

# A Relational Account of Objects

Clara Murdaca & C. Barry Jay  
University of Technology, Sydney  
{murdaca,cbj}@it.uts.edu.au

## Abstract

A relational account of objects provides a single unifying data model for both object-oriented programming languages and relational databases. Type variables used to represent unknown fields in the programming language correspond to discriminators in relations.

## 1 Introduction

Relational databases have proved their worth in storing and manipulating large amounts of data, while object-oriented programming languages are excellent at organising computation in a way that is flexible and maintainable. Ideally, object-oriented programs should be compiled to take full advantage of data stored in relational tables but this is difficult, if not impossible, while ever the data model for classes differs from that for relations. This paper proposes a relational account of objects suitable for both programming and the organisation of relational databases.

Various approaches to storing objects have been considered. Simplest is to store the objects in the database just as they are stored in working memory of the language, to create an *object base* (see e.g. (Bloom & Zdonik 1987, Breazu-Tannen, Buneman & Ohori 1991, Atkinson, Bancilhon, DeWitt, Dittrich, Maier & Zdonik 1994, Leontiev, Ozsu & Szafron 2002)). Alternatively, one can aim for *persistence* (Agrawal, Dar & Gehani 1993, Richardson, Carey & Schuh 1989), or attempt a mixture of data models, as in C-omega (Bierman, Meijer & Schulte 2005). Unfortunately, these all reduce the time and space efficiency of the store in ways which have proved resistant to improvement. An emerging approach (described in Section 2) is to try and model the classes as relations, with objects modeled as rows in tables. This is non-trivial since the standard model of classes is quite different from that of relations. In particular, relations do not support a natural interpretation of sub-classes, resulting in a variety of competing approaches; these can be compared in terms of the efficiency with which objects can be stored and retrieved, and the ease of maintenance.

However, none of these approaches to storage is able to combine efficient data storage and retrieval with the modular compilation of classes. Modular compilation is highly desirable when creating large systems. It is also a pre-condition for being able to invest in significant optimisation of code to take advantage of the efficiency of the underlying query lan-

guage. The typical approach has an object-oriented process request an object from the database using some tool, updates the object and then uses the tool to return it to the database. With a shared object model and modular compilation it may be possible to compile some methods directly into queries.

One consequence of modular compilation is that the introduction of new sub-classes should not change the representation of their super-classes, e.g. should not require re-factorisation. The simplest way to achieve this is to require that each class have its own relation.

Here then are four desirable properties that such a model should have.

- (P1) Data storage is efficient.
- (P2) Compilation is modular.
- (P3) There is one relation per class.
- (P4) Compiled methods take advantage of queries.

Object bases and systems with persistence do not have the first property, so let us focus on modeling classes as relations, as described in (Ambler 2003). Only one of the approaches therein satisfies (P3); in the other three approaches the relation(s) describing a class evolve as new sub-classes are created. However, all of the approaches invoke a method by converting a row into an object and proceeding in the usual object-oriented style: determine its class and execute the corresponding class code. To adopt this approach when compiling into queries would be to import the whole notion of class into the query language, rather than exploit the inherent nature of queries.

More suitable for the purpose is an alternative approach to object-orientation, outlined in (Jay 2004b, Jay 2004d). Objects are represented by data structures built from their fields, while methods are given as functions that use pattern-matching to identify the appropriate algorithm. Sub-classing and sub-typing are handled by using type variables to represent unknown fields, in an approach similar to, but not identical, to that of *row variables* (Remy 1989). This makes it easy to satisfy properties (P1) and (P3). The sub-type relationship is captured using *discriminators* in relations. Matching on constructors in the programming language becomes a query on the discriminator, making it easier to represent methods as queries (P4). Finally, separate compilation (P2) can be supported by treating method specialisation as the addition of a new case to an existing (compiled) pattern-matching function.

The elaboration of this approach is ongoing work. The basic theory is supplied by the *pattern calculus* (Jay 2004c) which has been implemented in the programming language **bondi** (pronounced “bond-eye”) (Jay 2004a) which is able to support classes and sub-typing. These can be used to create relations in a

relational database so that objects can be created, stored and retrieved in a natural way. Class compilation is already modular in **bondi** but does not yet exploit the power of the query language.

This paper introduces the *discriminating object model* and shows how it satisfies properties (P1–P3). It underpins both the class model of **bondi** and its organisation of relational data, which can then be used to illustrate the ideas. This provides a foundation for exploring (P4).

The structure of the rest of the paper is as follows. Section 2 reviews current object relational mapping techniques. Section 3 reviews the pattern calculus. Section 4 presents our object model. Section 5 draws conclusions and considers future work.

## 2 Object-Relational Mapping

```

public abstract class Entity
{ private int entity_id;
  private String name;
  private String address;
}

public class Student extends Entity
{ private String course;
}

public class Employee extends Entity
{ private double salary;
  public double SalaryIncrease()
  { return salary * 1.10; }
}

public class Casual extends Employee
{ private double hours;
  private double SalaryIncrease()
  { return salary * 1.06; }
}

```

Figure 1: Running example in Java

There are various ways in which objects can be mapped to relational tables (Fussell 1997, Keller 1998). These can be classified into four main object relational mapping techniques according to their treatment of class hierarchies (Ambler 2003).

The example given in Figure 1 will be used to illustrate these four mapping techniques. Figure 1 defines a class hierarchy using Java syntax. It contains an abstract *Entity* class with subclasses *Student* and *Employee* and a *Casual* subclass of *Employee*. Both the *Employee* and *Casual* classes have a *salaryIncrease* method. A 10% increase salary is given to ordinary employees while casual employees are given a 6% increase in salary.

Let us now look at how each of the mapping techniques would map the classes defined in Figure 1 to tables in a relational database.

**ORM1** All the subclass attributes of a root (super) class are stored in a single table. As given in Figure 2 the attributes of the *Employee*, *Student* and *Casual* classes are combined into a single table, called the *Entity* table. In this mapping technique the fields that are not relevant to particular row entry will be populated with Null values. For example, an ordinary *Student* will have a Null entry for the salary and casual fields corresponding to the *Employee* and *Casual* classes respectively. Unfortunately if there are

many sub-classes then most fields will be Null. Also, each time a new subclass is defined the existing table needs to be restructured which makes database maintenance expensive. Note that although property (P4) can be satisfied, properties (P1), (P2) and (P3) will fail.

<u>Entity</u> entity_id name address course salary hours
--

Figure 2: ORM1: One table per root class

**ORM2** One table is defined per concrete class but abstract classes do not correspond to a single table. Property (P3) holds for concrete classes and there is no need to restructure existing tables when a new sub-class is defined. However, all abstract class attributes must be duplicated in the relation of each concrete subclass so (P1) will fail. For example, in Figure 3 the fields of the abstract *Entity* class form part of the table mappings for the *Student*, *Employee* and *Casual* relational tables. In this technique the creation of a new subclass does not require restructuring of existing tables. However, it is not clear how to compile methods into queries, so (P4) is in doubt. For example, to compile the *salaryIncrease* method for employees requires being able to determine which employees in the table are ordinary employees and which are casual employees.

**ORM3** A single table is defined for each class, regardless of whether it is an abstract or a concrete class, as illustrated in Figure 4. A single identification key, given as the *entity\_id* is used as both a primary and foreign key to link the tables to replicate the class hierarchy.

Now (P1) and (P3) is satisfied, and additions to the class hierarchy do not require the restructuring of existing tables. Also, (P4) can be satisfied as the *Employee* and *Casual* classes are linked by the *entity\_id* field. However, in order to establish whether an employee is casual it is necessary to query the table for casual employees. This may prove expensive, especially if there are many sub-classes to consider. So, (P3) will only be satisfied if it is possible to create links to new tables, corresponding to new sub-classes, without re-compiling the existing programs. We shall return to this point below.

**ORM4** This technique is a variation of ORM3 in that a *discriminator field* is used to record the name of the sub-class (or table) to which the object belongs. The use of discriminators is well-known from the representation of variant records. A composite primary key field is defined allowing for the support of multiple inheritance. As can be seen in Figure 5 an additional discriminator field is incorporated in each class to relational table representation. The advantage of this approach over ORM3 is that, for the cost of an additional column per superclass one can check the status of an object (what kind of employee an entry is) from within the table corresponding to the super-class. As in ORM3 before, (P1) and (P3) are satisfied and (P4) is more easily satisfied, since it is only necessary to visit another table if the object is known to be in a

<u>Student</u>	<u>Employee</u>	<u>Casual</u>
entity_id	entity_id	entity_id
name	name	name
address	address	address
hours	salary	salary
		level

Figure 3: ORM2: One table per concrete class

<u>Entity</u>	<u>Student</u>
entity_id	entity_id
name	course
address	
<u>Employee</u>	<u>Casual</u>
entity_id	entity_id
salary	hours

Figure 4: ORM3: One table per class

sub-class. That is, the creation of sub-classes that are not used imposes almost no overhead on existing programs.

Summarising, it appears that ORM4 has the greatest promise, provided one is able to add method specialisations to existing compiled code (P2). More generally, there is the issue of how to convert the fields and methods of a class definition into relations and queries.

### 3 The Pattern Calculus

The pattern calculus (Jay 2004c) supports pattern-matching functions in which different cases may have different type specialisations of a common default as well as supporting specialisation through sub-typing. This section will identify some of the key features that are relevant to this paper.

The syntax of the *patterns* (meta-variable  $p$ ) and *raw terms* (meta-variable  $t$ ) of the pattern calculus is given by

$$\begin{aligned}
 p &::= x \mid c \mid p \ p \\
 t &::= x \mid c \mid t \ t \mid \text{at } p \ \text{use } t \ \text{else } t \mid \text{let } x = t \ \text{in } t
 \end{aligned}$$

The *variables* are represented by the meta-variable  $x$ . The *constructors* (meta-variable  $c$ ) are constants of the language which do not appear at the head of any evaluation rule. Other constants may be added if desired but their evaluation rules will not be considered explicitly in the formal development. The *application*  $s \ t$  applies the function  $s$  to its argument  $t$ . The novel term form is the *extension*  $\text{at } p \ \text{use } s \ \text{else } t$  where  $p$  is the *pattern*,  $s$  is the *specialisation* and  $t$  is the *default*. The let-term  $\text{let } x = s \ \text{in } t$  binds  $x$  to  $s$  in  $t$ . The declaration is recursive, in that free occurrences of  $x$  in  $s$  are bound to  $s$  itself.

Extensions combine abstraction over bound variables with a branching construction. For example, the  $\lambda$ -abstraction  $\lambda x.s$  is short-hand for the extension

$$\text{at } x \ \text{use } s \ \text{else } \text{err}$$

where  $\text{err}$  is some form of error term, e.g. a

<u>Entity</u>	<u>Student</u>
entity_id	Student_id
name	course
address	discriminator
discriminator	
<u>Employee</u>	<u>Casual</u>
Employee_id	Casual_id
salary	hours
discriminator	discriminator

Figure 5: ORM4: One table per class with a discriminator field

non-terminating expression (such as  $\text{let } x = x \ \text{in } x$ ) or an exception.

The *substitution*  $s\{u/x\}$  of a term  $u$  for a variable  $x$  in term  $s$  is defined in the usual way, as are bound variables and their  $\alpha$ -conversion. The *terms* are defined to be equivalence classes of raw terms under  $\alpha$ -conversion.

A *constructed term* is a term whose head is a constructor, i.e. a term which is either a constructor or of the form  $t_1 \ t_2$  in which  $t_1$  is constructed. Let  $c$  be a constructor. A term  $u$  *cannot become*  $c$  if it is either a constructed term other than  $c$  or an extension. A term  $u$  *cannot become applicative* if it is either a constructor or an extension.

$  \begin{aligned}  &(\text{at } x \ \text{use } s \ \text{else } t) \ t_1 > s\{t_1/x\} \\  &(\text{at } c \ \text{use } s \ \text{else } t) \ c > s \\  &(\text{at } c \ \text{use } s \ \text{else } t) \ t_1 > t \ t_1 \\  &\quad \text{if } t_1 \ \text{cannot become } c \\  &(\text{at } p_1 \ p_2 \ \text{use } s \ \text{else } t) \ (t_1 \ t_2) > \\  &\quad (\text{at } p_1 \\  &\quad \quad \text{use at } p_2 \ \text{use } s \ \text{else } \lambda y.t \ (p_1 \ y) \\  &\quad \quad \text{else } \lambda x,y.t \ (x \ y) \\  &\quad ) \ t_1 \ t_2 \ \text{if } t_1 \ \text{is a constructed term} \\  &(\text{at } p_1 \ p_2 \ \text{use } s \ \text{else } t) \ t_1 > t \ t_1 \\  &\quad \text{if } t_1 \ \text{cannot become applicative} \\  &\text{let } x = s \ \text{in } t > t\{s/x\}  \end{aligned}  $
--

Figure 6: Reduction rules for the pattern calculus

The *basic reduction rules* of the constructor calculus are given by the relation  $>$  in Figure 6. Let us consider the cases. Suppose that the pattern is a variable  $x$ . Specialisation is achieved by  $\beta$ -reduction, with the argument  $u$  being substituted for  $x$  in the specialisation. Suppose that the pattern is some constructor  $c$  and the argument is a constructed term  $u$ . If  $u$  is  $c$  then the specialisation is returned else the default is applied to  $u$ . Suppose that the pattern is an application  $p_1 \ p_2$  and the argument is a constructed term  $u$ . If  $u$  is an application  $u_1 \ u_2$  then specialisation tries to match  $p_1$  with  $u_1$  and  $p_2$  with  $u_2$ ; if either of these matches fails then evaluation reverts to applying the default to a reconstructed version of  $u_1 \ u_2$  (not  $u_1 \ u_2$  itself since this may require re-evaluation). If  $u$  is a constructor then the default is applied to it. Reduction of a let-term replaces the bound variable by its recursive definition.

## 4 The Discriminating Object Model

The approach adopted in the pattern calculus is to separate a class definition into a datatype declaration and some associated functions. The datatype definitions that correspond to the classes defined in Figure 1 are given in Figure 7. In turn, the datatype declarations for extensible classes employ a type variable to represent any additional fields that may be required. In particular, this type variable “is” the type of the discriminator field in the corresponding table. As can be seen by the use of the type variable X, Y, Z, and U in Figure 7 for any additional fields for the datatypes Entity, Student, Employee and Casual respectively.

```
datatype Entity X =
  Entity of X * int * string * string;;

datatype Student_Data Y =
  Student of Y * string;;

datatype Employee_Data Z =
  Employee of Z * float;;

datatype Casual_Data U =
  Casual of U * float;;
```

Figure 7: Class definitions as datatypes

In using the pattern calculus combined with ORM4 we can define one relational table per class. Therefore the creation of a new subclass does not require restructuring of objects. Method specialization, subtyping and type variable instantiation are all supported by the type safety of the Pattern Calculus. These coupled with ORM4 ensures that the implementation of an inherited method is unchanged by the creation of its subclass.

Let us now look at how our running example can be represented in this approach.

```
>-|> class Entity [a] {
  entity_id : int;
  name : string;
  address : string; }

rest_Entity : Entity[a] → a
entity_id : Entity[a] → int
name : Entity[a] → string
address : Entity[a] → string
```

Figure 8: Entity class in bondi

Given in Figure 8 is the class definition of Entity as defined in **bondi**, along with the corresponding **bondi** session output. Programs typed by the user follow the prompt >-|>. System responses are given in italic. The representation of Entity and its associated fields are displayed along with the additional *rest\_Entity* field. In the simplest case of an entity, the rest field is instantiated to be the unit type unit whose unique value can be called Null. In the database, the simplest case of an entity will correspond to a single row of data in the entity table where the discriminator field will be populated with Null.

In Figure 9 we see defined the Student class and its representation in **bondi**. A Student Entity has type Entity (Student\_Data Y) or just Entity (Student\_Data

```
>-|> class Student [a] extends Entity {
  course : string; }

rest_Student : Student[a] → a
course : Student[a] → string
```

Figure 9: Student class in bondi

Unit) if the student has no additional fields. In the database, a Student Entity will correspond to an entry in both the entity and student tables. The discriminator field in the entity table will be populated with the string Student while in the student table the discriminator field in will be Null. The entity\_id field is defined as both the primary and foreign key in the entity table as it’s also the primary key field in the student table, that is the field student\_id.

```
>-|> class Employee [a] extends Entity {
  salary : float;
  salaryIncrease() =
    {this.salary * 1.10} }

rest_Employee : Employee[a] → a
salary : Employee[a] → float
salaryIncrease : Employee[a] * unit → float
```

Figure 10: Employee class in bondi

In Figure 10 is defined the Employee class and its field representations in **bondi**. An Employee Entity has type Entity (Employee\_Data Y) or just Entity (Employee\_Data Unit) if the employee has no additional fields. In the database, an Employee Entity will correspond to an entry in both the entity and employee tables. Similarly to how the discriminator field is used to represent student entity entries the same occurs for employee entity entries. That is the discriminator field in the employee table will be populated with the string Employee while in the employee table the discriminator field in will be Null, for ordinary employees. Note that the use of the discriminator field combined with the entity\_id enables the support of multiple inheritance. A situation may arise where a student is also an employee. In this case they will have an entry in the entity, student and employee tables.

```
>-|> class Casual [a] extends Employee {
  hours : float;
  salaryIncrease() =
    { this.salary * 1.06} }

rest_Casual : Casual[a] → a
hours : Casual[a] → float
salaryIncrease : Casual[a] * unit → float
```

Figure 11: Casual class in bondi

Figure 11 defines the Casual class and its field representations in **bondi**. The type of a casual employee is some Entity (Employee\_Data (Casual\_Data U)). In the database, an ordinary casual employee will correspond to an entry in each of the entity, employee and casual tables. The discriminator field in the entity table will be given as employee, in the employee table it

will be given as casual and in the casual table will be given as Null. The same identity value will be used across all three tables as the `entity_id`, `employee_id` and `casual_id` respectively. Note that the type of a student or of a casual employee is always of the form Entity T for some type T, so that functions which act on arbitrary entities can always have the argument type Entity X. Again, each of these types has a single constructor. For example, that for entities is

```
Entity X : X * int * string * string → Entity X
```

Hence, a single pattern of the form Entity  $\lambda x$  where  $\lambda x$  is a binding variable will be able to match an arbitrary entity.

The notion of extensions in the pattern calculus allows us to specialise methods in the subclass without the need to recompile methods from the super class. This can best be exemplified through the salaryIncrease method defined in the running example. The salaryIncrease method defined in the Employee class is compiled to give a 10% increase in salary to all Employees as follows:

```
let (salaryIncrease :
  Entity (Employee_Data X) → float) =
  at Entity (Employee_Data (x,s),e,n,a)
  use s * 1.10
  else err;;
```

If the object is an employee entity that is of type Entity (Employee X) then a salary increase is calculated otherwise it will error.

This method defined in the superclass will translate into a stored procedure of something similar to the following code:

```
create procedure sp_salaryIncrease
begin transaction
  update Employee
  set salary = salary * 1.10
commit
```

Notice that since this method was defined for all employees the above sql code does not contain a where clause.

The specialisation of the salaryIncrease method in the Casual Employee class introduces a new case to the existing (compiled) method. Instead of recompiling the method for salaryIncrease, the pattern calculus defines a new method as follows:

```
let (newSalaryIncrease :
  Entity (Employee_Data X) → float) =
  at Entity (Employee_Data (Casual h y,s),
            e,n,a)
  use s * 1.06
  else salaryIncrease;;
```

```
let salaryIncrease = newSalaryIncrease;;
```

The new method encompasses the new case for Casual Employees as well as the existing code for ordinary Employees, without having to recompile the existing method.

We envisage that this method will translate into sql code such as the following:

```
create procedure sp_newSalaryIncrease
begin transaction
  if discriminator = "Casual" then
    update Employee
    set salary = salary * 1.06
    where discriminator = "Casual"
  else if discriminator = NULL then
    sp_salaryIncrease
endif
commit
```

```
sp_rename newSalaryIncrease salaryIncrease;;
```

Similarly to how we are able to translate the notion of the type variables in the programming language with discriminator fields in the database, we can see that notion of separate compilation can be applied to queries similarly to how they apply to methods. This is made possible due to the combination of a number of features.

The linking of the data model of the pattern calculus to that in the database through ORM4 allows for the support of the four desirable properties. Efficient data storage, one relation per class and modular compilation all are attained. The final property (P4) of compiled methods taking advantage of queries appears promising from the manual translation of the salaryIncrease method.

## 5 Conclusion and Future Work

The discriminating object model is able to represent objects and classes in both programming and relational databases. Each class is represented by a single table whose fields are those of the class plus a discriminator field to indicate which sub-class, if any, the object belongs to. The addition of new sub-classes does not change the relation itself, but merely creates new options for the discriminator. Hence, the relationship between the class and the database is stable, and so compilation can be modular. Also, the addition of the discriminator field has little or no impact on efficiency. This approach has been implemented in **bondi**.

The benefits (P1–P3) derived from the discriminating object model are not limited to a particular account of object-orientation and should prove useful for a variety of languages.

Future work will explore techniques for compiling simple methods into queries. This is particularly suited to development in the pattern calculus, as it supports an incremental approach to defining methods. Further, their execution can be driven by the discriminators without consulting any class hierarchy in the programming language. If successful, this would combine the expressive power of the object-oriented programming style with the efficiency of query languages.

## References

- Agrawal, R., Dar, S. & Gehani, N. H. (1993), The o++ database programming language: Implementation and experience, Technical Report 61-70, ATT Bell Laboratories.
- Ambler, S. W. (2003), *Agile Database Techniques: Effective Strategies for the Agile Software Developer*, Wiley, chapter 14.
- Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D. & Zdonik, S. (1994), 'The object-oriented database system manifesto'. <http://www-2cs.cmu.edu/People/clamen/OODBMS/Manifestor/htManifesto/node31.htm>.
- Bierman, G., Meijer, E. & Schulte, W. (2005), 'The essence of data access in *cw*. the power is in the dot!', <http://research.microsoft.com/emeijer/Papers/popl.pdf>. (Accepted for publication at ECOOP 2005).
- Bloom, T. & Zdonik, S. T. (1987), 'Issues in the design of object-oriented database programming languages', *OOPSLA '87 Proceedings* pp. 441–451.

- Breazu-Tannen, V., Buneman, P. & Ogori, A. (1991), Data structures and data types for object-oriented databases, *in* 'IEEE Data Engineering bulletin, Special Issue on Theoretical Foundations of Object-Oriented Database Systems', Vol. 14, IEEE, pp. 23–27.
- Fussell, M. L. (1997), Foundations of object relational mapping, Technical report, Chimu publications.
- Jay, C. B. (2004a), 'bondi', [www-staff.it.uts.edu.au/cbj/bondi](http://www-staff.it.uts.edu.au/cbj/bondi).
- Jay, C. B. (2004b), Methods as pattern-matching functions, *in* 'Foundations of Object-Oriented Languages, 2004: informal proceedings', p. 16 pp. <http://www.doc.ic.ac.uk/scd/FOOL11/patterns.pdf>.
- Jay, C. B. (2004c), 'The pattern calculus', *ACM Transactions on Programming Languages and Systems (TOPLAS)* **26**(6), 911–937.
- Jay, C. B. (2004d), 'Unifiable subtyping', [www-staff.it.uts.edu.au/cbj/Publications/unifiable-subtyping.pdf](http://www-staff.it.uts.edu.au/cbj/Publications/unifiable-subtyping.pdf).
- Keller, W. (1998), Object/relational access layers - a roadmap, missing links and more patters, *in* 'EuroPLoP'.
- Leontiev, Y., Oszu, M. & Szafron, D. (2002), 'On type systems for object-oriented database programming languages', *ACM Computing Surveys* **34**(4), 409–449.
- Remy, D. (1989), Typechecking records and variants in a natural extension of ML, *in* L. Cardelli, ed., 'Proceedings of POPL'98, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages', ACM.
- Richardson, J. E., Carey, M. J. & Schuh, D. T. (1989), The design of the e programming language, Technical report, University of Wisconsin.