Database Programming in Machiavelli – a Polymorphic Language with Static Type Inference*

Atsushi Ohori Peter Buneman Val Breazu-Tannen

Department of Computer and Information Science
University of Pennsylvania
200 South 33rd Street
Philadelphia, PA 19104-6389

Abstract

Machiavelli is a polymorphically typed programming language in the spirit of ML, but supports an extended method of type inferencing that makes its polymorphism more general and appropriate for database applications. In particular, a function that selects a field f of a records is polymorphic in the sense that it can be applied to any record which contains a field f with the appropriate type. When combined with a set data type and database operations including join and projection, this provides a natural medium for relational database programming. Moreover, by implementing database objects as reference types and generating the appropriate views — sets of structures with "identity" — we can achieve a degree of static type checking for object-oriented databases.

1 Introduction

The term "impedance mismatch" has been coined [Mai89] to describe the phenomenon that the data types available in a programming language do not usually match the structures provided in a database system. This problem will be painfully familiar to anyone who has used a high-level programming language to communicate with a database. This mismatch is particularly unfortunate when database applications programming cannot make full use of the rich, statically checked type systems available in a number of modern programming languages. Database schemas can be large and complex structures, and our experience is that most programming errors in database applications would show up as *type errors* were the schema a part of the type structure of the program. Thus a type system in which such errors can be anticipated by a static analysis of the program is, we believe, a prerequisite for a good database programming language.

The designers of certain database programming languages, notably Pascal-R [Sch77] and Galileo [ACO85] have recognized this mismatch problem and have implemented languages in which a database can be directly represented in the type system of the language. Type checking in both these languages is static and the database types are relatively simple and elegant extensions to the existing type systems of the programming languages on which they are based. However, in these languages it is sometimes difficult to write the kinds of "generic" or "polymorphic" programs that are desirable for many database applications. Contrast this with persistent languages such as PS-algol [ABC*83] and some of the more recent object-oriented database languages such as Gemstone [CM84], EXODUS [CDJS86] and Trellis-Owl [OBS86] where, if it is at all possible to write generic code, some dynamic type-checking is required. See [AB87] for a survey.

In this paper we describe how a polymorphic type system, in conjunction with suitable data types for sets and records can be used to achieve a natural representation for databases within a programming language. The form of polymorphism available in languages such as ML [HMT88] is intimately connected with a type inference system; and we regard type inference as a strategy for re-

* Appeared in *Proceedings of ACM SIGMOD Conference on Management of Data*, Portland, Oregon, May, 1989. Pages 46 – 57.

This research was supported in part by grants NSF IRI86-10617, ONR NOOO-14-88-K-0634, ARO DAA6-29-84-k-0061 This research was done during the first author was on leave from Oki Electric Industry. Atsushi Ohori's current address: Oki Electric Industry, Kansai Laboratory, Crystal Tower, 1-2-27 Shiromi, Chuo-ku, Osaka 540, JAPAN.

alizing this polymorphism. In addition type inference has the obvious advantage that it can "discover" the generic properties of some piece of code, which would otherwise be both difficult and time-consuming to write down explicitly. These ideas are embodied in Machiavelli, an experimental programming language in the tradition of ML, developed at University of Pennsylvania. A prototype implementation has been developed that demonstrates most of the material presented here with the exception of reference types, and some form of persistence. We will show how Machiavelli's type system provides a natural representation of relational databases moreover, when combined with reference types we obtain representations similar to those used in object-oriented databases. Our hope is that Machiavelli (or some language like it) will provide a framework for dealing uniformly with both relational and object-oriented databases.

Let us illustrate the flavor of programming in Machiavelli with an example. Consider a function which takes a set of records (i.e. a relation) with Name and Salary information and returns the set of all Name values which correspond to Salary values over 100K. For example, applied to the relation

```
{[Name = "Joe", Salary = 22340],
 [Name = "Fred", Salary = 123456],
 [Name = "Helen", Salary = 132000]}
```

this function should yield the set {`"Fred", "Helen"`}. Such a function is written in Machiavelli (whose syntax mostly follows that of ML [HMT88]) as follows

```
fun Wealthy(S) = select x.Name
                 where x <- S
                 with x.Salary > 100000;
```

The `select ... where ... with ...` form is simple syntactic sugar for more basic Machiavelli program structure (see section 2).

Although no data types are mentioned in the code, Machiavelli *infers* the type information

```
Wealthy: {[("a) Name:"b,Salary:int]} -> {"b}
```

by which it means that `Wealthy` is a function that takes a homogeneous set of records, each of type `[("a) Name : "b, Salary : int]`, and returns a homogeneous set of values of type `"b`, where `("a)` and `"b` are *type variables.* `"b` represents an arbitrary type on which equality is defined. `("a)` represents an arbitrary extension to the record structure that does not contain `Name` and `Salary` fields; this is superficially similar to the "row variables" in [Wan87]. `"b` and `("a)` can be *instantiated* by any type and record extension satisfying the above conditions. Consequently, Machiavelli will allow `Wealthy` to be applied, for example, to relations of type

```
{[Name: string, Age:int, Salary: int]}
```

and also to relations of type

```
{[Name: [First: string, Last: string],
  Weight: int, Salary:int]}.
```

The function `Wealthy` is *polymorphic* with respect to the type `"b` of the values in the Name field (as in ML) but is also polymorphic with respect to extensions `("a)` to the record type `[Name:"b ,Salary: int]` In this second form of polymorphism, `Wealthy` can be thought of as a "method" in the sense of object-oriented programming languages where methods associated with a class may be inherited by a subclass, and thus applied to objects of of that subclass.

For the purposes of finding a typed approach to object-oriented programming, Machiavelli's type system has similar goals to the systems proposed by Cardelli and Wegner [Car84a, CW85]. However, there are important technical differences, the most important of which is that database values have *unique types* in Machiavelli while they can have multiple types in [Car84a]. Based on the idea suggested in [Wan87], Machiavelli achieves the same goals of representing objects and inheritance (see also [Sta88, JM88] for related studies). These differences allow Machiavelli to overcome certain anomalies (see [OB88], which also gives details of the underlying type inference system).

Another important extension to these type systems for objects and inheritance is that Machiavelli uniformly integrates *set types* and a number of operations on complex objects and objects with "identity" which are essential to database programming. This paper describes how database structures are naturally represented in the type system of Machiavelli and how the type system supports powerful yet type-safe programming for databases. In particular we show that by exploiting type inference we are able to achieve what we believe to be the desirable features of programming with object-oriented databases or "semantic" data models [AH87], by the use of coercions or "views".

Section 2 discusses the use of sets in a programming language and, in particular, how higher-order relations are treated. Section 3 contains a description of the language itself. Section 4 and 5 respectively show how Machiavelli can be used to represent relational and object oriented databases. Section 6 discusses the further work that is needed to make the language useful in dealing with external databases. Limitations of space make it difficult to cover the language fully; the authors intend to present a more detailed description in another paper.

2 Sets and Relations

If relations are to be properly incorporated into a polymorphic programming language, it is clear that we must break with the first-normal-form assumption that underlies most implemented relational database systems and most of the traditional theory of relational databases. Indeed, the type

```
{[Name: [First: string, Last: string],
    Salary:int]}
```

is the type of a "non-first-normal-form" relation in which the Name field is itself a record type. In this case it is a relatively easy matter to flatten such a relation into a first-normal form relation but were the Name field to be a reference to a name or to be a set of names [Eli82], we could not perform such a flattening operation without modifying the intended "semantics" of the database.

A set type $\{\tau\}$ in Machiavelli can be defined over any data type τ for which equality is available. We shall call such types *description types*; they are similar to "equality types" in ML, but have more operations available. There are four basic functions and values associated with sets:

- `{}` — the empty set
- `{x}` — the singleton set constructor
- `union` — set union
- `hom` — homomorphic extension

of these operations `hom` requires some explanation. hom is a primitive function in Machiavelli similar to the "pump" operation in FAD [BBKV88] and the "fold" or "reduce" of many functional languages whose definition is

$$\begin{aligned}&\texttt{hom}(f, op, z, \{\}) = z\\ &\texttt{hom}(f, op, z, \{x_1, x_2, ..., x_n\}) =\\ &\qquad op(f(x_1), op(f(x_2), ..., op(f(x_n), z)...))\end{aligned}$$

In general the result of this operation will depend on the order in which the elements of the set are encountered; however if *op* is an associative commutative operation and f has no side-effects, then the result of `hom` will be independent of the order of this evaluation. When this happens we shall call the application of `hom` *proper*. Machiavelli cannot guarantee that every application of `hom` is proper; indeed improper applications of *hom* are frequently useful. However proper applications are what we mean by functions on sets, and they also have the property of being computable in parallel.

There is an alternative form of `hom`, `hom*` that applies to non- empty sets and does not require the argument z. Thus

$$\begin{aligned}&\texttt{hom*}(f, +, \{x_1, x_2, ..., x_n\}) =\\ &\qquad f(x_1) + f(x_2) + ... + f(x_n)\end{aligned}$$

When z is an identity for *op*, `hom` behaves as `hom*` on non-empty sets.

For example the following useful functions can be defined using `hom`:

```
  fun map(f,S) =
    hom((fn(x) => {f(x)}), union, {}, S)
  fun filter(p,S) =
    hom((fn(x) => if p(x) then {x} else {}),
 union,
 {},
 S)
```

Here, `map(f,S)` is the set of results of applying `f` to each member of `S` — the direct image of `S` by `f`, and `filter(p,S)` is the set of elements of `S` that satisfy `p`. Notice that both of these applications are proper. For readers unfamiliar with the syntax of ML, `fun ... = ...` is a function definition, and `(fn ... => ...)` is a lambda abstraction (anonymous function definition).

In addition to these examples `hom` can be used to define set intersection, membership in a set, set difference, the cartesian product (`prod`) of sets and the powerset (the set of subsets) of a set. Also, the form

```
    select E
    where x1 <- S1,
          x2 <- S2,
          ...
          xn <- Sn
    with P
```

which is provided in the spirit of relational query languages and the "comprehensions" of Miranda [Tur85], can be implemented as

```
  map((fn(e,p) => e),
      filter((fn(e,p) => p),
     map((fn(x1,x2, ...xn) => (E,P)),
 prod(S1,S2, ... Sn))))
```

Where `map`, `filter` and `prod` are the functions we have just described, and `(E,P)` is a pair of values (implemented in Machiavelli as records).

However, it should be noted that unlike Miranda, which operates on streams and unlike most relational systems, which operate on bags or lists, Machiavelli's sets are sets in the mathematical sense of the term.

We now turn to operations on records. The first primitive operation on records is *projection* which "throws away" certain information. For example

```
  project([Name="Joe", Age=21, Salary=22340],
  [Name:string, Salary:int])
```

is `[Name ="Joe", Salary=22340]`. In this case projection has preserved the Name and Salary fields. A more complicated projection is

```
  project([Name=[First="Joe", Last="Doe"],
   Salary=12345],
  [Name:[Last:string]])
```

In general, if r is any value of type τ and σ corresponds to a *substructure* of τ then `project`(r, σ) is well defined. The substructure relationship is described more fully in the next section. `project` is defined for all description types in the language, but except for types that contain records it is an uninteresting operation. For example `project(3,int)` is simply 3. For general description types, projection is "lifted" according to their structures. As an example, a projection on sets `project`$(S, \{\tau\})$ is is equivalent to `map((fn(x)=>project(x,`τ`)),`S`)`. When the set type $\{\tau\}$ is a set of records this is a generalization of relational projection.

Two records are *consistent* if they are both projections of some common record. For example `[Name = [First = "Joe"], Age=21]` and `[Name = [Last = "Doe"]]` are consistent, while `[Name = "Joe", Age = 21]` and `[Name = "Sue"]` are inconsistent.

Machiavelli has a predicate `con` which decides whether two records are consistent and an operation `join`, which "joins" two records when they are consistent as shown in the following example:

```
join([Name=[First="Joe"], Age=21],
     [Name=[Last="Doe"]]) =
[Name=[First="Joe", Last="Doe"], Age = 21]
```

For `join` and `con` to be well-defined, they must have consistent types, thus

```
join([Name=[First="Joe"], Age=21], [Name="Joe"])
```

will cause a (static) type error. The types for `con` and `join` are explained in section 3.

Based on a general property of database sets studied in [BJO89], where a natural join for higher-order relations was described, `join` can again be "lifted" to sets. When the join is applied to two sets of records (higher order relations), it results in the natural join of the two relations.

A useful property of join is that it coincides with intersection when applied to two sets of the same base type, such as {`int`}. It also provides an interesting and useful generalization of intersection when applied to sets of "objects". This is discussed in section 5.

3 The Language Machiavelli

Machiavelli is an extension of the programming language ML. While preserving ML's features of complete static type inference and polymorphism, it extends ML's type system with variants, sets and general recursive types and supports a number of operations that are useful for databases and object-oriented programming including join and projection generalized to arbitrary complex descriptions. This extension also eliminates ML's severe restriction on functions manipulating records and mandatory requirement of recursive type declarations. Here we give an overview of the language with an emphasis on the features that are relevant to database programming. Formal properties underlying the language are described in [OB88, Oho88, Oho89].

3.1 Types

Let l range over a set of labels. The types of Machiavelli (ranged over by τ) are represented by the following syntax:

$$\begin{aligned}\tau \quad ::= \quad & \mathit{unit} \mid \mathit{int} \mid \mathit{bool} \mid \mathit{string} \mid \mathit{real} \mid \tau \rightarrow \tau \mid \\ & [l : \tau, \ldots, l : \tau] \mid \langle l : \tau, \ldots, l : \tau \rangle \mid \{\tau\} \mid \\ & \mathit{ref}(\tau) \mid \mathit{rec}\ v.\ \tau(v)\end{aligned}$$

$[l : \tau, \ldots, l : \tau]$ represent record types and $\langle l : \tau, \ldots, l : \tau\rangle$ represent variant types. $\mathit{rec}\ v.\ \tau(v)$ represents recursive types where $\tau(v)$ is a type expression possibly containing the symbol v. Formally, the set of types of Machiavelli is defined as the set of labeled *regular trees* [Cou83] constructed from base types and type constructors. Infinite trees correspond to recursive types.

A type τ is a *description type* if it does not contain a function type constructor $\rightarrow$ outside of the scope of any *ref* constructors. On description types, equality, as well as database operations, are available. We use $\delta, \delta_1, \ldots$ for description types.

For convenience, we assume special labels #1, #2, ... and write $\tau_1 * \tau_2 * \cdots * \tau_n$ for $[\#1 : \tau_1, \ldots, \#n : \tau_n]$ and $\tau_1 + \tau_2 + \cdots + \tau_n$ for $\langle \#1 : \tau_1, \ldots, \#n : \tau_n\rangle$. The following are examples of types representable in Machiavelli:

$$\begin{aligned}\mathit{person} &= [\mathit{Name} : \mathit{string}, \mathit{Age} : \mathit{int}] \\ \mathit{personObj} &= \mathit{ref}([\mathit{Name} : \mathit{string}, \mathit{Age} : \mathit{int}]) \\ \mathit{intlists} &= \mathit{rec}\ v.\ (\mathit{unit} + (\mathit{int} * v))\end{aligned}$$

3.2 Expressions

Let c, x, δ stand respectively for constants, variables and description types. Expressions are defined by the following syntax:

$$\begin{aligned}e \quad ::= \quad & c \mid x \mid e(e) \mid (\mathit{fn}(x, \ldots, x) \Rightarrow e) \mid \\ & \mathit{if}\ e\ \mathit{then}\ e\ \mathit{else}\ e \mid \\ & [l = e, \ldots, l = e] \mid e.l \mid \mathit{modify}(e, l, e) \mid \\ & (l\ \mathit{of}\ e) \mid \\ & (\mathit{case}\ e\ \mathit{of}\ l\ \mathit{of}\ x \Rightarrow e, \ldots, l\ \mathit{of}\ x \Rightarrow e) \mid \\ & (\mathit{case}\ e\ \mathit{of}\ l\ \mathit{of}\ x \Rightarrow e, \ldots, \mathit{other} \Rightarrow e) \mid \\ & \{e, \ldots, e\} \mid \mathit{union}(e, e) \mid \mathit{hom}(e, e, e, e) \mid \\ & \mathit{ref}(e) \mid (!e) \mid e := e \mid \\ & \mathit{con}(e, e) \mid \mathit{join}(e, e) \mid \mathit{project}(e, \delta) \mid \\ & \mathit{let}\ x = e\ \mathit{in}\ e \mid \mathit{rec}(x, e)\end{aligned}$$

where $e.l$ is field selection from a record, $(l\ \mathit{of}\ e)$ is injection to a variant, $\mathit{ref}(e)$ is reference creation, $(!e)$ is dereference, and $\mathit{rec}(x, e)$ is a recursive description construction. $\mathit{modify}(e_1, l, e_2)$ modifies the l-field of the record expression e_1 with e_2. It is important to note that *modify* does *not* have a side-effect. It is a function that returns a modified copy of its argument. The variable that appears in $l\ \mathit{of}\ x \Rightarrow e$ in *case* construction is bound to the actual value of l-variant (when selected) in e. *hom* is an operation we have already described, and $\mathit{con}, \mathit{join}, \mathit{project}$ are described in the next section.

3.3 Type Inference and Evaluation

One important feature of Machiavelli, inherited from ML, is the static type inference. The type system statically determines whether a given program is type correct. Moreover, by using the inference strategy described in [OB88] which is an extension of Milner's method [Mil78] for ML, Machiavelli's type system finds a *principal conditional type-scheme* for any type correct program. Rather than describe the strategy in detail we give some program examples.

Machiavelli can be used interactively, and the top level input is either a value binding of the form

```
-> val x = EXPR ;
```

a function definition of the form,

```
-> fun f(x,...,x) = EXPR ;
```

or an expression. `->` is the input prompt. The following is a very simple session in Machiavelli:

```
-> 1;
>> val it = 1 : int
-> fun id(x) = x;
>> val id = fn : 'a -> 'a
-> id(1);
>> val it = 1 : int
```

`>>` is Machiavelli's output prefix, and `it` is a name for the result of evaluation of an expression. `'a` is a type variable representing an arbitrary type. The function `id` is a typical *polymorphic* function. It can be applied to any value of of any type τ and will return a value of the same type τ. This mechanism attains much of the flexibility of untyped languages without sacrificing the benefit of static type-checking.

In the example above Machiavelli behaves exactly as ML, however it is also possible to infer types for expressions involving records and variants. Figure 1 shows some examples. In the notation `[('a)` $l_1 : \tau_1, ..., l_n : \tau_n$`] ('a)` stands for any sequence of label-type pairs that does not contain the labels $l_1, ..., l_n$. A similar convention, `<('a) ...>` is used for variants. Such type expressions are inferred by the type inferencing method so that, for example, the function `phone` can be applied to any record that contains a `Status` field of either an `Employee` variant of any record containing an `Extension` field or to a `Consultant` variant of any record containing a `Telephone` field. Moreover, Machiavelli always finds the exact result types of such applications. This eliminates the problem of *loss of type information*, which was observed, but not eliminated, in the language FUN [CW85] (see [OB88] for an analysis of this problem). Also note that the functions involving field modification such as `increment_age` are not well treated in FUN.

Next we show how Machiavelli infers types of programs containing `con, join` and `project`. These three operations are defined on arbitrary description types. `con`(d_1, d_2) checks the consistency of two descriptions and `join`(d_1, d_2) computes the combination of the two descriptions if they are consistent. Projection is generalized to a projection on arbitrary description types. If δ is a description type, then `project`(d, δ) is the projection of d onto δ. In order to infer correct types for these operations and support them as polymorphic operations that work uniformly on arbitrary complex descriptions, we introduce the *information ordering* $\leq$ on description types. Let $\delta, \delta_1, \ldots$ denote description types. $\delta_1 \leq \delta_2$ iff δ_1 can be obtained from δ_2 by deleting zero or more record labels that appear outside of scopes of *ref* type constructors. On finite description types, $\leq$ is equivalent to the following inductive definition:

$$\begin{array}{rcl} b & \leq & b \ (b \text{ a base type}) \\ \{\delta\} & \leq & \{\delta'\} \\ & & \text{if } \delta \leq \delta' \\ [l_1 : \delta_1, \ldots, l_n : \delta_n] & \leq & [l_1 : \delta'_1, \ldots, l_n : \delta'_n, \ldots] \\ & & \text{if } \delta_i \leq \delta'_i \text{ for each } i \\ < l_1 : \delta_1, \ldots, l_n : \delta'_n > & \leq & < l_1 : \delta'_1, \ldots, l_n : \delta_n > \\ & & \text{if } \delta_i \leq \delta'_i \text{ for each } i \\ ref(\tau) & \leq & ref(\tau) \end{array}$$

$\tau_1 \leq \tau_2$ captures the intuitive notion that τ_2 is a *bigger* structure than τ_1. Note that $\leq$ is a partial ordering. We will denote the least upper bound of δ_1, δ_2, whenever it exists, by $\delta_1 \sqcup \delta_2$. With this `con,join,project` are given the following polymorphic types.

$$\begin{array}{rcl} con & : & (\delta_1 * \delta_2) \rightarrow bool \ \text{ if } \delta_1 \sqcup \delta_2 \text{ exists} \\ join & : & (\delta_1 * \delta_2) \rightarrow \delta_1 \sqcup \delta_2 \ \text{ if } \delta_1 \sqcup \delta_2 \text{ exists} \\ project_\delta & : & \delta' \rightarrow \delta \ \text{ if } \delta \leq \delta' \end{array}$$

For their precise typing rules and semantics, readers are referred to [OB88, Oho88] respectively. The following example shows how expressions involving join and projection are typed by using the information ordering on description types.

```
-> val fun Join3(x,y,z) = join(x,join(y,z));
>> val Join3 = fn : ("a * "b * "c) -> "d
   where { "d = "a lub "e, "e = "b lub "c }
-> Join3([Name="Joe"],[Age=21],[Office=27]);
>> val it = [Name="Joe",Age=21,Office=27]
     : [Name:string,Age:int,Office:int]
-> project(it,[Name:string]);
>> val it = [Name="Joe"] : [Name:string]
```

`"a` represents arbitrary description types and `"d = "a lub "e` in the `where` clause represents the condition that the instance of `"d` must be the least upper bound of the instances of `"a,"e` under the information ordering. `Join3` computes the join of three (joinable) complex objects. If `r1,r2,r3` are three joinable flat relations, then `Join3(r1,r2,r3)` is exactly the natural join of the three. As seen in the example, Machiavelli always maintains if-and-only-if conditions associated with operators such as `join` that do not have a conventional principal type-scheme. This mechanism makes type inference complete.

```
-> val joe  = [Name="Joe", Age=21,
               Status=(Consultant of [Address="Philadelphia", Telephone=2221234])];
>> val it = [Name="Joe", Age=21,
             Status=(Consultant of [Address="Philadelphia", Telephone=2221234])]
   : [Name:string, Age:int,Status:<('a) Consultant:[Address:string,Telephone:int]>]
-> fun phone(x) = (case x.Status of Employee of y => y.Extension,
                                    Consultant of y => y.Telephone);
>> val phone = fn
   : [('a) Status:<Employee:[('b) Extension:'d], Consultant:[('c) Telephone:'d]>] -> 'd
-> phone(joe);
>> val it = 2221234 : int
-> fun increment_age(x) = modify(x, Age, x.Age + 1);
>> val increment_age = fn : [('a) Age:int] -> [('a) Age:int]
-> increment_age([Name="John",Age=21]);
>> val it = [Name="John",Age=22] : [Name:string,Age:int]
```

Figure 1: Some Simple Machiavelli Examples

```
-> parts;
>> val it =
     {[Pname="bolt",P#=1,Pinfo=(BasePart of [Cost=5])], ...
      [Pname="engine",P#=2189,
       Pinfo=(CompositePart of [SubParts={[P#=1,Qty=189], ...}, AssemCost=1000])], ...}
    : {[Pname:string,P#:int,
        Pinfo:<BasePart:[Cost:int],
               CompositePart:[SubParts:{[P#:int,Qty:int]},AssemCost:int]>]}
 -> suppliers;
 >> val it =
      {[Sname="Baker",S#=1,City="Paris"], ... } : {[Sname:string,S#:int,City:string]}
 -> supplied_by;
 >> val it = {[P#=1,Suppliers={[S#=1],[S#=12],....}], ... }
    : {[P#:int,Suppliers:{[S#:int]}]}
```

Figure 2: A Part-Supplier Database in Generalized Relational Model

```
(* Select all base parts *)
-> join(parts,{[Pinfo=(BasePart of [])]});
>> val it = {[Pname="bolt", P#=1, Pinfo=(BasePart of [Cost=5])], ...}
    : {[Pname:string,P#:int,
        Pinfo:<BasePart:[Cost:int],
               CompositePart:[SubParts:{[P#:int,Qty:int]},AssemCost:int]>]}
(* List part names supplied by "Baker" *)
-> select x.Pname
   where x <- join(parts,supplied_by)
   with Join3(x.Suppliers,suppliers,{[Sname="Baker"]}) <> {};
>> {"bolt",...} : {string}
```

Figure 3: Some Simple Queries

4 Generalized Relational Models

Machiavelli supports arbitrarily complex structures that can be constructed with records, variants and sets. This allows us to define directly in Machiavelli databases supporting complex structures including non-first-normal form relations, nested relations and complex objects. Figure 2 shows an example of a database containing non-flat records, variants, and nested sets. With the availability of a generalized join and projection, we can immediately write programs that manipulate such databases. Figure 3 show some simple query processing for the database example in figure 2. From this example, one can see that join and projection in Machiavelli faithfully extends the natural join and projection in the relational model to complex objects.

The most important feature of Machiavelli is that these data structures and operations are all "first-class citizens" in the language. This eliminates the problem of "impedance mismatch" we discussed in the introduction. Data and operations can be freely mixed with other features of the language including recursion, higher-order functions, polymorphism. This allows us to write powerful query processing programs relatively easily. The type correctness of programs is then automatically checked at compile time. Moreover, the resulting programs are in general polymorphic and can be shared in many applications. Figure 4 shows a simple implementation of a polymorphic transitive closure function. By a using renaming operation, this function can be used to compute the transitive closure of any binary relation. Figure 5 shows query processing on the example database using polymorphic functions. The function `cost` taking a part record as argument computes the total cost of the part. Without proper integration of the data model and programming language, defining such a function and checking type consistency is a rather difficult problem. It should also be noted that functions `cost` and `expensive_parts` are polymorphic and can be applied to many different types sharing the same common structures. This is particularly useful when we have several different parts databases with the same structure of cost information. Even if the individual databases differ in the structure of other information, these functions can be shared by all those databases.

5 Manipulation of Object-Oriented Databases

In this section we first show how to represent object-oriented databases within Machiavelli's type system and then suggest how Machiavelli might be used to communicate with external, object-oriented databases. We believe that the notion of "objects" can be accurately captured by reference types. References support sharing of structure and mutability. For example, if we define a department record

```
val d = ref([Dname="Sales", Building=45]);
```

and from this we define two employee records

```
val emp1 = ref([Name = "Jones",
Department = d]);
val emp2 = ref([Name = "Smith",
Department = d]);
```

then an update to the building of the department as seen from `emp1`

```
let val d = (!emp1).Department
in d:=modify(!d, Building, 67)
end;
```

will be reflected in the department as seen from `emp2`. Another important property of reference types is that they support "object identity": two references are equal only if they are the result of the same invocation of the function `ref` which creates references. For example, `ref(3) = ref(3)` is *false*, the two applications of `ref` generate different (unequal) references.

A second property of object oriented databases has to do with the connection between classes and extents. When we say an Employee ISA Person, there are at least two things we could understand by this relationship. One of them is that the "methods" that apply to a Person object can also be applied to an Employee; another is that the database contains a set of objects and that the set of Employee objects is a subset of the set of Person objects. Now there is no *a priori* reason why these two definitions of ISA should have anything to do with each other. Indeed, if we think of Person and Employee as types and objects as values, the second (extensional) definition of ISA is excluded because database values in Machiavelli have a unique type; moreover, as we have already noted, there is a problem with the assumption that a database value can have multiple types.

Nevertheless it seems to be a desideratum of object-oriented databases that these two definitions of ISA should be coupled: if you select the Employee objects from the database, you get a subset of the Person objects in the database and the methods available for Employee objects form a superset of the methods available for Person objects. But note that this argument only asks that the two definitions of ISA are coupled *relative to some database*; we see no reason for having a distinguished extent associated with each class, as happens in many database programming languages. Among other things, this restriction implies that a program written in such languages cannot deal comfortably with more than one database at a time.

The way we capture this idea in Machiavelli is through coercions or *views*. The type of an object will, in general, be a reference to a rather complicated type, say

```
-> fun Closure R =
       let val r = select [A=x.A,B=y.B]
               where x <- R, y <- R
               with (x.B = y.A) andalso not(member([A=x.A,B=y.B],R))
       in if r = {} then R else Closure(union(R,r))
       end;
>> Closure = fn : {[A:"a,B:"b]} -> {[A:"a,B:"b]}
```

Figure 4: A Simple Implementation of Polymorphic Transitive Closure

```
(* function computes the total cost of a part *)
-> fun cost(p) =
     (case p.Pinfo of
        BasePart of x=>x.Cost,
        CompositePart of x=>
          x.AssemCost +  hom((fn(y)=>y.SubpartCost * y.Qty),+,0,
                           select [SubpartCost=cost(z),Qty=w.Qty]
                           where w <- x.SubParts, z <- parts
                           with z.P#=w.P#));
>> val cost = fn
   : [('a) Pinfo:<BasePart:[('c) Cost:int],
                  CompositePart:[('d) SubParts:{[('e) P#:int,Qty:int]},AssemCost:int]>]
        -> int

(* select names of "expensive" parts *)
-> fun expensive_parts(partdb,n) =
     select x.Pname
     where x <- partdb
     with cost(x) >n;
>> val expensive_parts = fn :
   : ({[('a) Pinfo:<BasePart:[('c) Cost:int],
                    CompositePart:[('d) SubParts:{[('e) P#:int,Qty:int]},AssemCost:int]>]},
      int) -> {string}

-> expensive_parts(parts,1000);
>> val it = {"engine",...} : {string}
```

Figure 5: Query Processing Using Polymorphic Functions

PersonObj. A database (or a part of it) will consist of a set D of such objects, i.e. a value of type {PersonObj}. A view of D is a set of relatively simple records in which we "reveal" a part of the structure of each member of D in a fashion that allows us to exploit the relational operations we have already developed. For example, {[Name: string, Id: PersonObj]} and {[Name: string, Age: int, Id:PersonObj]} are both views of set D. But notice that within these records we have kept a distinguished Id field that contains the object itself, and this field, being a reference type can also be treated as an "identity" or key when we have a set of objects. Because of the presence of this field, we can perform generalized set operations on views even though they are of different type. In fact we have already seen one such operation, the natural join. When applied to views it is an operation that takes the intersection of sets of identities, but produces a result that has a join type and gives us the union of the "methods". In fact we shall simply define a *class* as any record type that contains an Id field, which will be assumed to be some reference type.

As an example, a part of the database could be a collection of "person" objects modeling the set of persons in a university. Among persons, some are students and others are employees. Such subsets naturally form a taxonomic hierarchy or *class* structure. Figure 6 shows a simple example. Note that the arrows not only represent inheritance of properties but also actual set inclusions; they also run opposite to the information ordering described earlier.. We use variant types to represent structures of objects that share common properties (e.g. being a person) but differ in special properties. The example is then represented by the types shown in figure 7. We should emphasize that the type names *PersonObj*, *Person*, etc are not names in Machiavelli, they are just convenient shorthands used in some of the examples that follow. The

```
PersonObj = ref([Name: string, Salary : <None: unit, Value: int>,
                 Advisor : <None: unit, Value: PersonObj>,
                 Class : <None:unit, Value: string>]);
Person = [Name: string, Id: PersonObj];
Student = [Name: string, Advisor: PersonObj, Id: PersonObj]
Employee = [Name: string, Salary: Integer, Id: PersonObj]
TeachingFellow = [Name: string, Salary: Integer, Advisor: PersonObj,
                 Class: String, Id: PersonObj]
```

Figure 7: Some Machiavelli Types

```
fun PersonView(S) = select [Name=(!x).Name, Id=x]
                    where x <- S
                    with true;
fun EmployeeView(S) = select [Name=(!x).Name, (Salary=(!x).Salary as Value), Id=x]
                      where x <- S
                      with (case (!x).Salary of Value of _ => true, other => false);
fun StudentView(S) = select [Name=(!x).Name, (Advisor=(!x).Advisor as Value), Id=x]
                     where x <- S
                     with (case (!x).Advisor of Value of _ => true, other => false);
fun TFView(S) = select join(x,[Class=(!x).Class as Value]
                where x <- join(StudentView(S),EmployeeView(S))
                with (case (!x).Class of Value of _ => true, other => false);
```

Figure 8: Definition of Views

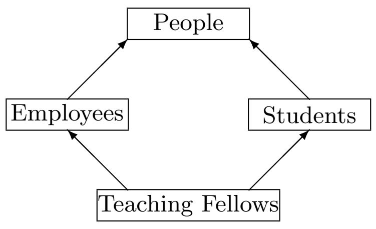

Figure 6: A Simple Class Structure

reference type *PersonObj* is the type of a person object. The type *Person*, *Employee* and *TeachingFellow* are types of person objects *viewed* as persons, employees and teching fellows respectively. For example, a person object is viewed as (or more precisely can be coerced to) an employee if it has name and salary attributes. A database would presumably contain a set of person objects, i.e. a set of type {`PersonObj`}, and views of any set of this type can be constructed in Machiavelli by the definitions shown in figure 8. where `(e as l)` is a shorthand for `(case e of l of x => x, other raise Error)`. The types inferred for these functions will be quite general, but the following are the instances that are important to us in the context of this example.

```
PersonView : {PersonObj} -> {Person}
EmployeeView : {PersonObj} -> {Employee}
StudentView : {PersonObj} -> {Student}
TFView : {PersonObj} -> {TeachingFellow}
```

In the definition of `TFView`, the join of two views models both the intersection of the two classes and the inheritance of methods. If τ_1, τ_2 are types of classes, then $\tau \leq \sigma$ implies that $\texttt{Project}(View_\sigma(S), \tau) \subseteq View_\tau(S))$ where $View_\tau$ and $View_\sigma$ denote the corresponding viewing functions on classes τ and σ. This property guarantees that the join of two views corresponds to the intersection of the two. The property of the ordering on types and Machiavelli's polymorphism also supports the inheritance of methods. For example, suppose we have a database `persons`. Then `join(StudentView(persons), EmployeeView(persons))` always represents the set of objects that are both student and employee. Moreover, methods defined on `StudentView(persons)` and `EmployeeView(persons)` are automatically inherited by Machiavelli's type inference mechanism. As an example of inheritance of methods, the function `Wealthy`, as defined in the introduction, has type `{[("a) Name:"b,Salary:int]} -> {"b}`, which is applicable to `EmployeeView(persons)`, is also applicable to `TFView(persons)`.

```
(* New view of people who are both Student and Employees *)
-> val supported_student = join(StudentView(persons),EmployeeView(persons));
>> val supported_student =  ....
   :  [Name:string, Salary:int, Advisor:PersonObj, Id:PersonObj]

(* Names of students who earn more than their advisors *)
-> select x.Name
   where x <- supported_student, y<-EmployeeView(persons)
   with x.Advisor=y.Id andalso x.Salary > y.Salary;
>> val it =  ...    :  string
```

Figure 9: Using join to find an intersection

Figure 9 shows how `join` can be used to construct a new view and gives a query on that view.

Dual to the join which corresponds to the intersection of classes, the union of classes can be also represented in Machiavelli. The primitive operation `unionc` is a generalization of the union defined in connection with `hom` to the operate on type $\{\delta_1\} * \{\delta_2\}$ for all description types δ_1, δ_2 such that $\delta_1 \sqcap \delta_2$ exists. Let s_1, s_2 be two sets having types $\{\delta_1\}, \{\delta_2\}$ respectively. Then $\texttt{union}(s_1, s_2)$ satisfies the following equation:

$$\texttt{union}(s_1, s_2) = \texttt{project}(s_1, \delta_1 \sqcap \delta_2) \cup \texttt{project}(s_2, \delta_1 \sqcap \delta_2)$$

which is reduced to the standard set-theoretic union when $\delta_1 = \delta_2$. This operation can be used to give a union of classes of different type. For example, `union(StudentView(person), EmployeeView(person))` correspond to the union of students and employees. On such a set, one can only safely apply methods that are defined both on students and employees. As with join, this constraint is automatically maintained by Machiavelli's type system because the result type is {`Person`}.

In addition one can easily define the "membership" operation on classes of disparate type.:

```
fun member(x,S) = join({x},S) <> {}
```

`member(x,S) = true` iff there is some member of s of `S` such that `x` and s have a common identity. In this fashion it is possible to extend a large catalog of set-theoretic operations to classes.

It is interesting to note that this approach, when considered as a data model, has some similarities with that proposed in the IFO model [AH87]. The database consists of a collection of sets of different types of which a set of type *PersonObj* in our example, would be one. Subclasses ("specializations" in IFO) correspond to views. However, unions of these cannot be formed directly, because the `Id` fields will have different types. The correct way to form a union (IFO's "generalizations") would be to exploit a variant type.

Of course, a good database programming language should not only be able to manipulate databases that conform to its own type system but others as well. In particular, most current object-oriented database languages do not have any static type-checking, but we would still like to deal with them in the same way that we have dealt with uniformly typed classes. This is possible through use of *dynamic* values. A dynamic value [Car84b] is one which carries its type description with it. Functions exist for interrogating this type description and for coercing dynamic values back to ordinary typed values. Let us assume that dynamic values also behave like references in that two dynamic values are equal only if they were created by the same invocation of the function `Dynamic`, which creates dynamic types.

We can now view an external database as a single large set of dynamic values, i.e. it has type {`dynamic`}. In the same fashion that we generated views above, we can generate views (probably by some external procedures) based on dynamic. Thus an employee view of the database might be a class of type

```
{[Name: string, Salary: int, Id: dynamic]}
```

and a department view could be a class of type

```
{[Dname: string, Building: string,
  Id: dynamic]}
```

with the "intersection" of these classes being empty. Once this has been done we can write programs to manipulate these structures in the type-safe way we have advocated throughout this paper even though the underlying database does not have any imposed type constraints. The implementation of views (in addition we would need procedures to perform updates) must, of course, respect the projection property we described earlier. But we believe that, for a given object-oriented database system, building these views will be straightforward and could be carried out by generating them automatically.

6 Conclusions and Directions for Further Investigations

We have shown that a variety of database structures can be mapped into the type system of Machiavelli. In particular relational databases (including higher order relations) can be directly represented as Machiavelli values. When we come to object-oriented databases, we still achieve a representation but we need to define views in order to gain the advantages of Machiavelli's polymorphism. In a sense, the definition of views corresponds to a "data definition language" for Machiavelli, and it would be preferable if this language could naturally merge with the language of types. Fully specifying object-oriented database models might require definitional facilities for inheritance relationships between types. Interestingly, similar requirements seem to arise when trying to integrate encapsulation (data abstraction) in a manner that accords with the object-oriented principle of code re-use.

Integrating type definition in a language based on complete type inference, and doing it in a conceptually uniform and elegant way constitutes a challenge. Some experiments are under way with Machiavelli in this direction. We hope that these experiments will help in making the right language design decisions.

It would be interesting to know if Cardelli and Wegner's view of inheritance can be reconciled with the inheritance-by-record-type-inference view of Wand [Wan87], which is the view supported by Machiavelli. One of us has recently been involved in some research [BCGS] that may shed some light on this problem by providing a new semantic interpretation of the Cardelli-Wegner type system. This semantics shows that interpreting subtyping as an *already lambda definable* coercion map is consistent with polymorphism, bounded quantification, and even with recursive types. This de-mysticizes the subtyping (inheritance) relation, and thus makes us feel more comfortable tampering with the type system (as long as the changes still fit the interpretation; but the interpretation turns out to be pleasantly flexible).

For example, one can have unique types for description type values and still have the some degree of "method inheritance" if one does away with the rule

$$\frac{e : \sigma \qquad \sigma \leq \tau}{e : \tau}$$

and replaces the rule

$$\frac{e : \sigma \to \tau \qquad e' : \sigma}{e(e') : \tau}$$

with the rule

$$\frac{e : \sigma \to \tau \qquad e' : \rho \qquad \rho \leq \sigma}{e(e') : \tau}$$

From a database perspective, there are a number of important ways in which Machiavelli needs to be augmented to make it a viable database programming language. The most important of these is the implementation of persistence and efficient evaluation of set expressions. On the other hand we feel that we do not need to deal in great detail with the efficiency of the whole range of database structures. Our hope is that Machiavelli can be parasitic on already implemented database management systems and will serve as a medium for communication between heterogeneous systems and, in particular, that it will allow us to achieve a clean integration of already implemented relational and object-oriented systems. Of course, this does not mean that we can automatically map an object-oriented database into a relational database, since most implemented relational databases are constrained to be in first normal form. But we hope that a programming language such as Machiavelli will make it relatively easy to transfer data between heterogeneous databases.

Acknowledgement

We would like to thank Malcolm Atkinson for his careful reading of this paper and many helpful comments.

References

[AB87] M.P. Atkinson and O.P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, June 1987.

[ABC*83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An Approach to Persistent Programming. *Computer Journal*, 26(4), November 1983.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A Strongly Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[AH87] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[BBKV88] F. Bancilhon, T. Briggs, S. Koshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 97–105, 1988.

[BCGS] V. Breazu-Tannen, T. Coquand, C.A. Gunter, and A. Scedrov. Inheritance and Explicit Coercion (Preliminary Report). Unpublished Manuscript, University of Pennsylvania.

[BJO89] P. Buneman, A. Jung, and A. Ohori. Using Powerdomains to Generalize Relational Databases. *Theoreical Computer Science*, To Appear, 1989. Available as a technical report from Department of Computer and Information Science, University of Pennsylvania.

[Car84a] L. Cardelli. A Semantics of Multiple Inheritance. In *Semantics of Data Types, Lecture Notes in Computer Science 173*, Springer-Verlag, 1984.

[Car84b] L. Cardelli. *Amber*. Technical Memorandum TM 11271-840924-10, AT&T Bell Laboratories, 1984.

[CDJS86] M. Carey, D. DeWitt, Richardson J., and E Sheikta. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of the 12th VLDB Conference, Kyoto, Japan*, August 1986.

[CM84] G. Copeland and D. Maier. Making Smalltalk a Database System. In *Proceedings of ACM SIGMOD*, pages 316–325, ACM, June 1984.

[Cou83] B. Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surverys*, 17(4):471–522, December 1985.

[Eli82] T.S. Eliot. *Old Possum's Book of Practical Cats.* Harcourt Brace Jovanovic, 1982.

[HMT88] R. Harper, R. Milner, and M. Tofte. *The Definition of Standard ML (Version 2).* LFCS Report Series ECS-LFCS-88-62, Department of Computer Science, University of Edinburgh, August 1988.

[JM88] L. A. Jategaonkar and J.C. Mitchell. ML with extended pattern matching and subtypes. In *Proc. ACM Conference on LISP and Functional Programming*, pages 198–211, Snowbird, Utah, July 1988.

[Mai89] D Maier. Why Database Langauges Are a Bad Idea. In F. Bancilhon and P. Buneman, editors, *Workshop on Database Programming Languages*, Addison-Wesley, 1989. To Appear.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[OB88] A. Ohori and P. Buneman. Type Inference in a Database Programming Language. In *Proc. ACM Conference on LISP and Functional Programming*, pages 174–183, Snowbird, Utah, July 1988.

[OBS86] P O'Brien, B Bullis, and C. Schaffert. Persistent and Shared Objects in Trellis/Owl. In *Proc. of 1986 IEEE International Workshop on Object-Oriented Database Systems.*, 1986.

[Oho88] A. Ohori. Semantics of Types for Database Objects. In *Proc. International Conference on Database Theory, Lecture Notes in Computer Science 326*, pages 239–251, Bruges, Belgium, August 1988. Extended version submitted to a special issue of *Theoretical Computer Science.*

[Oho89] A. Ohori. A Simple Semantics for ML Polymorphism. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture*, September 1989. To appear.

[Sch77] J.W. Schmidt. Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems*, 5(2), 1977.

[Sta88] R. Stansifer. Type Inference with Subtypes. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 88–97, 1988.

[Tur85] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 201*, pages 1–16, Springer-Verlag, 1985.

[Wan87] M. Wand. Complete Type Inference for Simple Objects. In *Proceedings of the Second Anual Symposium on Logic in Computer Science*, pages 37–44, Ithaca, New York, June 1987.