A Polymorphic Calculus for Views and Object Sharing[*]

Atsushi Ohori
Research Institute for Mathematical Sciences
Kyoto University
Sakyo-ku, Kyoto 606-01, JAPAN
E-mail: ohori@kurims.kyoto-u.ac.jp

Keishi Tajima
Department of Information Science
University of Tokyo
Hongo, Bunkyo-ku, Tokyo 113, JAPAN
E-mail: tajima@is.s.u-tokyo.ac.jp

Abstract

We present a typed polymorphic calculus that supports a general mechanism for view definition and object sharing among classes. In this calculus, a class can contain inclusion specifications of objects from other classes. Each such specification consists of a *predicate* determining the subset of objects to be included and a *viewing function* under which those included objects are manipulated. Both predicates and viewing functions can be any type consistent programs definable in the polymorphic calculus. Inclusion specifications among classes can be cyclic, allowing mutually recursive class definitions. These features achieve flexible view definitions and wide range of class organizations in a compact and elegant way. Moreover, the calculus provide a suitable set of operations for views and classes so that the programmer can manipulate views and classes just the same way as one deals with ordinary records and sets.

The proposed calculus uniformly integrates views and classes in a polymorphic type system of a database programming language similar to Machiavelli. The calculus has a *type inference algorithm* that relieves the programmer from complicated type declarations of views and classes. The polymorphic type system of the calculus is also shown to be sound, which guarantees a complete static check of type consistency of programs involving classes and views. Through these properties, the programmer can enjoy full advantages of polymorphism and type inference when writing object-oriented database programs.

1 Introduction

Inheritance is a term that implies hierarchically organizing objects through a partial ordering on classes usually called an *IS-A* relation. As discussed in [5, 8, 10], this concept appears to be used for two different purposes in object-oriented databases. As in object-oriented programming languages in Smalltalk [13] tradition, it is used for *code sharing*; by asserting that Employee *IS-A* Person, we expect that any method applicable to Person objects is also applicable to Employee objects. In object-oriented databases, inheritance is also used for maintaining *extent inclusion*; by the above assertion we usually assume that the set of Employee objects is a subset of the set of Person objects. Several object-oriented database systems and languages have been proposed by using these two forms of inheritance [7, 4]. While IS-A relation provides a compact and intuitively appearing way of organizing objects and methods, there seems to be no a priori reason why these two mechanisms should be controlled by simple partial orderings.

For code sharing, IS-A relation achieves some flexibility of method application by allowing an object of one type to have its supertypes. This usage of IS-A relation can be regarded as an "approximation" to polymorphic typing in programming with records. Recent years have seen that *polymorphic type inference* for records provides a better alternative for method sharing [29, 17, 20, 24]. In this paradigm, the fact that Employee can inherit a method defined on Person corresponds the property that the method is *polymorphic* so that it can also be applicable to elements of Employee. Since type inference directly captures the polymorphic nature of a method through inferring the principal type of the method, it achieves more general and rigorous model for method sharing. The IS-A relation is then regarded as a special case of method applicability represented by polymorphic typing for methods. As demonstrated by Machiavelli [22, 9], this paradigm can be successfully applied to database programming for various forms of complex objects.

We believe that a similar situation exists for object sharing. Extent inclusion through IS-A relation is certainly useful for many cases. However, there are others where a simple partial ordering is inadequate for expressing the desired structure of object sharing. For example, suppose we have classes Student and Employee and want to define a new class FemaleMember. We

[*]This is an authors's version of the paper appeared in **ACM Symposium on Principles of Database Systems, pages 99–112, 1993.**

naturally want the class FemaleMember to share Student objects and Employee objects that are considered as female. Such a situation cannot be directly modeled by a simple partial ordering. There are some proposals [28, 26, 1] that attempt to overcome this limitation by dynamically generating appropriate classes and then placing them in a class hierarchy. However, this process generally requires the generation of intermediate classes to conform to the presupposed model of extent inclusion based on a partial ordering. A more direct and natural approach is to develop a framework for object sharing among classes where the programmer can specify desired object sharing relations with an arbitrary condition between arbitrary two classes. Extent inclusion, where one class unconditionally shares the entire extent of another class, then becomes a special case of object sharing. The motivation of our work is to develop such a framework. A preliminary idea on generalized extent inclusion is described in [27].

In this paper, we present a typed polymorphic calculus for view definitions and object sharing. We allow a class definition to contain inclusion specifications of objects from other classes. Each such specification consists of a *predicate* that selects the subset of objects to be included and a *viewing function* under which the included objects are manipulated. Both predicates and viewing functions can be any type consistent programs definable in the polymorphic calculus. This achieves flexible sharing among classes. Moreover, the dependence structure among classes induced by the inclusion specifications can be cyclic, and therefore *mutually recursive class definitions* are possible.

The first step in our development is to define a flexible mechanism for views of individual objects. The ability for a class to share objects with another class necessarily implies the ability to view an object differently depending on the context in which the object is manipulated. For example, if FemaleMember class imports some Employee object, then the imported Employee object should behave like a FemaleMember object when evaluating a query on FemaleMember class. To develop a proper view mechanism, we regard an object as an association of a *raw object* and a *viewing function*. A raw object is usually implemented by a record, and serves as a carrier of identity and attribute information. A viewing function specifies how the attribute information should be presented to the user. Since we believe that it is this data structure that properly represents the notion of "objects" in object-oriented databases, we use the term objects for such associations of a raw object and a viewing function.

The idea of representing views through functions is not new. In the relational model, a view is essentially an expression specifying how the relations be transformed. Gottlob, Paolini and Zicari [14] developed a theory of relational view update based on the notion of *dynamic views*, which is regarded as an association of a data object and a viewing function. In the context of object-oriented databases, Heiler and Zdonik [15] uses a similar notion to provide a viewing mechanism. "View object" in [30, 11] can also be regarded as an unevaluated viewing function. The general idea of views is also related to "roles" of objects [25, 3]. These proposals contain a mechanism to attach multiple roles to objects, which can be regarded as a simple method to implement some aspect of views. However, these existing approaches in object-oriented databases only describe some desired features of views operationally; there seems to be no formal framework for systematic manipulation of objects and views with a rigorous semantics. Here we attempt to provide such a framework within a paradigm of type theory of programming languages. We hope that the framework presented here will serve as a typed foundation for statically typed polymorphic object-oriented database programming languages.

Views and object sharing have also been considered in deductive approaches, where views are represented by a form of deductive rules. Based on this general idea, in [16, 2], a semantically sound accounts for views are given. These approaches seems to depends crucially on the properties of the underlying framework, i.e. formal logic. It is not at all clear how the notions of views and objects in deductive framework are related to those in database programming languages. Precise comparison of our approach with those in deductive databases is certainly interesting, but it may require a more abstract characterization of views and objects. We will comment more on this in conclusion.

In order to define a programming language for uniform manipulation of views and objects, we should define a proper set of operations that allow the programmer to manipulate objects just the same way as one manipulates ordinary records representing raw objects. This uniformity is essential for extending the language for objects to classes, which correspond to collections of those objects that have the same view type but may have different types of raw objects. In this paper, we give such a set of operations and define their precise semantics and an effective implementation algorithm by developing a systematic method to translate the operations into a polymorphic calculus with records.

We then develop a mechanism for classes with general sharing relations on top of the language for objects. A class is conceptually represented as a pair consisting of a set of objects representing its *own extent* and an *inclusion function* that computes a set of objects included from other classes. Similar to what we do for objects, we define a set of operations on those classes in such a way that the programmer can manipulate classes just the same way as one manipulate sets of objects. We then define their precise semantics and an effective implementation algorithm by developing a systematic method to translate the operations into the language we have defined for objects.

We carry out this development in a formal framework of a polymorphic record calculus developed in [23]. We show that the two translations we described preserve typings, which establishes that the language with views and classes can be statically type-checked. Moreover, there is an algorithm to infer a principal type for any type consistent program involving views and classes.

One issue we have not addressed in this paper is a proper treatment of persistent data, which require some form of dynamic typing. Connor *et. al.* [12] demonstrated that various features of views of persistent data can be represented by combining a form of localized dynamic typing and existential types. Their techniques seems to be complemental to our method.

The rest of the paper is organized as follows. Section 2 defines the core language for records and sets. Section 3 extends the core to objects. Section 4 extends the language to classes. Section 5 discusses some further issues and concludes this paper.

Page limitation makes it difficult to present the calculus and its various properties fully; the authors intend to present a more detailed description in another paper.

2 The Core Language

This section defines a polymorphic calculus similar to Machiavelli. This serves as the core language on which view and object sharing mechanisms will be developed later. We first explain two important data structures: record and set.

The syntax for records is:

$$[f, \ldots, f]$$

where f denotes *fields* whose syntax is either $l = e$ for immutable fields or $l := e$ for mutable fields. For example,

$$\mathsf{joe} = [\mathsf{Name} = \text{"}\mathsf{Doe}\text{"}, \mathsf{Salary} := \mathsf{3000}]$$

will yield a record with immutable Name field and mutable Salary field. Records are used to implement raw objects as seen in the next section. To properly capture identity of raw objects, we decide that evaluation of a record expression creates a new identity (internally implemented by a reference) and that equality on records is identity, i.e. L-value equality.

There are three operations on records. The first is field extraction: $r{\cdot}l$, which extracts the value of l field from the record r. The extracted value is always an ordinary value, i.e. R-value even if the l field in r is mutable. The second is a special form of field extraction: $\mathsf{extract}(r, l)$, which extracts the L-value of the mutable field l of the record r. The extracted L-values can only be used as field values in a record. For example,

$$\begin{array}{l}\mathsf{Doe} = [\mathsf{Name} = \text{"}\mathsf{Doe}\text{"}, \mathsf{Income} := \mathsf{extract}(\mathsf{joe}, \mathsf{Salary})] \\ \mathsf{john} = [\mathsf{Name} = \text{"}\mathsf{John}\text{"}, \mathsf{Salary} = \mathsf{extract}(\mathsf{joe}, \mathsf{Salary})]\end{array}$$

are legal, resulting in joe's Salary field, Doe's Income filed, and john's Salary field all sharing the same L-value. If one changes joe's Salary field, then that change will be reflected to Doe and also to john even though john's Salary field is immutable. However, both of the following are illegal and will be rejected by the type system we shall define later:

$$\begin{array}{l}[\mathsf{Name} = \text{"}\mathsf{Joe\ Doe}\text{"}, \mathsf{Income} = \mathsf{extract}(\mathsf{joe}, \mathsf{Salary}) * 2] \\ [\mathsf{Name} = \mathsf{extract}(\mathsf{joe}, \mathsf{Name}), \mathsf{Income} := \mathsf{joe}{\cdot}\mathsf{Salary}]\end{array}$$

The first try to perform arithmetic operation on an extracted L-value and the second attempts to extracts the L-value of an immutable field. Distinguishing these two forms of field extraction allow us to properly transfer mutability of fields to views. The third operation on records is field update: $\mathsf{update}(r, l, e)$, which changes the value of the mutable l field of the record r to e. For example, update(joe, Salary, 4000) will change joe' s Salary value to 4000, while update(joe, Name, "Peter") is illegal and is rejected by the type system, since joe's Name filed is immutable.

Another data structure we consider here is set, whose syntax is:

$$\{e_1, \ldots, e_n\}$$

The basic operations for sets are: $\mathsf{union}(e, e)$ and $\mathsf{hom}(S,\ f,\ op,\ z)$. hom [22] is a general iteration operation similar to "pump" in FAD[6]. Its intuitive meaning can be explained by the equation:

$$\begin{array}{l}\mathsf{hom}(\{e_1, \ldots, e_n\},\ f,\ op,\ z) \\ \quad = op(f(e_1), op(f(e_2), \ldots, op(f(e_n), z) \ldots))\end{array}$$

There are other possibilities for a general elimination operation for sets. We believe that adoption of a different elimination operation for sets will not affect the mechanisms for views and classes we shall develop in this paper. The following operations are definable using union and hom: $\mathsf{member}(e_1,e_2)$ which tests if e_1 is a member of e_2, $\mathsf{prod}(e_1,\ldots,e_n)$ which computes n-ary products of sets $e_1, \ldots, e_2$, $\mathsf{map}(e_1,e_2)$ which maps a function e_1 over a set e_2, and $\mathsf{filter}(e_1,e_2)$ which selects all the elements x in a set e_2 such that $(e_1\ x) = \mathsf{true}$. These operations will be used later.

By integrating records and sets in a lambda calculus, we define the syntax of the core language:

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

c^τ stands for constants of type τ. () is the only value of type *unit* that can be returned by functions such as update. x stands for variables of the calculus. $\mathsf{eq}(e_1, e_2)$ is equality test. For records and functions, eq uses L-value equality; for other types, it uses the usual value equality. $\lambda x.e$ stands for lambda abstraction, and $(e\ e)$ for function application. $\mathsf{fix}\ x.e$ is for recursive function definition where x can occur free in e, and $\mathsf{let}\ x = e\ \mathsf{in}\ e\ \mathsf{end}$ is ML's polymorphic let construction. By combining fix, let, lambda abstraction, and record, it is possible to define the following mutually recursive function definition:

$$\mathsf{fun}\ f_1\ x_1 = e_1\ \mathsf{and} \cdots \mathsf{and}\ f_n\ x_n = e_n$$

where the functions $f_1, \ldots, f_n$ being defined may be used in the bodies $e_1, \ldots, e_n$ of those function definitions. We use pairs (e_1, e_2) as an abbreviation for two element records with numeric labels, and the projections $e{\cdot}1$ and $e{\cdot}2$ for the corresponding field extractions. Accordingly, we write $\tau_1 \times \tau_2$ for $[1 = \tau_1, 2 = \tau_2]$. We also write $\lambda().e$ for a function whose domain type is *unit*.

The type system of the language is an adaptation of that of [23] with a refinement for distinguishing immutable and mutable record fields. The set of monotypes (ranged over by τ) of the language is given by the syntax:

$$\tau ::= b \mid unit \mid t \mid \tau \to \tau \mid \{\tau\} \mid \mathcal{L}(\tau) \mid [F, \ldots, F]$$

b stands for base types such as *string*, t for type variables, $\tau \to \tau$ for function types, $\{\tau\}$ for set types whose element type is τ. $\mathcal{L}(\tau)$ is a type of L-values of a field of type τ in a record. $[F, \ldots, F]$ stands for record types where F is either $l = \tau$ for immutable fields or $l := \tau$ for mutable fields.

To represent polymorphic types for operations on records, we place *kind constraint* on type variables. The set of kinds (ranged over by K) is given by the grammar:

$$K ::= U \mid [\![F, \ldots, F]\!]$$

U denotes arbitrary types, while $[\![F_1, \ldots, F_n]\!]$ denotes those record types that contain the fields $F'_1, \ldots, F'_n$ and possibly others, where each F_i must satisfy the following condition: if F_i is $l := \tau$ then F'_i must be $l := \tau$, on the other hand if F_i is $l = \tau$ then F'_i can be either $l = \tau$ or $l := \tau$. We write $F_i < F'_i$ if for this condition. Using kinds, the set of polytypes is defined by the syntax:

$$\sigma ::= \tau \mid \forall t :: K.\sigma$$

$\forall t :: K.\sigma$ is a polymorphic type where t is quantified over the subset of types denoted by the kind K.

The type system of this language has two forms of judgements. A kinding judgement $\mathcal{K} \vdash \tau :: K$ asserts that the type τ has the kind K under the kind assignment $\mathcal{K}$, which is a mapping from type variables to kinds. A typing judgements $\mathcal{K}, \mathcal{A} \rhd e : \sigma$ asserts that the expression e has the type σ under $\mathcal{K}$, and a type assignment $\mathcal{A}$, which is a mapping from variables to types. Figure 1 shows some of the rules to derive these two forms of judgements. Others are the same as in [23].

Since ML-style polymorphic typing is not sound with respect to the usual operational semantics of mutable values [19], we place the restriction that the type of mutable fields be ground monotypes. With this restriction, we can show that the type system is sound with respect to an operational semantics of the language in the style of [23], and that "well typed programs cannot go wrong" as shown by Milner [19] for ML. Moreover, there is an algorithm to compute a principal type of any type consistent program. The following properties can be shown by the techniques developed in [23].

Proposition 1 *If $\mathcal{K}, \mathcal{A} \rhd e : \tau$ and e evaluates to v under an environment that respect the type assignment and the kind assignment $\mathcal{K}, \mathcal{A}$, then v has the type τ.*

Proposition 2 *For any e and $\mathcal{K}, \mathcal{A}$, if e has a typing under $\mathcal{K}$ and $\mathcal{A}$, then e has a principal type under $\mathcal{K}, \mathcal{A}$ such that any other type under $\mathcal{K}, \mathcal{A}$ is its instance. Moreover, a principal type is computed effectively.*

3 View Extension

As explained in the introduction, we regard an object as a combination of a raw object and a viewing function. This section extends the core with a mechanism for manipulating those objects.

3.1 An algebra for objects and views

The set of expressions is extended with the following set expression constructors, which serves as an "algebra" for objects and views:

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

The intended meanings of these constructors are explained below. Their precise semantics is given later by defining a translation of them into the core language.

Object creation: **IDView(e).** This turns a raw object e into an object with the *identity view.*

View composition: **(e_1 as e_2).** Given an object e_1 and a function e_2, this creates a new object whose raw object is the same as that of e_1 and whose viewing function is the composition of the viewing function of e_1 and the function e_2. By this construction, we obtain the desired closure property of objects and view definition; defining a view on an object yields another object having the same identity, which has the same formal status as being an object, and therefore yet another views can freely be defined on it.

$$\mathcal{K} \vdash \tau :: U \quad \text{for all } \tau$$

$$\mathcal{K} \vdash t :: [\![F_1, \ldots, F_n]\!] \quad \text{if } t \in dom(\mathcal{K}), \mathcal{K}(t) = [\![F'_1, \ldots, F'_n, \ldots]\!] \quad \text{such that} \quad F_i < F'_i (1 \leq i \leq n)$$

$$\mathcal{K} \vdash [F'_1, \ldots, F'_n, \ldots] :: [\![F_1, \ldots, F_n]\!] \quad \text{such that} \quad F_i < F'_i (1 \leq i \leq n)$$

$$\text{(rec)} \quad \frac{\mathcal{K}, \mathcal{A} \rhd e_1 : \tau'_1, \ \ldots\ , \mathcal{K}, \mathcal{A} \rhd e_n : \tau'_n}{\mathcal{K}, \mathcal{A} \rhd [l_1 @_1 e_1, \ldots, l_n @_n e_n] : [l_1 @_1 \tau_1, \ldots, l_n @_n \tau_n]} \quad \text{where } @_i \text{ is } = \text{ or } :=, \text{ and } \tau'_i \text{ is either } \tau_i \text{ or } \mathcal{L}(\tau_i)$$

$$\text{(dot)} \quad \frac{\mathcal{K}, \mathcal{A} \rhd e : \tau_1 \qquad \mathcal{K} \vdash \tau_1 :: [\![l = \tau_2]\!]}{\mathcal{K}, \mathcal{A} \rhd e \cdot l : \tau_2} \qquad \text{(ext)} \quad \frac{\mathcal{K}, \mathcal{A} \rhd e : \tau_1 \qquad \mathcal{K} \vdash \tau_1 :: [\![l := \tau_2]\!]}{\mathcal{K}, \mathcal{A} \rhd \mathsf{extract}(e,\, l) : \mathcal{L}(\tau_2)}$$

$$\text{(upd)} \quad \frac{\mathcal{K}, \mathcal{A} \rhd e_1 : \tau_1 \qquad \mathcal{K}, \mathcal{A} \rhd e_2 : \tau_2 \qquad \mathcal{K} \vdash \tau_1 :: [\![l := \tau_2]\!]}{\mathcal{K}, \mathcal{A} \rhd \mathsf{update}(e_1,\, l,\, e_2) : unit}$$

$$\text{(gen)} \quad \frac{\mathcal{K}\{t \mapsto k\}, \mathcal{A} \rhd e : \sigma}{\mathcal{K}, \mathcal{A} \rhd e : \forall t :: k.\sigma} \quad t \text{ not free in } \mathcal{A} \qquad \text{(inst)} \quad \frac{\mathcal{K}, \mathcal{A} \rhd e : \forall t :: k.\sigma \quad \mathcal{K} \vdash \tau :: k}{\mathcal{K}, \mathcal{A} \rhd e : \sigma[\tau/t]}$$

Figure 1: Kinding rules and some of typing rules

We do not usually regard a function that changes the state of an object as a viewing function. So it would be useful for the type system to check whether e_2 in this construct changes the state of the raw object or not. It seems to be not hard to refine the type system to check whether e_2 involves update operations (directly or indirectly through other functions called from e_2). However, this significantly increases the complexity of the type system, and is not dealt with here. Also, there might be a case where a function with side effect can be considered as a suitable viewing function.

Query on views: query(e_1, e_2). This evaluates a query specified by a function e_1 against an object e_2. It is this operation that actually evaluates a viewing function associated with a raw object. Intuitively, this expression first evaluates or "materializes" the view by applying the viewing function of e_2 to the raw object of e_2 and then applies the function e_1 to the result of the application. As a special case, if e_1 is the identity function, then this construct simply returns the current value under the view.

Generalized equality on objects: fuse(e_1,e_2). This first tests whether e_1 and e_2 have the same raw object. If they do then return a singleton set of a new object whose raw object is the same as that of argument objects and whose view encode the views of both e_1 and e_2 in a product type. If e_1 and e_2 have different raw objects then the result is the empty set. This operation is inspired by a similar operation considered in [9], and is regarded as a generalization of equality test for objects.

Relation object creation: relobj(l_1=e_1,. . .,l_n=e_n). This creates a relation object from the given set of objects. Different from all the previous operations, this creates a new raw object, and therefore new identity. The new raw object is a record whose l_i field is the raw object of e_i. Its viewing function is composed of the viewing functions of $e_i, \ldots, e_n$, and corresponds to the relation of their views. Combining with sets, this construction can be used to represent relation objects between sets of objects.

Although we have not yet developed a formal basis to discuss expressive power of languages for objects and views, we believe that the operations just defined form a sufficient set of operations for manipulating objects and views; any other operations for objects and views seem to be definable using these together with lambda abstraction. Some useful operations for objects and views definable using these four operations include:

- Equality test for objects: objeq(e_1, e_2). This tests whether two objects has the same raw object and therefore have the same identity. This is implemented as

  not(eq(fuse(e_1,e_2), {}))

- Select view mapping: (select as e from S where p). This selects those objects that satisfies p from S with a new viewing function e. This is implemented as:

  map(λx.(x as e), filter(p,S))

- Intersection:intersect(e_1, e_2) Given sets e_1 and e_2 of objects, this returns a new sets that corresponds to the intersection of the two. The type of objects of the resulting set is the product type of the types of objects of the two sets. This is implemented as:

  hom(prod(e_1,e_2), λx.fuse(x·1,x·2), union, {})

  It is easy to generalize this to n-ary intersection intersect(e_1,. . .,e_n), which will be used in the following development.

- Relation style query:

  relation $[l_1{=}e_1,\ldots,l_n{=}e_n]$
  from $x_1 \in S_1,\ldots,x_m \in S_m$
  where P

  This creates a set of relation objects satisfying the predicate P from the sets S_i of objects. This can be implemented by lifting the techniques we used in Machiavelli [9] to objects and views using relation object creation. One implementation is the following:

```
map(λx.x·1,
   filter(λy.y·2,
      map(λX. (relobj(l1=X·1,...,ln=X·n),P),
         prod(S1,S2,...,Sn))))
```

The last three involve sets of objects. Before we proceed, there is one decision to be made on the semantics of sets of objects. Since there are two form of equality on objects, we need to determine which equality is used in formation of a set of objects. Here we choose objeq. This decision yields a simpler formulation of recursive classes and views we shall develop later.

Choosing objeq implies that we need to "collapse" elements that are objeq whenever we make a union of two sets. Conceptually, there would be two alternatives. One is to require that if a union of two sets contains two objects that are objeq, then they must also have the same viewing function, i.e. the function with the same L-value. The other alternative is to select one object among those that have the same identity. Here we choose the latter approach and assume that if $e_1 \in S_1$ and $e_2 \in S_2$, then $S_1 \cup S_2$ will choose e_1 and discard e_2. This will yield a more flexible mechanism for classes and object sharing. However, the other alternative is equally possible. The following development can easily be adopted for the other semantics of sets of objects.

3.2 Typing and semantics of objects and views

One important feature of our approach is that such viewing mechanisms are typed operations in our polymorphic type system. We introduce a new type constructor $obj(\tau)$ for objects of type τ. As seen below, this type is internally implemented as a type of the form $\tau' \times (\tau' {\rightarrow} \tau)$ where τ' is the type of the raw object which is hidden from the programmer manipulating objects. The typing rules for the new expression constructors are shown in Figure 2.

The semantics for these new constructors for objects is given by the systematic translation for these new expression constructors into the core language. Figure 3 show a set of rules recursively eliminate the new constructors we have introduced in this section.

The view extension preserves the type soundness. We establish this by showing that the translation $\mathbf{tr}(e)$ preserves typing. Let τ be a type of the extended language and τ' be a type of the core language. We say that τ' is an *internal representation* of τ if τ' is obtained by repeatedly substituting any component type of τ of the form $obj(\tau'')$ with a type of the form $\tau_1 \times (\tau_1 {\rightarrow} \tau'')$ for some type τ_1. We then have the following desirable property:

Proposition 3 *Let e be an expression possibly containing object expressions. If $\mathcal{K},\mathcal{A} \rhd e : \tau$ is derivable in the extended language then $\mathcal{K},\mathcal{A} \rhd \mathbf{tr}(e) : \tau'$ is derivable in the core language for some type τ' that is an internal representation of τ.*

Proof outline This is proved by establishing the following stronger property by induction on the typing derivations in the extended language: if $\mathcal{K},\mathcal{A} \rhd e : \tau$ is provable in the extended system then there is a type τ' of the core calculus that is determined uniquely by a derivation of $\mathcal{K},\mathcal{A} \rhd e : \tau$, τ' is an internal representation of τ, and $\mathcal{K},\mathcal{A} \rhd \mathbf{tr}(e) : \tau'$ is derivable in the core language. ∎

This result and the soundness result of the core language (Proposition 1) imply that the polymorphic type system with objects and views is sound with respect to the operational semantics defined by translating the extended language with the function $\mathbf{tr}(\_)$ and then evaluating the resulting expression in the core language. Therefore, the slogan that "a well typed program cannot go wrong" also hold for our language with objects and views. The extended language also preserves the existence of a complete type inference algorithm. This is easily seen the shape of the new typing rules.

3.3 Examples of views

The mechanism defined above allows us to represent most of the features of views discussed in literature. We demonstrate below some of them.

Basic functionalities of views: The example below includes attribute renaming, attribute hiding, computed attribute and access restriction.

```
joe = IDView([Name = "Joe", BirthYear = 1955,
              Salary:= 2000, Bonus := 5000])
      : obj([Name = string, BirthYear = int,
             Salary := int, Bonus := int])
joe_view = (joe as λx.[Name = x·Name,
                       Age = This_year() - x·BirthYear ,
                       Income = x·Salary,
                       Bonus := extract(x, Bonus)])
      : obj([Name = string, Age = int,
             Income = int, Bonus := int])
```

joe and joe_view are objects with the same raw object, and objeq(joe, joe_view) is true. joe_view renames Salary attribute to Income, hides BirthYear, added a "computed attribute" Age. It also prohibit the user from updating Salary field by making Income to be immutable.

$$\text{(id)}\quad \frac{\mathcal{K},\mathcal{A} \rhd e : \tau \qquad \mathcal{K} \vdash \tau :: [\![\,]\!]}{\mathcal{K},\mathcal{A} \rhd \mathsf{IDView}(e) : obj(\tau)} \qquad \text{(vcomp)}\quad \frac{\mathcal{K},\mathcal{A} \rhd e_1 : obj(\tau_1) \qquad \mathcal{K},\mathcal{A} \rhd e_2 : \tau_1 \to \tau_2}{\mathcal{K},\mathcal{A} \rhd (e_1 \ \mathsf{as}\ e_2) : obj(\tau_2)}$$

$$\text{(query)}\quad \frac{\mathcal{K},\mathcal{A} \rhd e_1 : \tau_1 \to \tau_2 \qquad \mathcal{K},\mathcal{A} \rhd e_2 : obj(\tau_1)}{\mathcal{K},\mathcal{A} \rhd \mathsf{query}(e_1, e_2) : \tau_2} \qquad \text{(fuse)}\quad \frac{\mathcal{K},\mathcal{A} \rhd e_1 : obj(\tau_1) \qquad \mathcal{K},\mathcal{A} \rhd e_2 : obj(\tau_2)}{\mathcal{K},\mathcal{A} \rhd \mathsf{fuse}(e_1,e_2) : \{obj(\tau_1 \times \tau_2)\}}$$

$$\text{(vrel)}\quad \frac{\mathcal{K},\mathcal{A} \rhd e_i : obj(\tau_i) \ (1 \le i \le n)}{\mathcal{K},\mathcal{A} \rhd \mathsf{relobj}(l_1{=}e_1,\ldots,l_n{=}e_n) : obj([l_1 = \tau_1, \ldots, l_n = \tau_n])}$$

Figure 2: Typing Rules for Objects and Views

$$\begin{aligned}
\mathbf{tr}(\mathsf{IDView}(e)) &= (\mathsf{e},\ \lambda \mathsf{x}.\mathsf{x}) \\
\mathbf{tr}((e_1\ \mathsf{as}\ e_2)) &= (\mathbf{tr}(e_1)\cdot 1,\ \lambda \mathsf{x}.(e_2\ (\mathbf{tr}(e_1)\cdot 2\ \mathsf{x}))) \\
\mathbf{tr}(\mathsf{fuse}(e_1,e_2)) &= \lambda \mathsf{x}.\mathsf{if}\ \mathsf{eq}(\mathbf{tr}(e_1)\cdot 1, \mathbf{tr}(e_2)\cdot 1)\ \mathsf{then}\ \{(\mathbf{tr}(e_1)\cdot 1,\ \lambda \mathsf{x}.((\mathbf{tr}(e_1)\cdot 2\ \mathsf{x}),\ (\mathbf{tr}(e_2)\cdot 2\ \mathsf{x})))\}\ \mathsf{else}\ \{\} \\
\mathbf{tr}(\mathsf{relobj}(l_1{=}e_1,\ldots,l_n{=}e_n)) &= ([l_1{=}\mathbf{tr}(e_1)\cdot 1,\ldots,l_n{=}\mathbf{tr}(e_n)\cdot 1],\ \lambda \mathsf{x}.[l_1{=}(\mathbf{tr}(e_1)\cdot 2\ (\mathsf{x}\cdot l_1)),\ldots,\ l_n{=}(\mathbf{tr}(e_n)\cdot 2\ (\mathsf{x}\cdot l_n))])
\end{aligned}$$

Figure 3: Semantics of Objects and Views through Transformation

Query on objects: The calculus achieves a uniform treatment of views and queries; i.e. they are essentially the same. Queries correspond to evaluation of views. For example, if we define a function

```
fun Annual_Income p =
    (p·Income) * 12 + p·Bonus
    : $\forall t::[\![\mathsf{Income} = int,\ \mathsf{Bonus} = int]\!].\ t \to int$
```

then the expression

```
query(Annual_Income, joe_view)
```

yields 29000. Moreover, query functions can be arbitrarily complex functions definable in the language.

View update: View update can be done by simply writing a query that changes some of mutable fields of a view by using an ordinary function that update record fields. For example,

```
adjustBonus =
   λp.query(λx.update(x, Bonus, x·Income * 3), p)
   : $\forall t::[\![\mathsf{Income}{=}int,\ \mathsf{Bonus}{:=}int]\!].obj(t) \to unit$
```

is used as a query that adjust an object's Bonus attribute. By applying this query as in

```
(adjustBonus joe_view)
```

Bonus field is correctly updated. After this query, query(λx.x, joe_view) will yield

```
[Name="Joe", Age=39, Income=2000, Bonus:=6000]
```

It should be noted that the renaming of attributes is transparent to programmers; he can program any update allowed by the type of a view. Note also that view evaluation is done lazily, so that an update made through one view is correctly reflected to any other views of the same raw object. So, after this, the query query(λx.x,joe) will yield

```
[Name="Joe", BirthYear=1955,
 Salary:=2000, Bonus:=6000]
```

where the change of Bonus filed through joe_view is correctly reflected.

Manipulation of sets of objects: By combining objects with views, we can write a query against a set of objects with views. The following function returns the set of objects whose Annual_Income exceeds 100000.

```
fun wealthy S =
  select as λx.[Name=x·Name,Age=x·Age]
  from S
  where λx.query(Annual_Income, x) > 100000;
  : $\forall t_1::U.\forall t_2::U.$
    $\forall t_3::[\![\mathsf{Income}{=}int,\ \mathsf{Bonus}{=}int,\ \mathsf{Name} = t_1,\ \mathsf{Age}{=}t_2]\!].$
      $\{obj(t_3)\} \to \{obj([\mathsf{Name}{=}t_1,\mathsf{Age}{=}t_2])\}$
```

which can be applied to any set of objects whose view type contains Name, Age, Income, and Bonus fields, as seen in the example:

```
Employees : $obj([\mathsf{Name}{=}string,\mathsf{Age}{=}int,\mathsf{Income}{=}int,$
                  $\mathsf{Bonus}{=}int])$
(wealthy Employees) : $obj([\mathsf{Name}{=}string,\mathsf{Age}{=}int])$
```

Creating a new relation from sets of objects are also be easily done by using relation ··· from ··· where ··· constructs we have defined. Moreover, in writing these queries, the programmer need not worry about the actual structure of raw objects; one can treat objects with views as if they are ordinary records.

4 Classes and Object Sharing

We are now going to develop a mechanism for classes with views and object sharing on top of the language we have just defined.

4.1 Class definitions and their typing

To deal with classes, the syntax of the language is extended with the following constructors:

```
e ::= ··· |
    class S include C_1^1, ..., C_1^{m_1} as e_1 where p_1 ···
            include C_n^1, ..., C_n^{m_n} as e_n where p_n end
    | c-query(e, e) | insert(e, e) | delete(e, e)
```

The intended meanings of these constructors are explained below. Their precise semantics is given later by defining a translation of them into the language for objects.

Class definition: The first component creates a class, where S is a set of objects that directly belong to the class, and each include $C_1, \ldots, C_n$ as e where p clause asserts that the class being defined includes all the objects that satisfy p from the intersection (in the sense of intersect) of the classes $C_1, \ldots, C_n$ under a new view specified by the function e. As seen in the semantic definition below, the actual inclusion does not take place until some query requiring the entire extent of the class is evaluated.

This general form of class definition combines four basic functionalities of object sharing: union through multiple include clauses, intersection, selection (where), and view composition (as). Although we believe that this syntax is general enough to represent most cases of object sharing, some other operations may also be included, and we leave this issue as a future investigation. Here we only point out one subtle issue in selecting operations in class definitions. In our model, a class definition involving an identity creating operation cannot be given a well founded semantics. This is a natural consequence of the intended semantics of a class definitions; in our model a class specifies new views and object sharing from other classes, both of them do not imply creation of new object identity. In analogy with SQL, one might think that the above class definition would be more general if one would have interpreted the set of classes $C_1, \ldots, C_n$ in an include clause as product formation. However, in our model as well as most of object-oriented data models, forming a product or more generally a record implies creation of new identity, which cannot be easily dealt with in class definition.

Class query: c-query(e_1, e_2)**.** This evaluates a query specified by a function e_1 against a class e_2, where e_1 may be any function on sets of objects whose type is the same as that of objects of the class. This construct allows the programmer to treat classes just as sets of objects.

Insertion: insert(e_1, e_2)**.** This inserts an object e_1 to a class e_2's own extent.

Deletion: delete(e_1, e_2)**.** This removes an object e_1 from a class e_2's own extent. Other semantics for delete can also be possible. We could have defined it so that if the specified element is imported from another class then it removes the element from that class, or it blocks the inclusion of the element from that class. The rationale of our choice is clarity and safety. Our semantics also appears to be more basic; the two other semantics stated above are definable by using delete$(_, _)$ under our semantics and other operations on views, sets and classes.

The extended syntax allows any mixture of classes and other expression constructors, i.e. classes can be treated as first-class values. This opens up the possibility of various powerful programming styles with classes, such as using class creating functions.

To define typing rules for classes, we introduce a new type constructor for classes:

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

denoting a class of objects having type $obj(\tau)$. The typing rules for classes and operations on classes are given in Figure 4.

4.2 An example of a class

We show a simple example of a class. Suppose Staff and Student are classes already defined. The following example defines a new class FemaleMember:

```
let FemaleMember = class {}
       includes Staff
         as λs.[Name = s·Name, Age = s·Age,
                Category = "staff"]
         where λs.query(λx.(x·Sex = "female"), s)
       includes Student
         as λs.[Name = s·Name, Age = s·Age,
                Category = "student"]
         where λs.query(λx.(x·Sex = "female"), s)
in ··· end
```

In this definition, the initial value for the immediate extent of FemaleMember is the empty set. Its own extents can be updated after the class formation by insert$(_, _)$ and delete$(_, _)$ operations. Class FemaleMember shares those objects of Staff and Student whose Sex is "female". Sex field of those shared objects are hidden and an additional field Category is set appropriately.

Below is an example of a query against the class FemaleMember:

```
let names = λs.map(λx.query(λy.y·Name, x), s) in
    c-query(names, FemaleMember)
```

This returns a set of names of all the female members.

The following is an exmaple of class definition where include claus contain multiple classes:

```
StudentStaff = class {}
  includes Staff, Student
    as λp.[Name = p·1·Name, Age = p·1·Age,
```

$$(\text{class})\quad \frac{\begin{array}{ll}\mathcal{K},\mathcal{A} \rhd \mathrm{S}:\ \{obj(\tau)\} & \mathcal{K},\mathcal{A} \rhd e_i\ :\ \tau_i^1 \times \cdots \times \tau_i^{m_i} \to \tau \\ \mathcal{K},\mathcal{A} \rhd C_i^j\ :\ class(\tau_i^j) & \mathcal{K},\mathcal{A} \rhd p_i\ :\ obj(\tau_i^1 \times \cdots \times \tau_i^{m_i}) \to bool\end{array}}{\mathcal{K},\mathcal{A} \rhd \ \mathsf{class}\ S\ \mathsf{include} \cdots \mathsf{as}\ e_1\ \mathsf{where}\ p_1 \cdots \mathsf{include} \cdots \mathsf{as}\ e_n\ \mathsf{where}\ p_n\ \ :\ class(\tau)}$$

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

$$(\text{insert})\ \frac{\mathcal{K},\mathcal{A} \rhd e_1\ :\ class(\tau_1) \qquad \mathcal{K},\mathcal{A} \rhd e_2\ :\ obj(\tau_1)}{\mathcal{K},\mathcal{A} \rhd \mathsf{insert}(e_1, e_2)\ :\ unit} \quad (\text{delete})\ \frac{\mathcal{K},\mathcal{A} \rhd e_1\ :\ class(\tau_1) \qquad \mathcal{K},\mathcal{A} \rhd e_2\ :\ obj(\tau_1)}{\mathcal{K},\mathcal{A} \rhd \mathsf{delete}(e_1, e_2)\ :\ unit}$$

Figure 4: Typing Rules for Classes

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathbf{tr}(\mathsf{class}\ S\ \mathsf{include}\ C_1^1, \ldots, C_1^{m_1}\ \mathsf{as}\ e_1\ \mathsf{where}\ p_1 \cdots \mathsf{include}\ C_n^1, \ldots, C_1^{m_n}\ \mathsf{as}\ e_n\ \mathsf{where}\ p_n\ \mathsf{end})} \\
 & = & [\mathsf{OwnExt}{:=}S, \\
 & & \ \mathsf{Ext}{=}\lambda().\mathsf{union}(S,\mathsf{union}(\mathsf{select\ as}\ \mathbf{tr}(e_1)\ \mathsf{from\ intersect}((\mathbf{tr}(C_1^1)\cdot\mathsf{Ext})(), \ldots, (\mathbf{tr}(C_1^{m_1})\cdot\mathsf{Ext})()) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{where}\ \mathbf{tr}(p_i),\mathsf{union}(\cdots)))] \\
\mathbf{tr}(\mathsf{c\text{-}query}(e, C)) & = & (\mathbf{tr}(e)\ ((\mathbf{tr}(C)\cdot\mathsf{Ext})\ ())) \\
\mathbf{tr}(\mathsf{insert}(e, C)) & = & \mathsf{update}(\mathbf{tr}(C),\ \mathsf{OwnExt},\ \mathsf{union}(\mathbf{tr}(C)\cdot\mathsf{OwnExt},\{\mathbf{tr}(e)\})) \\
\mathbf{tr}(\mathsf{delete}(e, C)) & = & \mathsf{update}(\mathbf{tr}(C),\ \mathsf{OwnExt},\ \mathsf{remove}(\mathbf{tr}(C)\cdot\mathsf{OwnExt},\{\mathbf{tr}(e)\}))
\end{array}$$

Figure 5: Semantics of Classes through Transformation

```
        Sex = p·1·Sex, Sal := extract(p·1, Salary),
        Deg := extract(p·2, Degree)]
    where λp.true
```

Intuitively, StudentStaff represents the intersection class of Staff and Student.

4.3 Semantics of classes

Classes are sets of objects that are evaluated lazily so that updates to classes propagate properly through sharing predicates. The precise semantics of classes is defined by giving a systematic translation of classes into the extended core language defined in the previous section. This also provides an effective algorithm to implement the extended language with classes.

The translation $\mathbf{tr}(\_)$ for expressions including classes is given in Figure 5. As seen in these definitions, lambda abstraction, $\lambda().\cdots$, of inclusion functions *delays* the materialization of the extent inclusion from other classes. The actual extent is created immediately before a query in $\mathsf{c\text{-}query}(e, C)$ expression. The translation of type constructor $class(\_)$ is defined as:

$$[\![class(\tau)]\!] = [\mathsf{OwnExt}{:=}\{obj(\tau)\},\ \mathsf{Ext}{=}\ unit \to \{obj(\tau)\}]$$

As seen in the translation above, function type is used only for delaying the extent creation and therefore its input type is the trivial type *unit*.

We can then prove the following property:

Proposition 4 *Let e be an expression possibly containing classes, operations on classes. If $\mathcal{K},\mathcal{A} \rhd e : \tau$ is derivable in the extended language then $\mathcal{K},\mathcal{A} \rhd \mathbf{tr}(e) : [\![\tau]\!]$ is derived in the language with objects defined in Section 3.*

Proof is by induction on the structures of expressions of the extended language. This property together with Proposition 1 and 3 establishes the soundness of the language with classes. As in the case of view extension, it is easily proved that the class extension preserves the existence of a complete type inference algorithm.

4.4 Recursive class definition

The above typing rule for class definition and its semantics do not allow cyclic sharing among various classes, which is sometimes useful. To support cyclic sharing, we add the following cyclic class definition:

```
let $c_1$= class $S_1$
        include ${}_1C_1^1, \ldots, {}_{l_1^1}C_1^1$ as $e_1^1$ where $p_1^1$ ···
        include ${}_1C_1^{m_1}, \ldots, {}_{l_1^{m_1}}C_1^{m_1}$ as $e_1^{m_1}$ where $p_1^{m_1}$
and $c_2$ = ···
 ⋮
and $c_n$= class $S_n$
          include ${}_1C_n^1, \ldots, {}_{l_n^1}C_n^1$ as $e_n^1$ where $p_n^1$ ···
          include ${}_1C_n^{m_n}, \ldots, {}_{l_n^{m_n}}C_n^{m_n}$ as $e_n^{m_n}$ where $p_n^{m_n}$
in $e$ end
```

${}_kC_i^j$ are either one of class identifiers $c_1, \ldots, c_n$ or any class expression that does not contain any of $c_1, \ldots, c_n$. e_i and p_i are as before and cannot contain class identifiers $c_1, \ldots, c_n$. This restricted use of recursion enables us to define a well founded semantics for the recursive class definitions.

The typing rule for this recursive class definition is given in Figure 6. Note that the restriction stated above is correctly enforced by using the type assignment $\mathcal{A}$ for p_i^j and e_i^j that does not contain the assumption on the variables $c_1, \ldots, c_n$. This works as follows. Suppose we

$$
\text{(rec-class)}\;\frac{\begin{array}{l}
\mathcal{K},\mathcal{A} \rhd S_i \;:\; \{obj(\tau_i)\}\\
\mathcal{K},\mathcal{A}\{c_1 : class(\tau_1),\ldots,c_n : class(\tau_n)\} \rhd_k C_i^j \;:\; class({}_k\tau_i^j)\\
\mathcal{K},\mathcal{A} \rhd e_i^j \;:\; {}_1\tau_i^j \times\cdots\times_{l_i^j} \tau_i^j \rightarrow \tau_i\\
\mathcal{K},\mathcal{A} \rhd p_i^j \;:\; obj({}_1\tau_i^j \times\cdots\times_{l_i^j} \tau_i^j) \rightarrow bool\\
\mathcal{K},\mathcal{A}\{c_1 : class(\tau_1),\ldots,c_n : class(\tau_n)\} \rhd e \;:\; \tau
\end{array}}{\mathcal{K},\mathcal{A} \rhd \textsf{let } c_1 = \cdots \textsf{ and } \cdots \textsf{ and } c_n = \cdots \textsf{ in } e \textsf{ end} \;:\; \tau}
$$

Figure 6: Typing rule for recursive class definition

have readly defined a class C and appempts to define mutually recursive classes C_1 and C_2 to represent the relation $C_1 = C \setminus C_2$ and $C_2 = C \setminus C_1$ by the code:

```
let $C_1$ = class {} includes $C$ as ···
            where λc.c-query(λs.not(member(c,s)), $C_2$)
and $C_2$ = class {} includes $C$ as ···
             where λc.c-query(λs.not(member(c,s)), $C_1$)
in ··· end
```

However, the intended semantics is itself ambiguous. In fact, this code cannot be well typed by our typing rules. On the other hand, the following recursive definition has clear meaning, and be well typed in our typing rules.

```
let $C_2$ = class {} includes $C_3$ as ···
            where λc.c-query(λs.not(member(c,s)), $C_1$)
and $C_3$ = class {} includes $C_2$ as ···
             where λc.c-query(λs.not(member(c,s)), $C_1$)
in ··· end
```

Let us show an example of classes with useful cyclic sharing. In the previous example on FemaleMember, if some object is newly inserted to the immediate extent of FemaleMember, we would like to make that object shared appropriately by either of Staff or Student. This implies FemaleMember and the other two classes should perform mutual sharing. One such class definition is shown in Figure 7. This examples shows that we can define useful mutual sharing by using recursive sharing.

To define an effective semantics for the recursive class definition, some care must be taken so that repeated inclusion of objects would be avoided. Let C be a recursive class definition of the form:

```
let $c_1 = \cdots$
and $c_i$= class $S_i$
      include ${}_1C_i^1, \ldots, {}_{l_i^1} C_i^1$ as $e_i^1$ where $p_i^1$ ···
      include ${}_1C_i^{m_i}, \ldots, {}_{l_i^{m_i}} C_i^{m_i}$ as $e_i^{m_i}$ where $p_i^{m_i}$
··· and $c_n = \cdots$
in $e$ end
```

We first define the set of mutually recursive function $\{f^i | 1 \leq i \leq n\}$ corresponding to the set of class identifiers $\{c_1, \ldots, c_n\}$ as follows:

```
$f^i = \lambda L \lambda ().S_i \cup \bigcup_{1\leq j\leq m_i}$ (select as $e_i^j$
                                   from intersect($\overline{{}_1C_i^j}, \ldots, \overline{{}_{l_i^j}C_i^j}$)
                                   where $p_i^j$)
```

where $\overline{{}_kC_i^j}$ is the following expression:

```
if ${}_kC_i^j = c_a$ then
  if $a \in L$ then {} else ($f^a$ ($L \cup \{a\}$)) ()
else $\mathbf{tr}({}_kC_i^j)$
```

The application $((f^i\ \{i\})\ ())$ computes the extent denoted by the class identifier c_i. The extra parameter L to f^i indicates the set of functions $\{f^a | a \in L\}$ that indirectly invoke the application in question. Thus if f^i is called with $\{a, \ldots\}$ then this call is to calculate the objects that are included in the class c_a and therefore the call to f^a is not needed.

The recursive definition is well defined in the following sense:

Proposition 5 *There is no infinite calling sequence of* $\{f^1, \ldots, f^n\}$ *starting from any* $(f^i\ \{i\})$.

Proof sketch Suppose we have an infinite sequence $(f^{a_1} L_1), (f^{a_2} L_2), \ldots, (f^{a_j} L_j), (f^{a_{j+1}} L_{j+1}), \ldots$ such that evaluation of $f^{a_j} L_j$ involves evaluation of $f^{a_{j+1}} L_{j+1}$. By the definition of each f^i, the following properties can be verified: (1) $|L_{j+1}| = |L_j| + 1$, and (2) $L_j \subseteq \{1, \ldots, n\}$. These three properties contradicts the fact that the sequence is infinite. ▌
This means that extent computation terminates whenever each p_i^j and e_i^j terminates.

It is a routine to translate the above definition into a term in the language with views. The translation for recursive class is then defined as follows:

```
$\mathbf{tr}(let\cdots)$ = let  $c_1$ = [OwnExt:=$S_1$, Ext=($f^1$ {1})]
             and ···
             and $c_n$ = [OwnExt:=$S_n$, Ext=($f^n$ {n})]
             in $e$ end
```

This operational semantics corresponds to the least (with respect to set inclusion) solution to the class definition when we consider it as a system of equation over sets of objects under objeq.

```
let Staff = class {} includes FemaleMember as λf.[Name = f·Name, Age = f·Age, Sex = "female"]
                                                   where λf.query(λx.(x·Category = "staff"), f)
and Student = class {} includes FemaleMember as λf.[Name = f·Name, Age = f·Age, Sex = "female"]
                                                     where λf.query(λx.(x·Category = "student"), f)
and FemaleMember = class {} includes Staff as λs.[Name = s·Name, Age = s·Age, Category = "staff"]
                                                   where λs.query(λx.(x·Sex = "female"), s)
                            includes Student as λs.[Name = s·Name, Age = s·Age, Category = "student"]
                                                     where λs.query(λx.(x·Sex = "female"), s)
in ··· end
```

Figure 7: A Simple Example of Mutually Recursive Class Definition

5 Conclusion

We have presented a typed polymorphic functional calculus that supports powerful view definitions and flexible object sharing. In this calculus, one can associate objects with views. Views can be any type consistent functions definable in the calculus. Those objects, i.e. a raw object associated with a viewing function can be manipulated by ordinary functions operating on records. The necessary code for functions to interface with objects is automatically generated by the system. One can define classes of those objects with general sharing specifications. Each such specification consists of a predicate on the objects to be included and a viewing function, both of which are defined in the same language for manipulating objects. It is also possible to define recursive classes, where inclusion relation contains cycles. We obtained the calculus by successively defining the necessary structures and operations for manipulating views and classes on top of the core calculus for records and sets. We also gave their precise semantics and an effective algorithm to implement them by developing a systematic method to translate those structures and operations into the calculus for records and sets.

One notable advantage of our approach is that our language uniformly integrates object-oriented database programming and typed polymorphic programming. We believe that this integration will open up the possibility of transferring various recent results in type theory to object-oriented database programming. For example, it is not hard to integrate the class structures we have developed with parametric classes for object-oriented programming [21].

Another interesting issue is abstract characterization of views. In developing a systematic translation for a language with views and classes into a language without them, we noticed an intriguing similarity between certain operations associated with Monads in category theory and our operations for lifting record operations to objects and classes. This suggests that we may be able to find a categorical characterization of object-oriented databases that can be applicable to wider range of data structures. Such an abstract analysis might be useful for comparing expressive powers of languages with views in different paradigm.

Acknowledgments

We would like to thank Kazuhiko Kato for valuable discussions and suggestions on object sharing. The second author is indebted to Takashi Masuda for his continuous support and encouragement.

References

[1] S. Abiteboul and A. Bonner. Objects and views. In *Proc. ACM SIGMOD Conference*, pp. 238–247, 1991.

[2] S. Abiteboul, G. Lausen, H. Uphoff, and E. Waller. Methods and rules. In *Proc. ACM SIGMOD Conference*, pp. 32–41, 1993.

[3] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Proc. VLDB Conference*, pp. 39–51, 1993.

[4] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. *ACM Transactions on Database Systems*, Vol. 10, No. 2, pp. 230–260, 1985.

[5] M.P. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrick, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. Deductive and Object-Oriented Database Conference*, 1989.

[6] F. Bancilhon, T. Briggs, S. Khoshafian, and P Valduriez. FAD a powerful and simple database language. In *Proc. Intl. Conf. on Very Large Data Bases*, pp. 97–105, 1988.

[7] F. Bancilhon, C. Delobel, and P. Kanellakis (eds). Building an OODBMS, the story of O_2. Morgan Kaufman, 1992.

[8] P. Buneman and A. Ohori. A type system that reconcile classes and extents. In *Proc. 3rd International Workshop on Database Programming Languages*, 1991.

[9] P. Buneman and A. Ohori. Polymorphism and type inference in database programming. Technical report, Universities of Pennsylvania, 1992. To appear in *ACM Transaction on Database Systems.*

[10] V. Breazu-Tannen, P. Buneman, and A. Ohori. Data structures and data types in object-oriented databases. *IEEE Data Engineering Bulletin*, Vol. 14, No. 2, pp. 23–27, 1991.

[11] T. Barsalou, N. Siambela, A.M. Keller, and G. Wiederhold. Updating relational databases through object-based views. In *Proc. ACM SIGMOD Conference*, pp. 248–257, 1991.

[12] R. Connor, A. Dearle, R. Morrison, and F. Brown. Existentially Quantified Types as a Database Viewing Mechanism. In *Proc. International Conference on Extended Database Technologies*, pp. 301–315, 1990.

[13] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley, 1983.

[14] G. Gottlob, P. Paolini, and R. Zicari. Properties and Update Semantics of Consistent Views. *ACM Transaction on Database Systems*, Vol. 13, No. 4, pp. 486–524, Dec. 1988.

[15] S. Heiler and S. Zdonik. Object Views: Extending the Vision. In *Proc. IEEE Data Engineering*, pp. 86–93, 1990.

[16] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. Technical report 90/14, State University of New York at Stony Brook, 1990.

[17] L. A. Jategaonkar and J.C. Mitchell. ML with extended pattern matching and subtypes. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 198–211, 1988.

[18] J. Meseguer and X. Qian. A logical semantics for object-oriented databases In *Proc. ACM SIGMOD Conference*, pp. 89–98, 1993.

[19] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, Vol. 17, pp. 348–375, 1978.

[20] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 174–183, 1988.

[21] A. Ohori and P. Buneman. Static type inference for parametric classes. In *Proc. ACM OOPSLA Conference*, pp. 445–456, 1989.

[22] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli – a polymorphic language with static type inference. In *Proc. ACM SIGMOD conference*, pp. 46–57, 1989.

[23] A Ohori. A compilation method for ML-style polymorphic record calculi. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 154–165, 1992.

[24] D. Remy. Typechecking records and variants in a natural extension of ML. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 242–249, 1989.

[25] J. Richardson and P. Schwarz. Aspects: extending objects to support multiple, independent roles. In *Proc. ACM SIGMOD Conference*, pp. 298–307, 1991.

[26] M. H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In *Proc. Deductive and Object-Oriented Databases Conference*, pp. 189–207, 1991.

[27] K. Tajima. A study on an object-oriented database system supporting multi-schema facilities. Master's thesis, The Graduate School of University of Tokyo, 1993.

[28] K. Tanaka, Y. Yoshikawa, and K. Ishikawa. Schema virtualization in object-oriented databases. In *Proc. IEEE Data Engineering Conference*, pp. 23–30, 1988.

[29] M. Wand. Complete type inference for simple objects. In *Proc. IEEE Symposium on Logic in Computer Science*, pp. 37–44, 1987.

[30] G. Wiederhold. Views, Objects, and Databases. In *IEEE Computer*, Vol. 19, No. 12, pp. 37–44, 1986.