Type Derivation Using the Projection Operation

Rakesh Agrawal   Linda G. DeMichiel

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Abstract

We present techniques for deriving types from existing object-oriented types using the relational algebraic projection operation and for inferring the methods that are applicable to these types. Such type derivation occurs, for example, as a result of defining algebraic views over object types. We refactor the type hierarchy and place the derived types in the type hierarchy in such a way that the state and behavior of existing types remain exactly as before. Our results have applicability to relational databases extended with object-oriented type systems and to object-oriented systems that support algebraic operations.

1 Introduction

In relational database systems, it is often useful to define *views* over sets of related data items for purposes of abstraction or encapsulation. Views are specified by using the standard algebraic query operations: e.g., projection, selection, join. A view is considered a virtual relation and is simply added to the the list of existing relations. Views are defined over relations, and the "type" of a view, like that of a relation, is implicitly given by the types of its attributes. The instantiation, or materialization, of a view is determined by the contents of those relations over which the view is defined.

Because of their usefulness in relational databases, views have also attracted considerable attention in object-oriented database systems [1] [9] [10] [11] [16] [17] [18]. However, unlike in relational systems, types and type extents are often decoupled in object-oriented type systems [3]. Thus it becomes important to separate the two aspects of view operations that are specific to the manipulation of types and type instances: (1) the derivation of new types as a result of the view operation; (2) the manipulation of instances of the source types of the view to obtain the instances of the type derived by the view operation. We term these aspects the *type derivation problem* and the *type instantiation problem* respectively. It is the first of these—the problem of dynamically deriving new types—that we shall address in this paper.

In object-oriented type systems, there are two aspects to such type derivation that must be considered:

- The behavior of the derived type must be inferred—that is, it must be determined which of the methods that are applicable to the source types of the derived type are applicable to the new type itself.
- The new type must be correctly positioned in the existing type hierarchy. The new type must be inserted into the type hierarchy in such a way that existing types are not affected: they must have both the same state and the same behavior as before the creation of the derived type.

We consider type derivation using the relational algebraic operations, focussing here only on the projection operation. Projection (Π) takes a type in terms of a set of attributes and creates a new type as defined by a subset of those attributes. Thus, if type T has attributes a, b, and c, the result of the projection operation $\Pi_{a,b}T$ is a type with two attributes, a and b. Of the relational operations, projection poses the greatest problems for type derivation because of the implicit refactorization of the type hierarchy that it entails.

1.1 Previous Work

Several proposals on views in object-oriented systems have suggested the derivation of a virtual type for a view [1] [9] [10] [11] [16] [17] [18]. As pointed out in [15], many of the current proposals for object-oriented types derived as the result of a view operation do not discuss the integration of the derived type into the existing type hierarchy. In some proposals the derived type is treated as a separate entity (e.g., [9]), in some the derived type is made a direct subtype of the root of the type hierarchy (e.g., [12]), and in some only the local relationship of the derived type with respect to the source type is established (e.g., [10] [14] [17]).

Previous work in the area of inserting derived types into the type hierarchy include [13] [15] [16] [19]. This problem has also been addressed in the knowledge representation literature (e.g., [4] [5] [7]). However, none of this work has addressed the problem of determining the behavior of the new type. It was proposed in [1] [6] that the type definer specify which existing methods are applicable to the new type. Determining which methods apply to a new type is a complex problem, and leaving it to the type definer is error-prone. Furthermore, it must be determined that the methods selected are indeed type-correct and mutually consistent.

1.2 Organization of this Paper

In Section 2, we give our model of an object-oriented type system. This model is intended to be sufficiently general that our results be applicable to a large number of specific systems. In Section 3, we motivate our approach with a simple example. Given a projection over a type T, we first determine which methods applicable to T will continue to be applicable to the new type, and these methods determine the behavior of the new type. The algorithm for determining applicable methods is presented in Section 4. This new type must now be integrated into the existing type hierarchy and its relationship to the other types established. To accommodate the new type in the existing type hierarchy, however, it may be necessary to refactor the existing hierararachy. This factorization derives a new type hierarchy that preserves the semantics of the original hierarchy and includes the new derived type. Section 5 discusses state factorization, and Section 6 discusses method factorization. We conclude by outlining some open problems in Section 7.

2 Model

We assume a general object-oriented type system in which a data type consists both of state and of a set of operations that can be applied to instances of the type. The state consists of a set of named *attributes* in which each attribute is associated with a type. Types are organized in a hierarchy and a *subtype relation* is defined over them. The meaning of this subtype relation corresponds to that of *subtype polymorphism*, or *inclusion polymorphism* [8]: A is a subtype of B, A and B not necessarily distinct, exactly when every instance of A is also an instance of B. The type hierarchy is a directed acyclic graph—that is, we allow multiple inheritance. The semantics of this inheritance is as follows:

- If A is a supertype of B, then every attribute of A is also an attribute of B.
- If D has supertypes B and C, and B and C have a common supertype A, then attributes of A are inherited only once by instances of D.

We assume that there is a precedence relationship among the direct supertypes of a type. (For a discussion of the role of this precedence relationship in method selection, see [2].) To simplify our presentation, we also assume that attribute names are unique.

Operations on the instances of types are defined by *generic functions*, where a generic function corresponds to a set of *methods* and the methods define the type-specific behavior of the generic function. A method is defined for a set of arguments of particular types and can be executed for any arguments that are instances of those types or their subtypes. The selection of the method to be executed depends on the types of the actual arguments with which the generic function is called at run time. A method can be an *accessor* method, which directly accesses the state associated with a single attribute of the type, or a method can be a general method, which may invoke other methods, including accessors. Accessor methods can be *reader* methods, which simply return the value of a particular attribute, or *mutator* methods, which alter the value of a particular attribute. The only access to the attributes of a type is through such methods.

We consider the general case in which methods are multi-methods. That is, when a generic function is called, the method dispatched at run time is selected on the basis of the types of all of the actual arguments to the call, as in, for example, CommonLoops, CLOS, and the proposed SQL3. In some object-oriented languages (e.g. C++, Smalltalk), there is a single, distinguished argument whose type determines which method is dispatched when a generic function is called. Since such single-argument method dispatch is a special case of multi-method dispatch, the results of our work can be applied to such languages as well.

We assume further that it is transparent to the user of a type whether a particular attribute or method of the type was defined locally at that type itself or inherited from one of its supertypes.

2.1 Notation

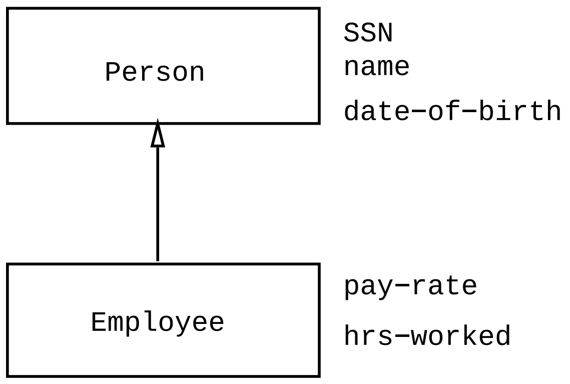
We represent the subtype relation by $\preceq$. If $A \preceq B \wedge A \neq B$, we say that A is a *proper subtype* of B and represent this relation by $\prec$. Since the proper subtype relation is a partial order, we can view such a system of types as a directed acyclic graph. There is a path from A to B if and only if A is a subtype of B. If $A \preceq B$, we also say that B is a *supertype* of A. The supertype relation is correspondingly denoted by $\succeq$. We will use upper case letters to denote types and the corresponding lower case letters to denote their instances. Thus, we will write a to denote an instance of type A. In our figures, we will draw an arrow from subtype to supertype to denote the subtype relationship. We denote the precedence relationships among the direct supertypes of a type by integers, with a lower number signifying higher precedence. Arrows in the figures are annotated to specify these precedence relationships.

We will denote a particular method m_k of an n-ary generic function m as $m_k(T_k^1, T_k^2, \ldots, T_k^n)$, where T_k^i is the type of the i^{th} formal argument of method m_k. The call to the generic function will be denoted without a subscript on m.

3 Projection over Types

The algebraic projection operation over a type T selects a subset of the attributes of T, specified in terms of a *projection list*, and derives a new type $\tilde{T}$. Intuitively, it seems natural that any methods associated with type T that are "appropriate" for $\tilde{T}$ should be applicable to instances of the new derived type $\tilde{T}$, and, correspondingly, that $\tilde{T}$ should be related to the source type T as its supertype, since $\tilde{T}$ contains a subset of the attributes of T. The following example illustrates what we mean. In Section 4, we formally define which methods are applicable to a derived type.

3.1 A Simple Example

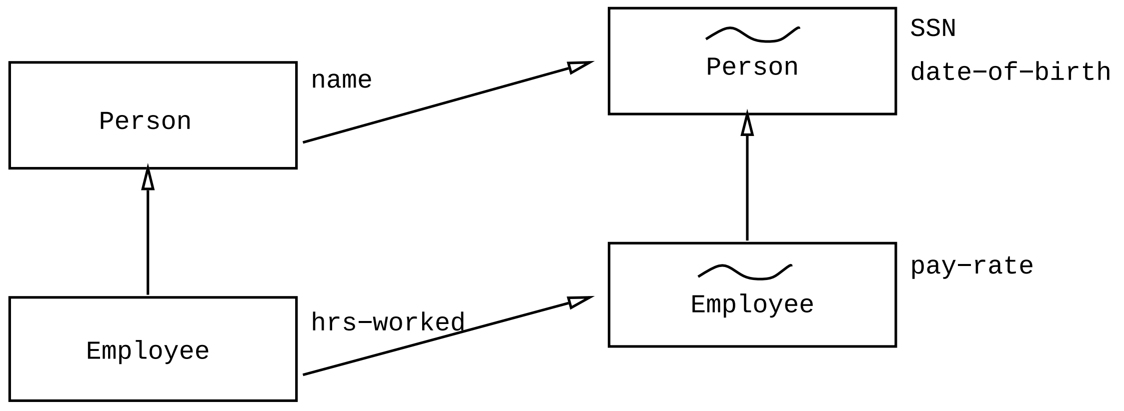


```
age(Person) = {...get_date-of-birth(Person)...}

income(Employee) = {...get_pay-rate(Employee),get_hrs-worked(Employee)...}

promote(Employee) = {...get_date-of-birth(Employee),get_pay-rate(Employee)...}
```

Figure 1: A simple type hierarchy

Consider the simple type hierarchy shown in Figure 1. The state of the type *Person* consists of attributes *SSN*, *name*, and *date-of-birth*. The type *Employee* inherits these attributes, and additionally defines the attributes *pay-rate* and *hrs-worked*. We assume that there exist accessor methods corresponding to each of these attributes: e.g. *get_SSN*, *get_name*, etc. The figure additionally shows the three methods *age*, *income*, and *promote*. The attributes used by each of these methods are indicated implicitly in the respective method bodies by calls to the corresponding accessor methods. The *age* method uses the attribute *date-of-birth* to compute the age of a person. Since *Employee* is a subtype of *Person*, this method can also be used to compute the age of an employee. The *income* method uses the *pay-rate* and *hrs-worked* attributes to compute an employee's income. The *promote* method uses the *date-of-birth* and *pay-rate* attributes to determine if an employee is eligible for promotion.

We now apply the projection operation to *Employee*, selecting only the *SSN*, *date-of-birth*, and *pay-rate* fields, thus deriving a new type, say, $\widetilde{Employee}$. We would like automatically to infer the methods that are applicable to $\widetilde{Employee}$, and to place $\widetilde{Employee}$ in the type hierarchy such that it inherits the correct state and behavior. In so doing, it is essential that the new type be inserted



$\mathtt{age(}\widetilde{\mathtt{Person}}\mathtt{) = \{...get\_date{-}of{-}birth(}\widetilde{\mathtt{Person}}\mathtt{)...\}}$

$\mathtt{income(Employee) = \{...get\_pay{-}rate(Employee),get\_hrs{-}worked(Employee)...\}}$

$\mathtt{promote(}\widetilde{\mathtt{Employee}}\mathtt{) = \{...get\_date{-}of{-}birth(}\widetilde{\mathtt{Employee}}\mathtt{),get\_pay{-}rate(}\widetilde{\mathtt{Employee}}\mathtt{)...\}}$

Figure 2: Refactored type hierarchy

into the type hierarchy in such a way that existing types are not affected.

The *income* method clearly does not apply to the new type $\widetilde{\textit{Employee}}$, since it depends on the *hrs-worked* attribute, which is not present in $\widetilde{\textit{Employee}}$. The *age* and *promote* methods, however, should be applicable to instances of $\widetilde{\text{Employee}}$.

The refactored type hierarchy is shown in Figure 2. In the refactored type hierarchy, *Employee* must be a subtype not only of *Person*, but also of $\widetilde{\textit{Employee}}$. Note, however, that $\widetilde{\textit{Employee}}$ cannot inherit from *Person* because of the absence of the *name* attribute in $\widetilde{\textit{Employee}}$. The type *Person* is therefore refactored into two types: $\widetilde{\textit{Person}}$, consisting of attributes *SSN* and *date-of-birth*, and *Person*, from which these attributes have been removed. Both *Person* and $\widetilde{\textit{Employee}}$ are thus made subtypes of the new type $\widetilde{\textit{Person}}$. It can be checked that the new type has the correct state and behavior, and that the types *Person* and *Employee* have both the same cumulative state and behavior as before the creation of the new type.

We present next the algorithms for determining the methods that are applicable to the derived type and for refactoring the type hierarchy.

4 Method Applicability

We say that a method $m_k(T_k^1, T_k^2, \ldots, T_k^n)$ is *applicable to a type* T if there is some T_k^i, $1 \leq i \leq n$ such that $T \preceq T_k^i$.

Any method m_k that is applicable to a type T is *applicable to a derived type* $\tilde{T}$, $\tilde{T} = \Pi_{a_1,\ldots,a_n} T$ unless m_k accesses an attribute of T not included in the projection list of $\tilde{T}$ or m_k invokes a generic function n on the argument T and there is no method of n that is applicable to $\tilde{T}$.

We say that a method $m_k(T_k^1, \ldots, T_k^n)$ is *applicable to a generic function call* $m(T^1, \ldots, T^n)$ if $\forall i$, $1 \leq i \leq n$, $T^i \preceq T_k^i$.

In general, because of subtype polymorphism, there can be more than one method that is applicable to a given generic function call. Given two methods m_i and m_j that are both applicable to a call, we assume that an ordering mechanism is provided that uniquely determines the precedence between those methods.[1] If m_i precedes m_j in accordance with this precedence ordering, we say that m_i is *more specific* than m_j. We term the method of highest precedence in this ordering the *most specific applicable method.*

4.1 Algorithm for Computing Method Applicability

Given the notion of method applicability, we can now describe how methods can be inferred for derived projection types.

The applicability of a method to a type can be determined by considering the call graph of the method. In the absence of recursion, the call graph must bottom out on accessor methods. If those accessor methods only access state that is present in the derived type, then the method being tested is applicable to that type. The actual algorithm presented below, however, is considerably more complex, because it takes into account cycles in the method call graph, the applicability of methods less specific than the most specific applicable method, and multiple arguments of the same source type.

The function *IsApplicable* comprises the heart of this algorithm. It is invoked on each method that is applicable to the source type T. *IsApplicable* tests a given method $m_k(T_k^1, T_k^2, \ldots, T_k^n)$ by analyzing its call graph, examining all generic function calls in the method body that are applicable to those method arguments that are supertypes of the source type T. In order for $m_k(T_k^1, T_k^2, \ldots, T_k^n)$ to be applicable, there must in turn be at least one applicable method for each such generic function call $n(T^1, \ldots, T^j, \ldots, T^m)$ in the body of m_k. We assume that the set of generic function calls in the body of m_k that need to be checked in this way is determined by data flow analysis.

[1]For a discussion of a number of such method precedence mechanisms and their relative power, see [2].

Consider a generic function call $n(T^1, \ldots T^j, \ldots, T^m)$ in the body of m_k. We distinguish two cases:

1. If only one of the arguments of the generic function call $n(T^1, \ldots T^j, \ldots, T^m)$—say T^j—is of type T or a supertype of T and corresponds to an argument of m_k, then the set of methods of n from which an applicable method must be found consists of those methods that are applicable to the call $n(T^1, \ldots T, \ldots, T^m)$.

   To see why, assume that $A \preceq B$ and that we are determining applicability of methods for the projection type $\tilde{A}$ derived from A. Assume that we have a method $m_k(B)$ whose body consists solely of the generic function call $n(B)$. We need to determine only if there is some method of n that is applicable to the call $n(A)$ that is also applicable to $\tilde{A}$. If such a method exists, then $m_k(B)$ should also be applicable to $\tilde{A}$, regardless of whether there is any method of n that is applicable to the call $n(B)$ that is also applicable to $\tilde{A}$.

2. If multiple arguments of the generic function call $n(T^1, \ldots T^j, \ldots, T^m)$ are of type T or supertypes of T and correspond to arguments of m_k, then we consider a method of n to be applicable to the call only if it is applicable to all combinations of non-null substitutions of $\tilde{T}$ for those supertypes. To obtain this, we require that the set of methods of n from which an applicable method is to be found consist of those methods that are applicable to the call $n(T^1, \ldots T^j, \ldots, T^m)$.

   To see why, assume that $A \preceq B$, $A \preceq C$ and that we are determining applicability of methods for the projection type $\tilde{A}$ derived from A. Consider a method $m_k(B, C)$ whose body consists solely of the generic function call $n(B, C)$. The existence of a method of n that is applicable to the call $n(\tilde{A}, \tilde{A})$ does *not* imply that there are applicable methods for the calls $n(B, \tilde{A})$ and $n(\tilde{A}, C)$.

We maintain three global data structures:

- *MethodStack* is a stack of methods, corresponding to the call stack of *IsApplicable*. Each entry in *MethodStack* is a pair <*method*, *dependencyList*>, where *dependencyList* keeps track of those methods whose applicability is contingent on the applicability of *method*.

- *Applicable* is a list of those methods that have been determined to apply to the derived type $\tilde{T}$ thus far. *Applicable* is computed optimistically in the following sense: Suppose a method m_i calls a generic function n which requires determination of the applicability of n_j. If the applicability of n_j is already in the process of being determined (i.e., n_j is a method inside *MethodStack*), then we assume that n_j is applicable, and conditionally determine the applicability of m_i. If n_j is later determined to be not applicable, then m_i is removed from the *Applicable* list.

- *NotApplicable* is a list of those methods that have been determined to not apply to the derived type $\tilde{T}$ thus far.

Initially, *MethodStack*, *Applicable*, and *NotApplicable* are all empty. *IsApplicable*$(m, T, projectionlist)$ is then called on all methods m_k that are applicable to T. After the end of each top-level call to the function *IsApplicable*, *MethodStack* is empty.

```
function IsApplicable(m:method, T: type, p: projectionlist) returns {applicable, notapplicable}
  /* first check if this method has already been processed */
  if $m \in$ Applicable then
    return applicable
  else if $m \in$ NotApplicable then
    return notapplicable
  if $m$ is an accessor method then
  /* $m$ must be a method on $T$ or a supertype of $T$ or we wouldn't be here */
    if $m$ accesses a field in $p$ then
      Applicable $\leftarrow$ Applicable $\cup$ $m$
      return applicable
    else
      NotApplicable $\leftarrow$ NotApplicable $\cup$ $m$
      return notapplicable
  /* $m$ is a general method */
  if $m$ is anywhere in MethodStack then
    dependencyList$(m)$ $\leftarrow$ dependencyList$(m)$ $\cup$ all methods above $m$ in MethodStack
    return applicable
  else
    push $m$ onto top of MethodStack /* set up for recursive calls to IsApplicable */
    for all generic functions calls $n(x_1, ...x_n)$ in the body of $m$ that are relevant to the arguments of $m$ do
      for all methods $n_k$ of generic function $n$ that are applicable do
        if IsApplicable$(n_k, T, p)$ then
        /* if any method checks out, the call to $n$ succeeds */
          continue to next generic function call in the body of $m$
      od
      /* falling out of inner loop means there is no applicable method for some function call $n$ */
      for all methods $d$ in dependencyList$(m)$ do
          Applicable $\leftarrow$ Applicable $-$ $d$
      od
      NotApplicable $\leftarrow$ NotApplicable $\cup$ $m$
```

```
        pop $MethodStack$ /* remove $m$ from top of stack */
        return $notapplicable$
    od
    /* falling out of outer loop means there are applicable methods for all function calls in $m$ */
    $Applicable \leftarrow Applicable \cup m$
    pop $MethodStack$ /* clean up the stack */
    return $applicable$
```

4.2 Example 1

Consider the type hierarchy as shown in Figure 3 below. We have annotated the type nodes with the names of attributes defined at those nodes.

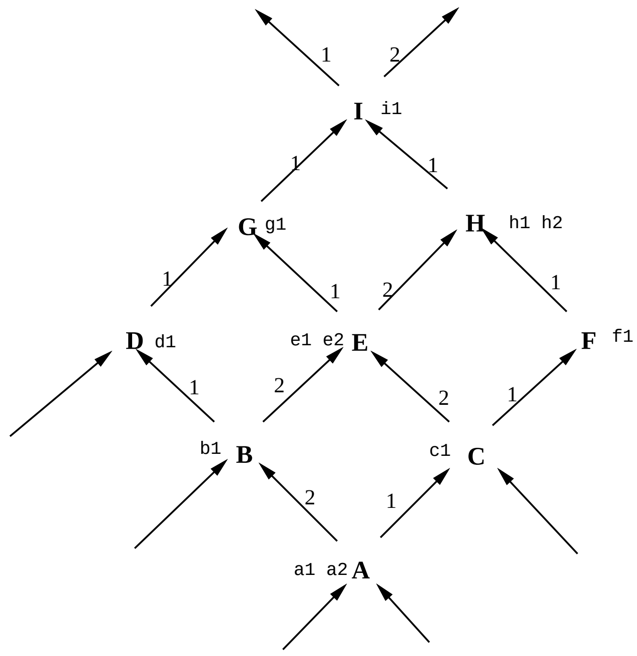


Figure 3: Original type hierarchy.

Consider also the following set of generic functions and methods:

$$u_1(A) = \{get\_a_1(A)\}$$
$$u_2(C) = \{get\_g_1(C)\}$$
$$u_3(B) = \{get\_h_2(B)\}$$

$$v_1(A, C) = \{u(A);\ w(C)\}$$
$$v_2(B, C) = \{get\_b_1(B);\ u(C)\}$$

$w_1(A) = \{get\_a_1(A)\}$
$w_2(C) = \{u(C)\}$

$x_1(A, B) = \{y(A, B);\ v(B, A)\}$

$y_1(A, B) = \{x(A, B)\}$

The subscripts after the generic function name are used to denote specific methods of that generic function. The function calls in a method body are shown enclosed in $\{\}$'s. We ignore for now the return values of the methods. The methods $get\_a_1(A)$, $get\_b_1(B)$, $get\_h_2(B)$, and $get\_g_1(C)$ are accessor methods and access the attributes a_1, b_1, h_2 and g_1 respectively; their bodies are not shown.

We consider the projection $\tilde{A} = \Pi_{a_2 e_2 h_2} A$ and compute the methods that are applicable to the derived type $\tilde{A}$.

First, we note that all the methods given are applicable to the source type A. We begin by testing the applicability of v_1 for $\tilde{A}$. Since this is the first method to be tested, *MethodStack*, *Applicable*, and *NotApplicable* are all empty. Interaction with *MethodStack* becomes interersting only for recursive function calls. To simplify exposition, we do not describe this interaction for the checking of v_1.

The initial call to *IsApplicable* is *IsApplicable*$(v_1, A, a_2e_2h_2)$. The body of v_1 contains the function calls $u(A)$ and $w(C)$, and we check them in that order. All three methods of u are applicable to an argument of type A. The order in which these methods are tested for applicability does not matter, since as long as one of them is applicable, the call $u(A)$ will succeed. Let us assume that we test them in the order in which they occur above. To test method u_1 we recursively call *IsApplicable*$(u_1, A, a_2e_2h_2)$. *IsApplicable* is then called again on the accessor method get_a_1, which is invoked in the body of u_1 and which accesses the field a_1, not present in the projection list. The method get_a_1 is thus added to the *NotApplicable* list and the last recursive call returns, and then u_1 is added to the *NotApplicable* list and the previous recursive call returns. The method u_2 is tested next by calling *IsApplicable*$(u_2, A, a_2e_2h_2)$. The method u_2 calls the accessor get_g_1, which accesses the field g_1, also not present in the projection list. Methods get_g_1 and u_2 are therefore likewise added to the *NotApplicable* list. We next call *IsApplicable*$(u_3, A, a_2e_2h_2)$. The accessor method u_3 accesses the field h_2, which is present in the projection list, and the test succeeds. We therefore add get_h_2 and u_3 to the *Applicable* list. The function call $u(A)$ in v_1 thus checks out.

We now have to check the function call $w(C)$. Since w is a function of only one argument, $w(C)$ will check out if there is an applicable method for the call $w(A)$. To check w_1, we call *IsApplicable* on the accessor method get_a_1. The method get_a_1 is already in the *NotApplicable* list. Therefore, *IsApplicable* returns *notapplicable* and w_1 in turn is added to the *NotApplicable* list. Next, we call

IsApplicable$(w_2, A, a_2e_2h_2)$. The method w_2 contains the call $u(C)$. This call will check out if there is any applicable method for the call $u(A)$. Methods u_1 and u_2 are in the *NotApplicable* list, but u_3 is in the *Applicable* list. Since $u(C)$ is the only call in the body of w_2, w_2 is therefore added to the *Applicable* list.

Dropping back a level, we see that there are no more function calls in the body of v_1, so we add v_1 to the *Applicable* list. Note that as a side effect of checking the applicability of v_1 we have determined the applicability of u_1, u_2, u_3, w_1, w_2, and the three accessor methods *get_a*$_1(A)$, *get_h*$_2(B)$, and *get_g*$_1(C)$.

Let us now check the method x_1. Note that the method x_1 is indirectly recursive. We shall show the interaction with the *MethodStack* in this case. Recall that *get_a*$_1(A)$, *get_h*$_2(B)$, *get_g*$_1(C)$, u_3, w_2, and v_1 are in the *Applicable* list, methods u_1, u_2, and w_1 are in the *NotApplicable* list, and the *MethodStack* is empty. The initial call is *IsApplicable*$(x_1, A, a_2e_2h_2)$. We push x_1 onto the *MethodStack*. The first call in the body of x_1 is $y(A, B)$. Method y_1 is applicable to this call and is checked next.

There is one function call in the body of y_1, $x(A, B)$. We push y_1 onto the *MethodStack*, and call *IsApplicable*$(x_1, A, a_2e_2h_2)$. In the body of *IsApplicable*, it is found that x_1 is already on the *MethodStack*. The only method above x_1 on the *MethodStack/* is y_1. We add y_1 to the *dependencyList*(x_1) (previously empty) and return *applicable*. Since we are in the process of determining the applicability of x_1, we are optimistically assuming that x_1 will succeed and keeping track that y_1 might have been declared applicable based on this assumption. If x_1 is later found to be applicable, nothing needs to be done. If, however, x_1 turns out to be not applicable, we will remove dependent methods from the *Applicable* list. Since there is no other call in the body of y_1, we add y_1 to the *Applicable* list, and pop y_1 from the *MethodStack*.

The next call in the body of x_1 is $v(B, A)$. The method v_1 does not apply to the call $v(B, A)$, but v_2 does. We push v_2 onto the stack and check for the applicability of *get_b*$_1(B)$. The accessor method *get_b*$_1(B)$ accesses the attribute b_1, which is not present in the projection list, so we add *get_b*$_1(B)$ to the *NotApplicable* list and return *notapplicable*. Method v_2 is added to the *NotApplicable* list, and we pop v_2 from the *MethodStack* and return *notapplicable*. Since there is no other applicable v method, this means that x_1 is not applicable. Since method y_1 is in the *dependencyList*(x_1), we must remove it from the *Applicable* list. We then add x_1 to the *NotApplicable* list and pop it from the *MethodStack*.

Note that we did not put y_1 in the *NotApplicable* list when we removed it from the *Applicable* list. Its status at this stage is still unknown, and this method will have to be checked again. It turns out that method y_1 will in fact be determined to be not applicable because x_1 is the only applicable method for the call $x(A, B)$. However, had there been another applicable method, say x_2, the outcome might have been different.

5 Factoring State

The creation of a derived type $\tilde{T}$ as a result of a projection operation induces a *refactorization* of the original type hierarchy in order to accommodate the inclusion of $\tilde{T}$. This factorization derives a new type hierarchy that preserves the semantics of the original hierarchy and includes the new derived type. Applicable methods are relocated in this refactored hierarchy such that all original types continue to have the same behavior as before, and the new type inherits all applicable methods and no others.

The original type hierarchy is factored into a derived type hierarchy by introducing what we shall call *surrogate types*. A *surrogate type* is a type that assumes a part of the state or behavior of the source type from which it is spun off. The types that are derived by the projection operator are thus themselves such surrogate types. The surrogate type plus the modified source type, when combined by means of inheritance, have exactly the state and behavior of the original source type. The state of the surrogate is determined by that portion of the projection list that applies to its source type: it consists precisely of those attributes that are contained both in the projection list and in the local attributes of the source type. The behavior of the surrogate type is determined according to which methods are applicable in the sense defined in Section 4.

The factorization into surrogate types is necessary to capture that portion of the state or behavior of the source type that is applicable to the derived type while not including attributes and behavior from the source type and its supertypes that are not. The factorization is recursive in the sense that each type Q through which the new derived type inherits attributes or methods is factored into two types: a surrogate type $\tilde{Q}$, which contains only those attributes that are inherited by the derived type $\tilde{T}$; and the modified source type Q, from which those attributes are removed. Q is then further modified to be a direct subtype of its surrogate $\tilde{Q}$. Type $\tilde{Q}$ is given the highest precedence of any of the supertypes of Q in order that this factorization be transparent from the standpoint of the state and behavior of the combined Q–$\tilde{Q}$ types.

We present below the algorithm for state factorization. Method factorization is presented in Section 6.

5.1 Algorithm for Factoring State

```
procedure FactorState(A:attributeList, T:type, $\tilde{R}$:type, P:precedence)
  if the surrogate type $\tilde{T}$ for $T$ and $A$ does not already exist then
    create a new type $\tilde{T}$
  make $\tilde{T}$ a supertype of $T$ such that $\tilde{T}$ has highest precedence among the supertypes of $T$
  if $\tilde{R} \neq NULL$ then
    make $\tilde{R}$ a subtype of $\tilde{T}$ with precedence $P$
```

```
if type $\tilde{T}$ was created in this call then
  $\forall\ a \in A$ such that $a$ is a local attribute of $T$ do
    move $a$ to $\tilde{T}$
  od
  let $S$ be the list of the direct supertypes of T, excluding $\tilde{T}$
    $\forall\ s \in\ S$ in order of inheritance precedence do
      let $p$ be the precedence of $s$ among the supertypes of $T$
        let $L$ be the list of attributes in $A$ that are available at $s$
          if $L \neq \emptyset$ then
            call $FactorState(L,\ s,\ \tilde{T},\ p)$
    od
```

The initial call is *FactorState*(*projection-list*, T, $NULL$, 0).

5.2 Example 2

Consider the type hierarchy shown in Figure 3. After the projection operation $\Pi_{a_2 e_2 h_2} A$, the type hierarchy will be as shown in Figure 4. We note that the projection list is $a_2 e_2 h_2$ and the initial call to *FactorState* is $FactorState(a_2 e_2 h_2, A, NULL, 0)$.

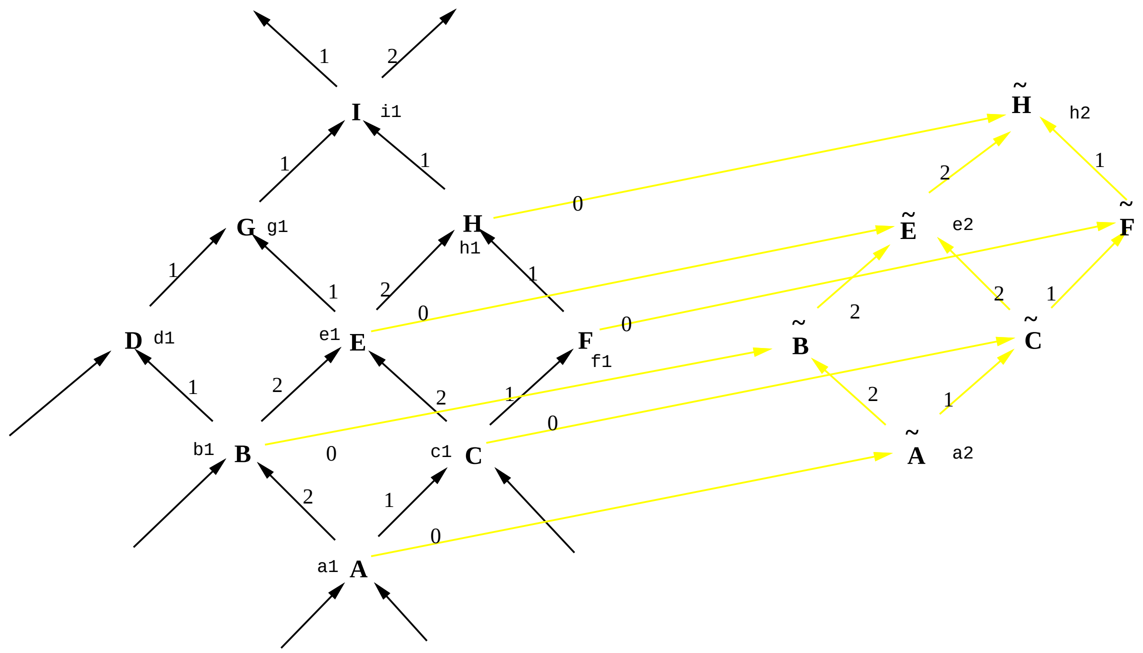


Figure 4: Factored type hierarchy after the projection $\Pi_{a_2 e_2 h_2} A$.

In the body of *FactorState*, a surrogate supertype $\tilde{A}$ is created for A. This supertype is given the highest precedence of A's supertypes, indicated as 0. The attribute a_2 is then moved from A to $\tilde{A}$. Types B and C are both supertypes of A, but C has higher precedence, so *FactorState* is first called recursively as *FactorState*(e_2h_2, C, $\tilde{A}$, 1), and then as *FactorState*(e_2h_2, B, $\tilde{A}$, 2).

In the body of the first of these calls, *FactorState*(e_2h_2, C, $\tilde{A}$, 1), a surrogate supertype $\tilde{C}$ is created for C, and then $\tilde{A}$ is made a subtype of $\tilde{C}$. The precedence of $\tilde{C}$ for C is set to 0, and the precedence of $\tilde{C}$ for $\tilde{A}$ is set to 1. No local attribute of C is present in the projection list, so no attribute is moved from C to $\tilde{C}$. Type C also has two supertypes, E and F, of which F has the higher precedence. Only the attribute h_2 is available at F; both the attributes e_2 and h_2 are available at E. *FactorState* is therefore called twice: first as *FactorState*$(h_2, F, \tilde{C}, 1)$, then as *FactorState*$(e_2h_2, E, \tilde{C}, 2)$.

Let us trace the first call. A surrogate supertype $\tilde{F}$ is created for F and $\tilde{C}$ is made a subtype of $\tilde{F}$. The precedence of $\tilde{F}$ for F is set to 0, and the precedence of $\tilde{F}$ for $\tilde{C}$ is set to 1. No attribute is moved from F to $\tilde{F}$. The attribute h_2 is available at H, the only supertype of F, so the next call to *FactorState* is *FactorState*$(h_2, H, \tilde{F}, 1)$. This call creates a surrogate supertype $\tilde{H}$ for H, makes $\tilde{F}$ a subtype of $\tilde{H}$, sets the precedence of $\tilde{H}$ for H to 0 and of $\tilde{H}$ for $\tilde{F}$ to 1, and moves h_2 from H to $\tilde{H}$. There is no further recursive call to *FactorState*.

We now trace the second call. This time, a surrogate supertype $\tilde{E}$ is created for E, $\tilde{C}$ is made a subtype of $\tilde{E}$, the precedence of $\tilde{E}$ for E is set to 0 and the precedence of $\tilde{E}$ for $\tilde{C}$ is set to 2, and e_2 is moved from E to $\tilde{E}$. Note that the relative precedence between E and F as specified by type C is preserved by $\tilde{C}$ in the relative precedence between $\tilde{E}$ and $\tilde{F}$. Type E has two supertypes. None of the attributes in the projection list are available at the higher precedence supertype G, so *FactorState* is not called for G. The call for the supertype H is *FactorState*$(h_2, H, \tilde{E}, 2)$. The surrogate supertype $\tilde{H}$ already exists. We therefore simply make $\tilde{H}$ a supertype of $\tilde{E}$ with precedence 2 and return.

We drop back to the second call to FactorState from A for B, which is FactorState(e_2h_2, B, $\tilde{A}$, 2). The surrogate supertype $\tilde{B}$ is created for B with precedence 0. $\tilde{B}$ is also made a supertype of $\tilde{A}$ with precedence 2. No state is moved to $\tilde{B}$. B has two supertypes, but none of the attributes in the projection list are present in D. The only recursive call therefore is *FactorState*$(e_2h_2, E, \tilde{B}, 2)$. The surrogate type $\tilde{E}$ already exists, so we simply make $\tilde{E}$ a supertype of $\tilde{B}$ with precedence 2 and return. The type refactorization is now complete.

6 Factoring Methods

Because the surrogate type is the direct supertype of its source type and furthermore because it is the supertype of highest precedence, any method $m_i(T_i^1, \ldots T_i^j, \ldots, T_i^m)$ that is applicable to a type $\tilde{T}_i^j$, $\tilde{T}_i^j \succ T_i^j$, can be treated as if it were a method on $m_i(T_i^1, \ldots \tilde{T}_i^j, \ldots, T_i^m)$.

The following algorithm associates the applicable methods for a type $\tilde{T}$ with those types from which $\tilde{T}$ inherits according to the refactored type hierarchy. This algorithm considers only method signatures. We discuss treatment of the method bodies in Section 6.3.

6.1 Algorithm for Factoring Methods

```
procedure FactorMethods(T:type)
  ∀ methods $m_k \in Applicable(\tilde{T})$ as determined by IsApplicable do
    let the signature of $m_k$ be $m_k(T_k^1, T_k^2, \ldots, T_k^n)$
    create a new signature for $m_k$ in which $T_k^i$ is replaced by $\tilde{T}_k^i$ $(i = 1 \ldots n)$
      for all $T_k^i$ for which a surrogate type $\tilde{T}_k^i$ was created by FactorState
  od
```

6.2 Example 3

Consider the methods given in Example 1. The *IsApplicable* procedure determined that the methods v_1, u_3, w_2, and get_h_2 are applicable. The other methods shown in the example were determined to be not applicable. The method factoring algorithm will therefore create the following new signatures:

$v_1(\tilde{A}, \tilde{C})$
$u_3(\tilde{B})$
$w_2(\tilde{C})$
$get\_h_2(\tilde{B})$

6.3 Processing of Method Bodies

Because of assignment and variable binding, however, modification of the method signature alone may not be sufficient. In particular, if done naively, such modifications may cause type errors in the method body.

Consider, for example, the following original method:

$z_1(c : C) = \{g : G;\ g \leftarrow c;\ \ldots\ u(c);\ \ldots\ return(g); \}$

If we change the method signature of z_1 to $z_1(c : \tilde{C})$, we introduce a type error in the assignment $g \leftarrow c$ if $\tilde{C}$ is not a subtype of G.

It is therefore necessary to analyze in the method body the reachability set for the use of all parameters that are to be converted to their corresponding surrogate types. The type declarations for any variables in this set need to be changed to declarations in terms of the corresponding surrogate types. In some cases, as in the above example, these surrogate types may not yet exist. The following algorithm shows how they are added to the extended type hierarchy. Note that no attributes need to be moved to these surrogates. The result type of the method is processed in the same way.

6.4 Augmenting the Type Hierarchy

```
let $X$ = set of types for which a surrogate was created by FactorState
let $F$ = set of methods determined applicable by IsApplicable
let $Y$ = set of types that are are assigned transitively a value of one of the types in $X$ by one of
        the methods in $F$ (this set is determined by the standard definition-use flow analysis)
let $Z = Y - X$

procedure Augment(T: type, Z: set of types)
  if $T$ has a supertype that is a subtype of one of the types in $Z$ then
    for all direct supertypes of $T$ except $\tilde{T}$ in order of precedence do
      let $S$ be the direct supertype with precedence $p$
        if $\tilde{S}$ does not exist then
          create $\tilde{S}$
          make $S$ a subtype of $\tilde{S}$ with highest precedence
          if $\tilde{T}$ is not already a subtype of $\tilde{S}$ then
            make $\tilde{T}$ a subtype of $\tilde{S}$ with precedence $p$
        Augment($S$, $Z$)
    od
```

The initial call is *Augment*(A,Z).

6.5 Example 4

Suppose that after the analysis it is found that $Z = \{D, G\}$. The augmented type factorization graph will be as shown in Figure 5 below.

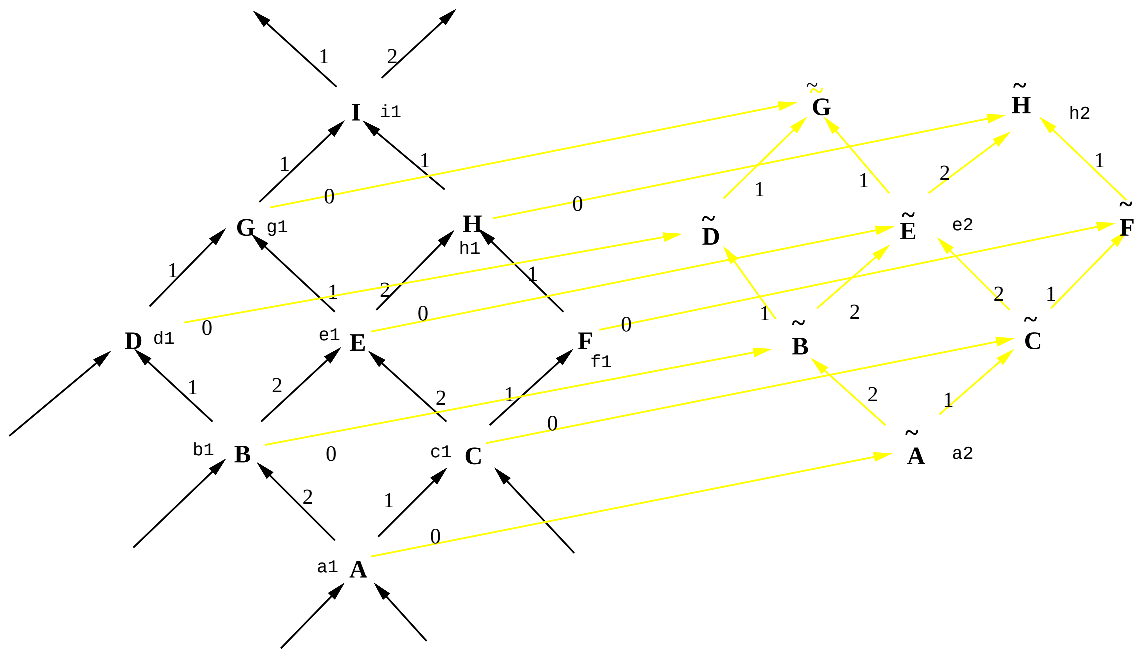


Figure 5: Augmented type hierarchy after processing of the method body for z_1.

7 Conclusions and Future Work

We have presented mechanisms for deriving new types from existing types using the relational algebraic projection operation and inferring those methods that continue to be applicable to the new view type. We have shown how the type hierarchy can be refactored and the new type placed in the type hierarchy in such a way that the cumulative state and behavior of existing types remain as before. Our results have applicability to relational databases extended with object-oriented type systems and to any object-oriented system that supports algebraic operations.

We believe that the work presented in this paper opens up several interesting areas for future work. From a practical point of view, it needs to be investigated how—if at all—the number of surrogate types with empty states can be reduced in the refactored type hierarchy, particularly when views are defined over views. The mechanisms presented in this paper turn out to be fairly complex because we have considered the general case involving multiple inheritance and multi-methods. It will be interesting to specialize the solutions presented in this paper for specific cases of object-oriented type systems that do not require this generality. Finally, the methodology presented in this paper needs to be applied to the remaining algebraic operations.

References

[1] Serge Abiteboul and Anthony Bonner. "Objects and Views." *Proc. of the ACM SIGMOD International Conference on Management of Data*, Denver, Colorado, May 1991, 238–247.

[2] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. "Static Type Checking of Multi-Methods." In *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Phoenix, Arizona, October 1991, 113–128.

[3] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. "The Object-Oriented Database System Manifesto." *Proc. of 1st International Conference on Deductive and Object Oriented Databases (DOOD)*, Kyoto, Japan, December 1989.

[4] H. W. Beck, S. K. Gala, and S. B. Navathe, "Classification as a Query Processing Technique in the CANDIDE Semantic Data Model." *Proc. of 5th International Conference on Data Engineering*, Los Angeles, California, February 1989, 572–581.

[5] S. Bergamaschi and C. Sartori. "On Taxonomic Reasoning in Conceptual Design", *ACM Transactions on Database Systems*, **17**(3), Sept. 1992, 385–422.

[6] Elisa Bertino. "A View Mechanism for Object-Oriented Databases." *Proc. of 3rd International Conference on Extending Database Technology (EDBT)*, Vienna, Austria, March 1992.

[7] A. Borgida, R.J. Brachman, D.L. McGuinnes, and L.A. Resnick. "CLASSIC: A Structural Data Model for Objects" *Proc. of the ACM SIGMOD International Conference on Management of Data*, Portland, Oregon, May-June 1989, 58–67.

[8] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, **17**(4), December 1985.

[9] Sandra Heiler and Stanley Zdonik. "Object Views: Extending the Vision." *Proc. of 6th International Conference on Data Engineering*, Los Angeles, California, February 1990, 86–93.

[10] M. Kaul, K. Drosten, E.J. Neuhold. "ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views." *Proc. of 6th International Conference on Data Engineering*, Los Angeles, California, February 1990, 2–10.

[11] Michael Kifer, Won Kim, and Yehoshua Sagiv. "Querying Object-Oriented Databases." *Proc. of the ACM SIGMOD International Conference on Management of Data*, San Diego, California, June 1992, 393–402.

[12] Won Kim. "A Model of Queries in Object-Oriended Databases", *Proc. of 15th Very Large Data Base Conference (VLDB)*, Amsterdam, The Netherlands, August 1989, 423–432.

[13] M. Missikoff and M. Scholl. “An Algorithm for Insertion into a Lattice: Application to Type Classification.” In *Foundations of Data Organization and Algorithms*, ed. W. Litwin, H.-J. Schek. Springer Verlag, 1989.

[14] M.M. Morsi, S.B. Navathe, H.K. Kim. “An Extensible Object-Oriented Database Testbed”. *Proc. of 8th International Conference on Data Engineering*, Tempe, Arizona, February 1992.

[15] Elke A. Rundensteiner. “MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases.” *Proc. of 18th Very Large Data Base Conference (VLDB)*, Vancouver, Canada, August 1992, 187–198.

[16] Marc H. Scholl, Christian Laasch, Markus Tresch. “Updatable Views in Object-Oriented Databases.” *Proc. of 2nd International Conference on Deductive and Object Oriented Databases (DOOD)*, Germany, December 1991.

[17] Michael Schrefl and Erich J. Neuhold. “Object Class Definition by Generalization Using Upward Inheritance.” *Proc. of 4th International Conference on Data Engineering*, Los Angeles, California, February 1988, 4–13.

[18] Gail M. Shaw and Stanley B. Zdonik. “A Query Algebra for Object-Oriented Databases.” *Proc. of 6th International Conference on Data Engineering*, Los Angeles, California, February 1990, 154–162.

[19] Katsumi Tanaka, Masatoshi Yoshikawa, and Kozo Ishihara. “Schema Virtualization in Object-Oriented Databases.” *Proc. of 4th International Conference on Data Engineering*, Los Angeles, California, February 1988, 23–30.