A Polymorphic Calculus for Views and Object Sharing

Atsushi Ohori

Kyoto University

Keishi Tajima

University of Tokyo
&
Kyoto University

$\boxed{\text{Backgrounds (1/2)}}$

IS-A Relation in Current OODBs

IS-A relation in current OODBs has two roles.
e.g. $Employee$ *IS-A* $Person$:

- method inheritance — a method on $Person$ is also applicable to $Employee$ objects, and
- extent inclusion — an instance of $Employee$ is also included in a extent of $Person$.

These two seem to be ad-hoc approximations to more fundamental mechanisms of method and object sharing!

Backgrounds (2/2)

Method Sharing by Polymorphic Type Inference

Polymorphic type inference for records can infer the most general applicability of the method.

e.g. in **Machiavelli**[SIGMOD89]:

```
fun wealthy S = select x·Name from x ∈ S where x·Salary > 100
    : {'a::[Name:'b, Salary:int]} →{'b}
```

This provides us a more precise framework for method sharing.

$\boxed{\text{Motivation}}$

Object Sharing by General Predicate

Simple partial ordering is inadequate. e.g.:

"define a new class *Foreigner* from *Student* and *Employee*"

A more direct and natural approach is:

to allow the programmer to specify desired object sharing predicates between arbitrary classes.

$\boxed{\text{The Goal of This Work}}$

We develop a framework for

- object sharing by general sharing predicate,
- objects with views,

  — to be shared by multiple classes, an object should have multiple views —

and

- integrate them into a polymorphic type inference system for method sharing.

Features of Our Language

1. Classes with Object Sharing

   - Class definitions include sharing specifications.

     class S include $Student$ as f_1 where $age > 20$
     include $Employee$ as f_n where $age > 20$ end

   - Recursive class can be defined.

     Sharing relation can be cyclic.

2. Objects with Views

   - Views are defined by mapping functions

     joe $=$ [Name $=$ "joe", Sex $=$ "male", BirthY $=$ 1968]

     joe as [Name $= x\cdot$Name, Age $=$ 1994 $- x\cdot$BirthY]

   - Uniform treatment for objects and views

Our Strategy (1/2)

An “object” is internally an association of

- a *raw object* — a record that has an identity,
- a *viewing function* – to map the raw object to its view.

A “class” is internally an association of

- an *immediate extent* — a set of immediate instances,
- a *sharing predicate* — a function evaluated dynamically.

Our Strategy (2/2)

To integrate "objects" with views and "classes" with object sharing into a polymorphic type system, we

1. define the polymorphic core calculus similar to that of **Machiavelli**,
2. extend the core to "objects", and then to "classes",
3. define the semantics of the extended language by giving a translation to the core, and
4. show the translation preserves typing.

$\boxed{\text{The Core Calculus}}$

$$
\begin{aligned}
e ::= {} & c^{\tau} \mid \mathsf{()} \mid x \mid \mathsf{eq}(e,\, e) \mid \lambda x.e \mid (e\ e) \mid \ \mathsf{fix}\ x.e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e\ \mathsf{end} \mid \\
& [f, \ldots, f] \mid e{\cdot}l \mid \mathsf{extract}(e,\, l) \mid \mathsf{update}(e,\, l,\, e) \mid \\
& \{e, \ldots, e\} \mid \mathsf{union}(e,\, e) \mid \mathsf{hom}(e,\, e,\, e,\, e)
\end{aligned}
$$

- f in $[f, \ldots, f]$ is one of:
  - l := e : for mutable fields
  - l = e : for immutable fields
- a record has an identity.
- $\mathsf{extract}(r,\, l)$ extracts the L-value of a mutable l field.

The type system can infer principal type. e.g.

fun wealthy S = select x·Name from x ∈ S where x·Salary > 100

: $\forall a{::}U.\forall b{::}[\mathsf{Name}{:}a,\ \mathsf{Salary}{:}\mathsf{int}].\{b\}\rightarrow\{a\}$

Extension for Objects with Views (1/3)

Expressions, Types, and Typing Rules

$$e ::= \cdots \mid \mathsf{IDView}(e) \mid (e\ \mathsf{as}\ e) \mid \mathsf{query}(e,\ e) \mid \mathsf{fuse}(e, e) \mid \mathsf{relobj}(l_1 = e_1, \ldots, l_n = e_n)$$

$$\tau ::= \cdots \mid obj(\tau)$$

(object creation) $$\frac{\mathcal{K}, \mathcal{A} \rhd e : \tau \quad \mathcal{K} \vdash \tau :: [\![\,]\!]}{\mathcal{K}, \mathcal{A} \rhd \mathsf{IDView}(e) : obj(\tau)}$$

(view composition) $$\frac{\mathcal{K}, \mathcal{A} \rhd e_1 : obj(\tau_1) \quad \mathcal{K}, \mathcal{A} \rhd e_2 : \tau_1 \to \tau_2}{\mathcal{K}, \mathcal{A} \rhd (e_1\ \mathsf{as}\ e_2) : obj(\tau_2)}$$

(query through view) $$\frac{\mathcal{K}, \mathcal{A} \rhd e_1 : \tau_1 \to \tau_2 \quad \mathcal{K}, \mathcal{A} \rhd e_2 : obj(\tau_1)}{\mathcal{K}, \mathcal{A} \rhd \mathsf{query}(e_1, e_2) : \tau_2}$$

$\boxed{\text{Extension for Objects with Views (2/3)}}$

Examples of Views

Attribute renaming, hiding, derivation and access control

```
val joe = IDView([Name = "Doe",
                  BirthY = 1955,
                  Salary:= 2000,
                  Bonus := 5000])
  : obj([Name = str, BirthY = int, Salary := int, Bonus := int])
```

```
val joeview = (joe as λx.[Name = x·Name,
                          Age = ThisYear - x·BirthY,
                          Income = x·Salary,
                          Bonus := extract(x, Bonus)])
  : obj([Name = str, Age = int, Income = int, Bonus := int])
```

Extension for Objects with Views (3/3)

Examples of Views (cont.)

Query on "objects" with views:

fun annual_income p = (p·Income) $*$ 12 $+$ p·Bonus

  : $\forall a{::}[\![\text{Income} = int,\ \text{Bonus} = int]\!].a \to int$

query(annual_income, joeview)

  : int

View update:

fun adjust_bonus p = update(p, Bonus, p·Income $*$ 3)

  : $\forall a{::}[\![\text{Income} = int,\ \text{Bonus} := int]\!].a \to ()$

query(adjust_bonus, joeview)

  : unit

$\boxed{\text{Classes and Object Sharing (1/2)}}$

Expressions, Types, and Typing Rules

$$
\begin{array}{rl}
e ::= \cdots & \mid \textsf{class}\ S\ \ \textsf{include}\ C_1^1, \ldots, C_1^{m_1}\ \textsf{as}\ e_1\ \textsf{where}\ p_1 \cdots \\
& \qquad\qquad \textsf{include}\ C_n^1, \ldots, C_n^{m_n}\ \textsf{as}\ e_n\ \textsf{where}\ p_n\ \textsf{end} \\
& \mid \textsf{c-query}(e,\ e) \mid \textsf{insert}(e,\ e) \mid \textsf{delete}(e,\ e) \\
\tau ::= \cdots & \mid class(\tau)
\end{array}
$$

$$
\text{(cquery)}\ \frac{\mathcal{K}, \mathcal{A} \rhd e\ :\ \{obj(\tau_1)\} \rightarrow \tau_2 \quad \mathcal{K}, \mathcal{A} \rhd C\ :\ class(\tau_1)}{\mathcal{K}, \mathcal{A} \rhd \textsf{c-query}(e, C)\ :\ \tau_2}
$$

Classes and Object Sharing (2/2)

Examples of Classes

Staff : $class$([Name = str, Sex = str, Salary := int])

Student : $class$([Name = str, Sex = str, Degree := int])

```
let FemaleMember = class {}
    include Staff
         as  λs.[Name = s·Name, Category = "staff"]
      where  λs.(s·Sex = "female")
    include Student
         as  λs.[Name = s·Name, Category = "student"]
      where  λs.(s·Sex = "female") end
: class([Name = str, Category = str])
```

$\boxed{\text{Extension for Recursive Definition}}$

We introduce a new construct for recursive definition.

```
let Staff = class {} include FemaleMember as ···
               where ··· (select only staffs)
and Student = class {} include FemaleMember as ···
               where ··· (select only students)
and FemaleMember = class {}
        includes Staff as ···
          where ··· (select only females)
        include Student as ···
          where ··· (select only females)
```

Semantics of Recursive Classes

The semantics of recursive definitions can be defined as minimal solution satisfying constraints.
This can be computed by simple searching with cycle avoidance.
Intuitively, the points are

- cycle of definition occur only in from clauses.
- include clause never produces new identity.
- elements of sets of objects are collapsed using fuse(o_1,o_2)

Semantics of the Extended Language

We define the precise semantics by giving a systematic translation:

$$\mathbf{tr}(\textsf{IDView}(e)) = (e,\ \lambda\textsf{x}.\textsf{x})$$

$$\mathbf{tr}((e_1 \textsf{ as } e_2)) = (\mathbf{tr}(e_1)\cdot 1,\ \lambda\textsf{x}.(e_2\ (\mathbf{tr}(e_1)\cdot 2\ \textsf{x})))$$

$$\mathbf{tr}(\textsf{query}(e_1, e_2)) = (e_1\ (\mathbf{tr}(e_2)\cdot 2\ \mathbf{tr}(e_2)\cdot 1))$$

$$\mathbf{tr}(\textsf{class } S \textsf{ include } C_1^1, \ldots, C_1^{m_1} \textsf{ as } e_1 \textsf{ where } p_1 \textsf{ include } \cdots \textsf{ end})$$

$$= [\textsf{OwnExt}{:=}S,\ \textsf{Ext}{=}\lambda().\textsf{union}(S,\ \textsf{union}(\textsf{select}\ \cdots))]$$

$$\mathbf{tr}(\textsf{c-query}(e, C)) = (\mathbf{tr}(e)\ (\mathbf{tr}(C)\cdot\textsf{Ext}\ ()))$$

$$\vdots$$

Properties of the Extended Language

The translation preserves typing. By this,

Proposition 1 *The type system of the extended language is sound with respect to the semantics given by the translation.*

Furthermore,

Proposition 2 *The type system of the extended language can still infer the principal type for any type consistent expression.*

Conclusions

We have

- developed a framework for object with views and class with object sharing
- given a precise semantics for them
- integrated those framework with a polymorphic type system

Some further investigations:

- integrate the language with parametric classes with multiple inheritance for object oriented programming
- abstract characterization of views and classes.