sophisticated optimizing techniques, there will still remain the problem of programming difficulties and error-proneness.

In [25], Tamai et al. proposed a role model *Epsilon* and a language based on the model *EpsilonJ* (the revised version is also available in [26]). The aim of this model was to realize object adaptability with a clear conceptual framework. A collaboration field called environment or context can be defined by a set of roles that interact with each other. An object can dynamically participate in a collaboration field and assume one of its roles so that it acquires functions of the role and capability of collaborating with other roles in the field. An object may assume multiple roles of different collaboration fields at a time so that it grows into a complex object with rich functions but its behavior can be clearly comprehensible from the base behavioral properties of roles.

However, such kind of flexibility also easily brings type-unsafety and other unreliabilities. To avoid such unreliabilities, constructs of Epsilon should formally be defined. In [25] or [26], however, only a brief description (described by examples) on the semantics of the language EpsilonJ was provided. A full language specification is released on the Web[24] but its description is informal. In fact, even though there are many formal studies on collaboration-based design, relatively few efforts have been made on formalizing object adaptation.

In this paper, we propose a small core calculus ϵ that formalizes some key concepts of object adaptation, described in [25]. In ϵ, three kinds of objects, context instances that represent collaboration fields, role instances, and class instances exist at run-time. A role instance is a member of a context instance where collaboration between role instances is encapsulated. A class instance can dynamically assume a role by binding itself with a role instance, and can also throw the role away. Their relationship can be changed during computation, and ϵ's type system assures that the computation does not go wrong, even though some exceptional cases concerning downcasting exist.

Even though ϵ is quite similar to EpsilonJ, it is not designed to be a small subset of EpsilonJ. Instead, the aim of this work is to understand the essences of object adaptation; therefore, there are some differences between EpsilonJ and ϵ. In short, ϵ puts more emphasis on stating clear language semantics, while EpsilonJ provides much liberal ways for writing programs. Nevertheless, this formalization clarifies the essential features of the object adaptation mechanism incorporated in EpsilonJ and provides a solid base for program analysis and language processor implementation.

2 An Overview of Object Adaptation

To make this paper self-contained, we briefly summarize the main features of object adaptation that are formerly described in [25] as Epsilon model. In this section, we informally describe these features by using Java-like syntax.

In the Epsilon model, three kinds of objects, context instances, role instances, and class instances, exist at run-time. A context instance is a collaboration field where role instances interact with each other. A role instance is a member of a context instance, and there may be multiple role instances associated with a context instance. A set (or a sequence) of role instances associated with a context instance is called a *role group*. Behind the scenes, contexts are augmented by an internal field representing the role group. A class instance is the same as in

```
context Company {
  role Employer {
    void pay() {
      Employee.getPaid();} }
  role Employee {
    int save, salary;
    Employee(int salary) {
      this.salary = salary;}
    void getPaid() {
      save += salary; } } }
```

```
class Person {
  int money; }

Person tanaka = new Person();
Person komiyama = new Person();
Company todai = new Company();
todai.Employer.newBind(komiyama);
todai.Employee.newBind(tanaka,1000);
((Company.Employer)komiyama).pay();
```

Fig. 1. Declaration of the context `Company` and object adaptation.

conventional Java-like languages, except that it can dynamically participate in a collaboration field (represented by a context instance) by assuming one of its role instances so that it acquires functions of the role instance and capability of collaborating with other role instances. Behind the scenes, classes are also augmented by an internal field representing the set of assuming roles.

How a context is declared is demonstrated in Fig.1. A context is declared using context declaration that begins with the keyword `context` followed by the name of context (that is `Company` in Fig.1). A Role is declared as a member of the context using role declaration that begins with the keyword `role`. In Fig.1, two roles, `Employer` and `Employee`, are declared. Each `context` and `role` is declared with fields, methods and constructors just like classes.

A role group is associated with the enclosing context instance and referred by the role name. We can access each instance of role group by using an iterator that iterates over the role group. In Fig.1, however, we use a more convenient syntactic sugar; we can apply a method declared in the role to the whole role group, and the method is invoked for all the role instances. Thus, the method call `Employee.getPaid()` is interpreted as calling the method `getPaid` of all the `Employee`'s instances.

In Epsilon model, contexts and roles are the first class constructs at runtime as well as at model description time. A context is instantiated by the `new` expression; the expression `new Company()` creates an instance of `Company`. A role instance is created by a special operation `newBind` that performs two things; (1) it creates a role instance that is a member of the receiver context; and (2) it binds the role instance with the class instance provided as an argument of `newBind`. In Fig.1, the instances of `Person`, `komiyama` and `tanaka`, assume the roles `todai.Employer` and `todai.Employee`, respectively. Note that the second and the following arguments, if any, of `newBind` call are arguments for constructor call of roles. After the binding, the object bound to the role acquires an access to the role instance and thus can use the role methods. Furthermore, it acquires type of the role, and the role methods are invoked through type-casting. In fact, the type of expression `(Company.Employer)komiyama` is a mixin composition[17] `Company.Employer::Person`, where `::` is a composition operator, thus we can safely access the `pay()` method declared in `Company.Employer`. Whether komiyama can be cast to Company.Employer or not (i.e. whether komiyama assumes a role instance of Company.Employer or not) is checked at run-time.

The binding of a class instance and a role instance may be dissolved at run-time; for this purpose, any role implicitly declares unbind method. For example, the following piece of code

```
((Company.Employee)tanaka).unbind();
```

firstly casts tanaka to its role Company.Employee, then calls unbind. After calling unbind, the connection between tanaka and Company.Employee is dissolved, and the dissolved role instance becomes garbage. Instead, another class instance may assume the unbound role instance if we use the swap method, which is also implicitly declared in a role. For example, the following code

```
Person sato = new Person();
((Company.Employee)tanaka).swap(sato);
```

results in that sato takes over tanaka's Company.Employee role.

There should be some interaction between a class instance and a role instance that are bound together so that the state and the behavior of the class instance should be affected by the binding. For this purpose, there is a way of defining an interface to a role and this is used at the time of binding with a class instance, requiring the class instance to supply the interface. This interface is declared with the requires phrase as follows.

```
role Employee requires { void deposit(int); } {
  void sendSalary(int salary) { deposit(salary); } }
```

When a required interface is declared to a role, methods can be imported to the role from the class instance. For example, suppose the class Person has a method deposit:

```
class Person { ... int money;
  void deposit(int s) { money+=s; } }
```

After the binding of todai.Employee.newBind(tanaka) in the previous program piece, the method deposit(int) of tanaka is imported to the Employee role instance through the interface. When an interface method is overridden by the corresponding role method, the replacing method of the binding object becomes hidden. If there is a need for invoking the hidden method in the context, either in the body of the overriding method or in other role (or context) methods, it is possible to invoke it by attaching the qualifier super to the method name.

Since the role instance requires the class instance to supply the requires interface, the class has to implement it. Note that the requires interface may be anonymous, just as shown in the above program. In other words, the class has to be a *structural subtype* of the requires interface. A similar mechanism is also found in [17].

$$\begin{array}{rcl}
A & ::= & C \mid X \\
T & ::= & A \mid X.R \mid X.R :: C \\
T_S & ::= & T \mid \{\ \bar{M_I}\ \} \\
L_C & ::= & \texttt{class}\ C\ \lhd\ D\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \} \\
L_R & ::= & \texttt{role}\ R\ \texttt{requires}\ \{\ \bar{M_I}\ \}\{\bar{T}\ \bar{f};\ \bar{M}\} \\
L_X & ::= & \texttt{context}\ X\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{L_R}\ \} \\
M & ::= & T\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \\
M_I & ::= & T\ m(\bar{T}\ \bar{x}); \\
r & ::= & e_0.R(\bar{e}) \\
e & ::= & x \mid e.f \mid e.m(\bar{e}) \mid (\texttt{new}\ C(\bar{e}), \bar{r}) \mid \texttt{new}\ X(\bar{e}) \mid e_0.R.\texttt{newBind}(e, \bar{d}) \mid \\
 & & e_0.\texttt{unbind}() \mid e_0.\texttt{swap}(e) \mid (X.R)e \\
v & ::= & (\texttt{new}\ C(\bar{v}), \bar{r}) \mid \texttt{new}\ X(\bar{v}) \mid (X.R)(\texttt{new}\ C(\bar{v}), \bar{r})
\end{array}$$

Fig. 2. Abstract syntax

Finally, we note that a class instance may assume multiple roles; for example, a person can be a customer, a patient, and an employee depending on the context. Current context of the person may change through downcasting. Furthermore, a class instance may change roles even within a context; for example, a person can change its role from an employee to an employer, which is possible because a class instance may discard a role and assume another role dynamically. By using the `swap` operation, the state of the old employer (e.g., unfinished tasks, responsibilities, and so on) is taken over by the new employer.

3 ϵ: the core calculus of Epsilon

This section provides a small core calculus ϵ of Epsilon model. This formalization is based on FJ[16], a minimum core language of Java, but includes some additional features found in the full Java language such as `super` calls that are needed to model important features of object adaptation.

Syntax. The abstract syntax of ϵ is shown in Fig.2. In this paper, the metavariable A ranges over class or context names; S, T, and U range over named types; T_S ranges over types (including `requires` interface); C, D and E range over class names; R ranges over role names; X ranges over context names; L_C ranges over class declarations; L_R ranges over role declarations; L_X ranges over context declarations; f and g range over field names; M and N range over method declarations; M_I ranges over interface method declarations; m ranges over method names; b, c, d and e range over expressions; x ranges over variables; r and s range over role instances; v and w range over values.

We put an over-line for a possibly empty sequence. Furthermore, we abbreviate pairs of sequences in a similar way, writing "$\bar{T}\ \bar{f}$" as a shorthand for "$T_1\ f_1, \cdots, T_n\ f_n$", where n is the length of $\bar{T}$ and $\bar{f}$, and so on (the same way introduced in [16]).

Object adaptation can be realized with some imperative features. However, imperative features will introduce complexity in the type system. We can take another approach to concentrate on the new features incorporated in Epsilon model. We design ϵ as a purely functional calculus; i.e. the state of context instance never changes after the constructor invocation (in this paper, constructor declarations are abbreviated and constructor invocations become implicit). To model the dynamic semantics of object adaptation, we regard each class instance as a pair of a constructor invocation and a sequence of roles that the class instance is bound with. Therefore, ϵ is considered as a runtime expression language, since the programmer does not write ($\texttt{new}\ C(\bar{e}), \bar{r}$) but only $\texttt{new}\ \ C(\bar{e})$, which is identical to ($\texttt{new}\ C(\bar{e}), \cdot$). The role instances $\bar{r}$ are generated during the evaluation and needed in the rules to check and maintain the roles of the class instance.

In ϵ, there are two kinds of types: named types and interface types. A named type is represented by a class name, a context name, a role name prefixed by a context name, or a mixin composition in the form of $X.R :: C$. These types may appear in field declarations, formal parameter types and return types. On the other hand, interface types, denoted by $\{\ \bar{M_I}\ \}$, may appear only in the `requires` clause.

As in FJ, we assume that the set of variables includes the special variable `this`, which is considered to be implicitly bound in every method declaration. Furthermore, we also assume that the set of variables includes the special variable `super`, which is considered to be implicitly bound in every role method declaration.

For the reduction and typing rules, we need a few auxiliary definitions, given in Fig.3. We write $m \notin \bar{M}$ to mean that the method definition of the name m is not included in $\bar{M}$. The fields of type T, written *fields*(T), is a sequence $\bar{T}\ \bar{f}$ pairing the type of each fields with its name. The type of method m in type T_S, written *mtype*(m, T_S), is a pair, written $\bar{T} \rightarrow T_0$, of a sequence of formal parameter types $\bar{T}$ and its return type T_0. If T is a role type $X.R$ and m is not found in $X.R$, its `requires` interface is searched. If T is a mixin composition, the left operand of `::` is searched first. Similarly, the body of method m in type T, written *mbody*(m, T), is a pair, written $(\bar{x}, e)$, of a sequence of formal parameters $\bar{x}$ and an expression e.

We also present a rule that checks whether a role can be bound to a class instance or not. The following predicate *bindable* is used for this checking. An instance of role $X.R$ can be bound to an instance of class C if C is a subtype of $X.R$'s required interface $\{\ \bar{M_I}\ \}$.

Subtyping rules of ϵ are shown in Fig.4. Subtyping is a reflexive and transitive closure induced by the subclassing relation. Furthermore, a class is a subtype of an interface if the class implements all the methods declared in the interface. This subtyping rule is used in checking whether a role can be bound to a class or not. There also exists some straightforward subtyping rules regarding mixin composition.

Field lookup:

$$\frac{\begin{array}{c}\texttt{class}\ C \ \lhd\ D\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \}\\ \mathit{fields}(D) = \bar{S}\ \bar{g}\end{array}}{\mathit{fields}(C) = \bar{S}\ \bar{g}, \bar{T}\ \bar{f}}$$

$$\frac{\begin{array}{c}\texttt{context}\ X\ \{\ \bar{T}\ \bar{f};\ \bar{N}\ \bar{L}_R\}\\ \texttt{role}\ R\ \texttt{requires}\ \{\ \cdots\ \}\{\ \bar{S}\ \bar{g};\ \cdots\ \} \in \bar{L}_R\end{array}}{\mathit{fields}(X.R) = \bar{S}\ \bar{g}}$$

$$\frac{\texttt{context}\ X\ \{\ \bar{T}\ \bar{f};\ \bar{N}\ \bar{L}_R\ \}}{\mathit{fields}(X) = \bar{T}\ \bar{f}}$$

$$\frac{\mathit{fields}(X.R) = \bar{S}\ \bar{g} \qquad \mathit{fields}(C) = \bar{T}\ \bar{f}}{\mathit{fields}(X.R :: C) = \bar{T}\ \bar{f}; \bar{S}\ \bar{g}}$$

$$\frac{\mathit{fields}(T) = \bar{T}\ \bar{f}}{\mathit{ftype}(f_i, T) = T_i}$$

Method body lookup:

$$\frac{\begin{array}{c}\texttt{class}\ C \ \lhd\ D\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \}\\ T\ m(\bar{S}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}\end{array}}{\mathit{mbody}(m, C) = (\bar{x}, e)}$$

$$\frac{\texttt{class}\ C \ \lhd\ D\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \} \qquad m \notin \bar{M}}{\mathit{mbody}(m, C) = \mathit{mbody}(m, D)}$$

$$\frac{\begin{array}{c}\texttt{context}\ X\ \{\ \cdots \bar{M}\ \bar{L}_R\ \}\\ T\ m(\bar{S}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}\end{array}}{\mathit{mbody}(m, X) = (\bar{x}, e)}$$

$$\frac{\begin{array}{c}\texttt{context}\ X\ \{\ \cdots \bar{L}_R\ \}\\ \texttt{role}\ R\ \texttt{requires}\ \{\ \bar{M}_I\ \}\{\ \cdots\ \bar{M}\ \} \in \bar{L}_R\\ T\ m(\bar{S}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}\end{array}}{\mathit{mbody}(m, X.R :: C) = (\bar{x}, e)}$$

$$\frac{\begin{array}{c}\texttt{context}\ X\ \{\ \cdots \bar{L}_R\ \} \qquad m \notin \bar{M}\\ \texttt{role}\ R\ \texttt{requires}\ \{\ \bar{M}_I\ \}\{\ \cdots\ \bar{M}\ \} \in \bar{L}_R\end{array}}{\mathit{mbody}(m, X.R :: C) = \mathit{mbody}(m, C)}$$

Method type lookup:

$$\frac{\begin{array}{c}\texttt{class}\ C \ \lhd\ D\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \}\\ T\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}\end{array}}{\mathit{mtype}(m, C) = \bar{T} \to T}$$

$$\frac{\texttt{class}\ C \lhd D\ \{\bar{T}\ \bar{f};\ \bar{M}\} \qquad m \notin \bar{M}}{\mathit{mtype}(m, C) = \mathit{mtype}(m, D)}$$

$$\frac{\begin{array}{c}\texttt{context}\ X\ \{\ \bar{T}\ \bar{f};\ \bar{N}\ \bar{L}_R\ \}\\ T\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}\\ \texttt{role}\ R\ \texttt{requires}\ \{\bar{M}_I\}\{\cdots\ \bar{M}\} \in \bar{L}_R\end{array}}{\mathit{mtype}(m, X.R) = \bar{T} \to T}$$

$$\frac{\begin{array}{c}\texttt{context}\ X\ \{\bar{T}\ \bar{f}; \bar{N}\ \bar{L}_R\} \qquad m \notin \bar{M}\\ \texttt{role}\ R\ \texttt{requires}\ \{\bar{M}_I\}\{\cdots\ \bar{M}\} \in \bar{L}_R\end{array}}{\mathit{mtype}(m, X.R) = \mathit{mtype}(m, \{\bar{M}_I\})}$$

$$\frac{T\ m(\bar{T}\ \bar{x}); \in \bar{M}_I}{\mathit{mtype}(m, \{\bar{M}_I\}) = \bar{T} \to T}$$

$$\frac{\mathit{mtype}(m, X.R) = \bar{T} \to T}{\mathit{mtype}(m, X.R :: C) = \bar{T} \to T}$$

$$\frac{\mathit{mtype}(m, X.R)\ \text{is undefined}}{\mathit{mtype}(m, X.R :: C) = \mathit{mtype}(m, C)}$$

$$\frac{\begin{array}{c}\texttt{context}\ X\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{L}_R\ \}\\ T\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}\end{array}}{\mathit{mtype}(m, X) = \bar{T} \to T}$$

Binding check:

$$\frac{\begin{array}{c}C\ \texttt{<:}\ \{\bar{M}_I\} \qquad \texttt{context}\ X\ \{\cdots\ \bar{L}_R\}\\ \texttt{role}\ R\ \texttt{requires}\ \{\bar{M}_I\}\{\ \cdots\ \} \in \bar{L}_R\end{array}}{\mathit{bindable}(X.R, C)}$$

Fig. 3. Auxiliary definitions

$$T_S \mathrel{<:} T_S \qquad \frac{\texttt{class}\ C \ \lhd\ D\ \{\ \cdots\ \}}{C \mathrel{<:} D} \qquad \frac{C \mathrel{<:} D \qquad D \mathrel{<:} E}{C \mathrel{<:} E}$$

$$\frac{T\ m(\bar{T}\ \bar{x}); \in \bar{M}_I \Rightarrow \mathit{mtype}(m, C) = \bar{T} \to T}{C \mathrel{<:} \{\bar{M}_I\}}$$

$$X.R :: C \mathrel{<:} C \qquad X.R :: C \mathrel{<:} X.R \qquad \frac{D \mathrel{<:} C}{X.R :: D \mathrel{<:} X.R :: C}$$

Fig. 4. Subtyping rules

An ϵ program is a pair (CT, e) of a *class table* CT and an expression e. A class table is a map from class names and context names to class declarations and context declarations, respectively. The expression e may be considered as the `main` method of the "real" program. The class table is assumed to satisfy the following conditions: (1) $CT(C) = \texttt{class}\ C\ \ \cdots$ for every $C \in \mathit{dom}(CT)$; (2) $CT(X) = \texttt{context}\ X\ \ \cdots$ for every $X \in \mathit{dom}(CT)$; (3) all roles R in $CT(X)$ are uniquely named; (4) $T \in \mathit{dom}(CT)$ for every class name, context name, and role name appearing in $\mathit{range}(CT)$.

Dynamic semantics. The reduction rules of ϵ are shown in Fig.5. The reduction relation is of the form $e \longrightarrow e'$, read "expression e reduces to expression e' in one step." We write $\longrightarrow^*$ for the reflective and transitive closure of $\longrightarrow$.

There are two rules for field access (as in FJ, we assume that all the field names are distinct); one is field access to a class instance or context instance (the rule R-FIELD)[1], and the other is field access to a role instance through type casting (the rule R-RFIELD). Note that R-RFIELD shows that a field access to a role instance reduces to the corresponding actual argument for the role constructor. During the computation, a class instance has to retain the *states* of the role instances which the class instance is bound with, which is why we formulate a class instance as a pair of a class constructor invocation and a sequence of role instances.

Similarly, there are two rules for method invocation. The method invocation reduces to the expression of the method body, substituting all the parameters $\bar{x}$ with the argument expressions $\bar{e}$ and the special variable `this` with the receiver of method invocation. The rule R-RINVK shows the case of role method invocation, where the variable `super` is also substituted with the receiver of method invocation (removing type casting). The rule R-BIND shows that a `newBind` expression takes a class instance as an argument, creates a role instance, and binds it with the argument class instance. The rule R-UNBIND shows that an `unbind` expression reduces to the receiver class instance of `unbind`, removing the designated role (by $\bar{r} - r$, we mean the role r is removed from the sequence $\bar{r}$). The rule R-SWAP shows that a `swap` expression takes a class instance as an argument

[1] We use `new` $X(\bar{e})$ and $(\texttt{new}\ X(\bar{e}), \cdot)$ interchangeably.

$$\frac{\mathit{fields}(A) = \bar{T}\ \bar{f}}{(\texttt{new}\ A(\bar{e}), \bar{r}).f_i \longrightarrow e_i} \qquad \text{(R-FIELD)}$$

$$\frac{\mathit{fields}(X.R :: C) = \bar{T}\ \bar{f} \qquad (\texttt{new}\ X(\bar{d})).R(\bar{e}) \in \bar{r} \qquad \bar{b}, \bar{e} = \bar{c}}{((X.R)(\texttt{new}\ C(\bar{b}), \bar{r})).f_i \longrightarrow c_i} \qquad \text{(R-RFIELD)}$$

$$\frac{\mathit{mbody}(m, A) = (\bar{x}, e_0)}{(\texttt{new}\ A(\bar{e}), \bar{r}).m(\bar{d}) \longrightarrow [\bar{d}/\bar{x}, (\texttt{new}\ A(\bar{e}), \bar{r})/\texttt{this}]e_0} \qquad \text{(R-INVK)}$$

$$\frac{\mathit{mbody}(m, X.R :: C) = (\bar{x}, e_0) \qquad (\texttt{new}\ X(\bar{d})).R(\bar{c}) \in \bar{r}}{\begin{array}{c}((X.R)(\texttt{new}\ C(\bar{e}), \bar{r})).m(\bar{d}) \longrightarrow \\ [\bar{d}/\bar{x}, (X.R)(\texttt{new}\ C(\bar{e}), \bar{r})/\texttt{this}, (\texttt{new}\ C(\bar{e}), \bar{r})/\texttt{super}]e_0\end{array}} \qquad \text{(R-RINVK)}$$

$$\frac{(\texttt{new}\ X(\bar{b})).R(\bar{d}) \notin \bar{r}}{\begin{array}{c}(\texttt{new}\ X(\bar{b})).R.\texttt{newBind}((\texttt{new}\ C(\bar{e}), \bar{r}), \bar{d}) \longrightarrow \\ (\texttt{new}\ C(\bar{e}), \bar{r}(\texttt{new}\ X(\bar{b})).R(\bar{d}))\end{array}} \qquad \text{(R-BIND)}$$

$$\frac{(\texttt{new}\ X(\bar{e})).R(\bar{d}) \in \bar{r}}{\begin{array}{c}((X.R)(\texttt{new}\ C(\bar{c}), \bar{r})).\texttt{unbind()} \longrightarrow \\ (\texttt{new}\ C(\bar{c}), \bar{r} - (\texttt{new}\ X(\bar{e})).R(\bar{d}))\end{array}} \qquad \text{(R-UNBIND)}$$

$$\frac{(\texttt{new}\ X(\bar{e})).R(\bar{e'}) \in \bar{r}}{\begin{array}{c}((X.R)(\texttt{new}\ C(\bar{c}), \bar{r})).\texttt{swap}((\texttt{new}\ D(\bar{d}), \bar{s})) \\ \longrightarrow (\texttt{new}\ D(\bar{d}), \bar{s}(\texttt{new}\ X(\bar{e})).R(\bar{e'}))\end{array}} \qquad \text{(R-SWAP)}$$

Fig. 5. Reduction rules

and binds it with the designated role, removing the class instance that is the receiver of `swap` from the context instance where the designated role resides.

Reduction rules may be applied to any subexpressions of an expression, so we also need the obvious congruence rules, which are omitted in this paper.

Typing. The typing rules for ϵ expressions are shown in Fig.6. An environment Γ is a finite mapping from variables to types, written $\bar{x} : \bar{T}$. The typing judgment for expressions has the form $\Gamma\ \vdash\ e : T$, read "in the environment Γ, expression e has type T."

The rules are syntax directed, with one rule for each form of expressions. The typing rules for method invocations and constructors check that each actual parameter has a type of the corresponding formal parameter. The rule T-INVK checks that the type of receiver of method invocation may be an interface type, thus method call to `super` is allowed. The rule T-NEW checks that all the role instances with which the class instance binds are also well-typed; i.e., the context

Expression typing:

$$\frac{\varGamma \vdash e_0 : S \qquad \mathit{ftype}(f, S) = T}{\varGamma \vdash e_0.f : T} \quad \text{(T-FIELD)}$$

$$\varGamma \vdash x : \varGamma(x) \quad \text{(T-VAR)}$$

$$\frac{\begin{array}{c}\varGamma \vdash e_0 : T_S \qquad \varGamma \vdash \bar{e} : \bar{S} \\ \mathit{mtype}(m, T_S) = \bar{T} \to T \qquad \bar{S} \texttt{<:} \bar{T}\end{array}}{\varGamma \vdash e_0.m(\bar{e}) : T} \quad \text{(T-INVK)}$$

$$\frac{\varGamma \vdash e : X.R :: C}{\varGamma \vdash e.\texttt{unbind}() : C} \quad \text{(T-UNBIND)}$$

$$\frac{\begin{array}{c}\mathit{fields}(C) = \bar{T}\ \bar{f} \\ \varGamma \vdash \bar{e} : \bar{S} \qquad \bar{S} \texttt{<:} \bar{T} \\ \text{for } r_j \in \bar{r}\ r_j = (\texttt{new } X_j(\bar{c}_j)).R_j(\bar{d}_j) \\ \varGamma \vdash \texttt{new } X_j(\bar{c}_j) : X_j \\ \mathit{fields}(X.R) = \bar{T}_j\ \bar{g}_j \\ \varGamma \vdash \bar{d}_j : \bar{S}_j \qquad \bar{S}_j \texttt{<:} \bar{T}_j\end{array}}{\varGamma \vdash (\texttt{new } C(\bar{e}), \bar{r}) : C} \quad \text{(T-NEW)}$$

$$\frac{\begin{array}{c}\varGamma \vdash e : X.R :: C \\ \varGamma \vdash d : D \qquad \mathit{bindable}(X.R, D)\end{array}}{\varGamma \vdash e.\texttt{swap}(d) : D} \quad \text{(T-SWAP)}$$

$$\frac{\begin{array}{c}\mathit{fields}(X) = \bar{T}\ \bar{f} \qquad \varGamma \vdash \bar{e} : \bar{S} \\ \bar{S} \texttt{<:} \bar{T}\end{array}}{\varGamma \vdash \texttt{new } X(\bar{e}) : X} \quad \text{(T-CNEW)}$$

$$\frac{\begin{array}{c}\varGamma \vdash e : C \qquad \mathit{bindable}(X.R, C) \\ \varGamma \vdash e_0 : X \qquad \bar{S} \texttt{<:} \bar{T} \\ \mathit{fields}(X.R) = \bar{T}\ \bar{f} \qquad \varGamma \vdash \bar{d} : \bar{S}\end{array}}{\varGamma \vdash e_0.R.\texttt{newBind}(e, \bar{d}) : C} \quad \text{(T-BIND)}$$

$$\frac{\varGamma \vdash e : C}{\varGamma \vdash (X.R)e : X.R :: C} \quad \text{(T-CAST)}$$

Wellformed definitions:

$$\frac{\begin{array}{c}\bar{x} : \bar{T}, \texttt{this} : C \vdash e_0 : T_0 \\ \texttt{class } C \ \lhd\ D\{ \cdots \}\end{array}}{T_0\ m(\bar{T}\ \bar{x})\{\ \texttt{return } e_0;\ \}\ \texttt{OK IN } C} \quad \text{(T-METHOD)}$$

$$\frac{\bar{M}\ \texttt{OK IN } X.R}{\begin{array}{c}\texttt{role } R \texttt{ requires } \{\ \bar{M_I}\ \}\{\ \bar{T}\ \bar{f};\ \bar{M}\ \} \\ \texttt{OK IN } X\end{array}} \quad \text{(T-ROLE)}$$

$$\frac{\bar{M}\ \texttt{OK IN } C}{\texttt{class } C \ \lhd\ D\{\ \bar{T}\ \bar{f};\ \bar{M}\}\ \texttt{OK}} \quad \text{(T-CLASS)}$$

$$\frac{\begin{array}{c}\bar{x} : \bar{T}, \texttt{this} : X \vdash e_0 : T_0 \\ \texttt{context } X\ \{ \cdots \}\end{array}}{T_0\ m(\bar{T}\ \bar{x})\{\ \texttt{return } e_0;\ \}\ \texttt{OK IN } X} \quad \text{(T-XMETHOD)}$$

$$\frac{\begin{array}{c}\bar{x} : \bar{T}, \texttt{this} : X.R, \texttt{super} : \{\ \bar{M_I}\ \} \vdash e_0 : T_0 \\ \texttt{context } X\ \{ \cdots\ \bar{L_R}\} \\ \texttt{role } R\ \{\ \bar{M_I}\ \}\{ \cdots \} \in \bar{L_R}\end{array}}{T_0\ m(\bar{T}\ \bar{x})\{\ \texttt{return } e_0;\ \}\ \texttt{OK IN } X.R} \quad \text{(T-RMETHOD)}$$

$$\frac{\bar{M}\ \texttt{OK IN } X \qquad \bar{L_R}\ \texttt{OK IN } X}{\texttt{context } X\ \{\ \bar{T}\ \bar{f}; \bar{M}\ \bar{L_R}\}\ \texttt{OK}} \quad \text{(T-CONTEXT)}$$

Fig. 6. Typing rules

instance of each role is well-typed, and each actual parameters of each role constructor has a type of the corresponding formal parameter. The rules T-BIND and T-SWAP check that the receiver role and argument class instance of `newBind` and `swap`, respectively, are compatible.

The type system assures that the receiver of `newBind` is a context, and the type of receiver of `unbind` and `swap` is a mixin composition of a role type and a class type, i.e., only (role) type casting expressions can be a receiver of `unbind` and `swap` operations.

Finally, we show the typing rules for method declarations, class declarations, context declarations, and role declarations. The rules for wellformed definitions are also shown in Fig.6. The type of the body of a method declaration is a subtype of the return type. The special variable `this` is bound in every method declaration, and for every method declaration in roles, a variable `super` is also bound. A class declaration is wellformed if all the methods declared in that class are wellformed. A role declaration is wellformed if all the methods declared in that role are wellformed. A context declaration is wellformed if all the methods and roles declared in that context are wellformed.

Finally, we show the properties of ϵ, which is every well-typed expression evaluates to a value or an expression containing casts, `newBind`, `unbind`, or `swap` that cannot be reduced further.

Theorem 1 (Subject Reduction). *If $\Gamma \vdash e : T$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : T'$ for some T' `<:` T.*

Theorem 2 (Progress). *If $\emptyset \vdash e : T$ and e is neither (1) a value, (2) an expression containing $(X.R)(\texttt{new}\ C(\bar{e}), \bar{r})$ where $(\texttt{new}\ X(\bar{b})).R(\bar{d}) \notin \bar{r}$ for some $\bar{b}, \bar{d}$, (3) an expression containing $e_0.R.\texttt{newBind}((\texttt{new}\ C(\bar{e}), \bar{r}))$ where $e_0.R(\bar{d}) \in \bar{r}$ for some $\bar{d}$, (4) an expression containing $((X.R)(\texttt{new}\ C(\bar{e}), \bar{r})).\texttt{unbind}()$ where $(\texttt{new}\ X(\bar{b})).R(\bar{d}) \notin \bar{r}$ for some $\bar{b}, \bar{d}$, and (5) an expression containing $((X.R)(\texttt{new}\ C(\bar{e}), \bar{r})).\texttt{swap}((\texttt{new}\ D(\bar{c}), \bar{s}))$ where either $(\texttt{new}\ X(\bar{b})).R(\bar{d}) \in \bar{s}$ or $(\texttt{new}\ X(\bar{b})).R(\bar{d}) \notin \bar{r}$ for some $\bar{b}, \bar{d}$, then $e \longrightarrow e'$ for some e'.*

Remark. Because of ϵ's ability to assume and discard roles at run time, the progress theorem shows that there are some unreliabilities associated to adaptable objects. The result shows that we have to accept the possibility that access to role's fields or methods or `newBind`/`unbind`/`swap` operations can fail at run time. In the full language, of course, such a failure does not stop whole the program; it generates an exception that can be caught by a surrounding exception handler.

One may consider that a more satisfactory type soundness result would be obtained by changing the typing rules. For example, we may change the rule T-NEW to make the result type be $(C, \bar{r})$ so that the T-BIND rule can check that in the type $(C, \bar{r})$ of `newBind`'s first argument e, there are no instance of $X.R$ in $\bar{r}$. However, it is hard to make this approach cooperate with other constructs such as `if` statements. With imperative features, some dynamic checking should be necessary. Even in purely functional languages, there may be a situation where we want to change which role the class instance is bound with according to the condition of `if` expression, but putting emphasis on type-safety prevents providing such flexibility. On the other hand, EpsilonJ's way of thinking is to provide convenient idioms such as downcasting that most conventional languages support, to make programmers flexibly bind roles and objects at their own risk. To our knowledge there are no pieces of work on object adaptation that carefully inspect the semantics of downcasting.

4 Discussions and Related Work

While developing ϵ, we found some significant differences between EpsilonJ and ϵ. Firstly, in ϵ every role instance binds with a class instance, and role instance methods and fields can be accessed only through type casting. EpsilonJ does not hold this property and we can write unsafe programs by explicitly accessing role instances. Another contribution of this work w.r.t. EpsilonJ is that it provides solid information for language processor implementation. For example, current implementation of EpsilonJ is based on reflective APIs, which results in significant performance degradation [26]. On the other hand, ϵ indicates that we can employ a more "natural" way to implement the language; e.g. a class may have a field that contains a set of role instances with which the class instance binds. Furthermore, type casting is realized as an operation that selects a role instance from that set, and `super` calls are modeled as delegations. Indeed, we have developed an EpsilonJ translator to Java based on this idea [20].

The programming language powerJava[3] is a quite similar language with EpsilonJ, in that roles and collaboration fields are the first class constructs, interaction between roles are encapsulated, and objects can participate in the interaction by assuming one of its roles. As in ϵ, the type of role depends on the enclosing context instance. However, powerJava lacks the feature of role groups that is a powerful mechanism of getting role instances associated with the context instance reflectively. Furthermore, no formalization is given for powerJava.

Delegation Layers[21] and Object Teams[14] provides more flexible object based composition of collaborations. For example, Delegation Layers combines the mechanism of delegation[18, 22] and virtual classes[19, 7], or Family Polymorphism[10]; roles may be represented by virtual classes, and composition is instance-base using delegation mechanism. Both of these approaches, however, do not successfully represent object adaptation described in this paper. For example, in ϵ the object after assuming a role may dynamically throw the role away, and even the thrown role may be assumed by another object and states held in the role instance are taken over by the latter object.

There are pieces of literature that formalize the feature of extending objects at run-time. Ghelli presented foundations for extensible objects with roles based on Abadi-Cardelli's object calculi[1], where coexistence of different methods introduced by incompatible extensions is considered [12]. Gianantonio et al. presented a calculus $\lambda Obj+$[13], an extension of λObj[11] with a type assignment system that allows self-inflicted object extension still statically catching the "message not found" errors. Drossopoulou et al. proposed a type-safe core language *Fickle*[9] that allows re-classification of objects, a mechanism of dynamically changing object's belonging classes which share the same "root" superclass. On the other hand, ϵ focuses on a foundation of object adaptation for Java-like languages (based on FJ) and the feature of assuming roles that are thrown by other objects (by `swap` operation).

Mixins[6, 2, 17] are similar to roles in EpsilonJ in that mixins form partial definitions that can be reused with a number of classes that conform the *requirements* of mixins. Even though mixin composition is originally performed at compile time, dynamic composition of mixins is also studied in a core calculus[4], and such kind of object level inheritance is also studied as *wrappers*[8, 5]. Dynamic trait (a stateless mixin) substitution is also studied in [23]. All of these pieces of work put more emphasis on type-safety, while ϵ supports more sophisticated mechanism such as the `swap` operation and object level downcasting to roles.

EpsilonJ supports context-oriented programming (COP)[15] in that contexts (layers in COP terms) are named first-class entities that can be referred to explicitly at run-time, and context-dependent object behavior can be changed by activating/deactivating contexts from anywhere in the code. In EpsilonJ, such activation/deactivation is performed by type casting.

5 Concluding Remarks

This paper reports a minimum core calculus of Epsilon model that has the notable feature of representing object adaptation. The calculus ϵ provides a precise, formal definition of all key essential features of Epsilon model. Its type system assures that the computation does not go wrong, even though some exceptional cases concerning downcasting exist. The formalization clarifies the essential features of object adaptation and provides solid information for program analysis and language processor implementation. For example, ϵ suggests a natural way to implement EpsilonJ, which has been partly achieved by the latest implementation.

References

1. Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.
2. Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam – designing a Java extension with mixins. *ACM TOPLAS*, 25(5):641–712, 2003.
3. M. Baldoni, G. Boella, and L. van der Torre. Interaction between objects in powerJava. *Journal of Object Technology*, 6(2):5–30, 2007.
4. Lorenzo Bettini, Viviana Bono, and Silvia Likavec. Safe and flexible objects with subtyping. *Journal of Object Technology*, 4(10):5–29, 2005.
5. Lorenzo Bettini, Sara Capecchi, and Elena Giachino. Weatherweight Wrap Java. In *SAC'07*, pages 1094–1100, 2007.
6. G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA 1990*, pages 303–311, 1990.

7. Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *ECOOP'98*, volume 1445 of *LNCS*, pages 523–549, 1998.
8. Martin Buchi and Wolfgang Weck. Generic wrappers. In *ECOOP 2000*, volume 1850 of *LNCS*, pages 201–225, 2000.
9. Sophia Drossopoulou, Ferruccio Damiani, and Mariangiola Dezani-Ciancaglini. Fickle: Dynamic object re-classification. In *ECOOP 2001*, volume 2072 of *LNCS*, pages 130–149, 2001.
10. Eric Ernst. Family polymorphism. In *ECOOP 2001*, volume 2072 of *LNCS*, pages 303–327, 2001.
11. K. Fisher, F. Honsell, and J.C̃. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.
12. Giorgio Ghelli. Foundations for extensible objects with roles. *Information and Computation*, (175):50–75, 2002.
13. Pietro Di Gianantonio, Furio Honsell, and Luigi Liquori. A lambda calculus of objects with self-inflicted extension. In *OOPSLA'98*, pages 166–178, 1998.
14. Stephan Hermann. Object Teams: Improving modularity for crosscutting collaborations. In *Net Object Days 2002*, volume 2591 of *LNCS*, 2002.
15. Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.
16. Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
17. Tetsuo Kamina and Tetsuo Tamai. McJava – a design and implementation of Java with mixin-types. In *2nd ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *LNCS*, pages 398–414. Springer, 2004.
18. Gunter Kniesel. Type-safe delegation for run-time component adaptation. In *ECOOP'99*, volume 1628 of *LNCS*, pages 351–366, 1999.
19. Ole Lehrmann Madsen and Birger Moller-Pdersen. Virtual classes: A powerful mechanism in object-oriented programming. In *OOPSLA'89*, pages 397–406, 1989.
20. S. Monpratarnchai and T. Tamai. The design and implementation of a role based language, EpsilonJ. In *Proc. ECTI-CON 2008*, pages 37–40, 2008.
21. Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP 2002*, volume 2374 of *LNCS*, pages 89–110, 2002.
22. Klaus Ostermann and Mira Mezini. Object-oriented composition untangled. In *OOPSLA'01*, pages 283–299, 2001.
23. Charles Smith and Sophia Drossopoulou. *Chai*: Traits for Java-like languages. In *ECOOP 2005*, volume 3586 of *LNCS*, pages 453–478, 2005.
24. T. Tamai. The language specification of EpsilonJ. http://www.graco.c.u-tokyo.ac.jp/t̃amai/pub/epsilon/spec.txt.
25. Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. An adaptive object model with dynamic role binding. In *International Conference on Software Engineering (ICSE 2005)*, pages 166–175, 2005.
26. Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. Objects as actors assuming roles in the environment. In *Software Engineering for Multi-Agent Systems V*, volume 4408 of *LNCS*, pages 185–203, 2007.
27. D. Thomas and A. Hunt. *Programming Ruby: A Pragmatic Programmer's Guide*. Addison-Wesley, 2000.