September 1990

Polymorphism and Type Inference in Database Programming

Peter Buneman
University of Pennsylvania

Atsushi Ohori
University of Glasgow

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-90-64.

Polymorphism and Type Inference in Database Programming

Abstract

The polymorphic type system of ML can be extended in two ways that make it appropriate as the basis of a database programming language. The first is an extension to the language of types that captures the polymorphic nature of field selection; the second is a technique that generalizes relational operators to arbitrary data structures. The combination provides a statically typed language in which relational databases may be cleanly represented as typed structures. As in ML types are inferred, which relieves the programmer of making the rather complicated type assertions that may be required to express the most general type of a program that involves field selection and generalized relational operators.

It is also possible to use these ideas to implement various aspects of object-oriented databases. 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. Moreover it is possible to exploit the type system to check the consistency of object-oriented classes (abstract data types with inheritance). A prototype language based on these ideas has been implemented. While it lacks some important practical features, it demonstrates that a wide variety of database structures can be cleanly represented in a polymorphic programming language.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-90-64.

Polymorphism And Type Inference In Database Programming

MS-CIS-90-64
LOGIC & COMPUTATION 23

Peter Buneman
University of Pennsylvania

Atsushi Ohori
University of Glasgow

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

September 1990

Polymorphism and Type Inference in Database Programming

Peter Buneman*
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104, U.S.A.

Atsushi Ohori†
Department of Computing Science
University of Glasgow
Glasgow G12 8QQ, Scotland

Abstract

The polymorphic type system of ML can be extended in two ways that make it appropriate as the basis of a database programming language. The first is an extension to the language of types that captures the polymorphic nature of field selection; the second is a technique that generalizes relational operators to arbitrary data structures. The combination provides a statically typed language in which relational databases may be cleanly represented as typed structures. As in ML types are inferred, which relieves the programmer of making the rather complicated type assertions that may be required to express the most general type of a program that involves field selection and generalized relational operators.

It is also possible to use these ideas to implement various aspects of object-oriented databases. 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. Moreover it is possible to exploit the type system to check the consistency of object-oriented classes (abstract data types with inheritance). A prototype language based on these ideas has been implemented. While it lacks some important practical features, it demonstrates that a wide variety of database structures can be cleanly represented in a polymorphic programming language.

1 Introduction

Expressions such as `3 + "cat"` and `[Name = "J. Doe"].PartNumber` contain *type errors*: the application of some primitive operation such as "+" or "." (field selection) to inappropriate values. The detection of type errors in a program before it is executed is, we believe, of great importance in database programming, which is characterized by the complexity and size of the data structures involved. For relational query languages checking of the type correctness of a query such as

```
select Name
from Employee
where Salary > 100000
```

is a straightforward process that is routinely carried out by the compiler. However, once we add some form of procedural abstraction to the language, the problem is no longer trivial. For example, how do we check the type correctness of a program containing definitions such as

*Supported by research grants NSF IRI86-10617, ARO DAA6-29-84-k-0061 and ONR NOOO-14-88-K-0634
†Supported by a British Royal Society Research Fellowship. On leave from OKI Electric Industry, Co., Japan.

```
function Wealthy(S) = select Name
                      from S
                      where Salary > 100000        ?
```

This function is polymorphic in the sense that it should be applicable to *any* relation S that contains **Name** and **Salary** fields of the appropriate type. In database programming languages there have been two general strategies. One is to follow the approach of Pascal-R [Sch77] and Galileo [ACO85] and insist that the parameters of procedures are given specific types, e.g. **function Wealthy(S:EmployeeRel)** $\cdots$. 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 not possible to express the kind of polymorphism inherent in a function such as **Wealthy**. The other approach is used in 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 polymorphic code, some dynamic type-checking is required. Napier [MBCD89] attempts to combine parametric polymorphism [Rey74] and persistence but its polymorphism does not extend to labeled records and other database structures. The current practice in database programming is to use a query language embedded in a host language. In this arrangement, communication between programs in different languages is so low-level that type-checking is effectively non-existent, so that programs that violate the intended types result in "junk". See [AB87] for a survey of the various approaches to type-checking.

The language ML [HMT88] has a *type inference system* which infers, if possible, a most general polymorphic type for a program. Because of this, ML enjoys much of the flexibility of untyped (or dynamically typed) languages without sacrificing the advantage of static type checking. Unfortunately the type inference of ML [Mil78, DM82] is not general enough to be applied to structures and operations needed for database programming. For example, it cannot infer a polymorphic type for a function containing field selection such as **Wealthy** above. Our goal in this paper is to show that a polymorphic programming language can uniformly incorporate databases. In particular, we will show that a polymorphic type system, when properly extended to suitable data types and database operations, will serve as a medium to represent both relational databases and more recent object-oriented databases directly within a polymorphic programming language. Database programming in such a language can make full use of a rich, statically checked polymorphic type system. 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, cyclic data and some form of persistence. Our hope is that Machiavelli (or some language like it) will provide a framework for dealing uniformly with both relational and object-oriented databases.

To illustrate a program in Machiavelli, consider the function **Wealthy**. Note that this function takes a set of records (i.e. a relation) with **Name** and **Salary** information and returns the set of all **Name** values which occur in records which contain **Salary** values over 100K. For example, applied to the relation

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

which is Machiavelli syntax for a set of records, this function should yield the set {**"Fred"**, **"Helen"**} of character strings. This function is written in Machiavelli (whose syntax mostly follows that of ML) as follows

```
fun Wealthy(S) = select x.Name
```

```
from x <- S
where x.Salary > 100000;
```

The **select** ... **from** ... **where** ... 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}
```

This *type scheme* may be instantiated with appropriate substitutions for the type variables "a and "b. For example

```
{[Name:string, Salary:int]} -> {string}
{[Name:string, Age:int, Salary:int]} -> {string}
{[Name:[First:string, Last:string], Weight:int, Salary:int]}
     -> {[First:string, Last:string]}
```

are legal instantiations of the type of **Wealthy**. On the other hand the following expressions and definitions involving **Wealthy**

```
Wealthy({[Name = "Joe"], [Name = "Fred"]})
Wealthy({[Name = "Joe", Salary = "nonsense"]})
fun F(S) = sum(Wealthy(union(S,{[Name = "Joe", Salary = 200000]})))
```

will be rejected by the compiler.

In order to extend ML's type inference to handle these examples, we need to consider two restrictions that control the way substitutions may be applied to type variables. The first of these is that "b, for example, should be a *description type* – one for which equality is available. The need for this is seen from the fact that the type expression {"b} indicates that values of type "b are members of some set, and therefore we must have an equality test for such values. Description types are a generalization of ML's *eqtypes* and also have available a number of useful relational database operations such as join and projection. We need to treat description types specially because equality is not available on certain values such as functions or, perhaps, certain base types. Description types are those that can be constructed from the allowed base types through any type construction other than a function type outside the scope of a reference type.

The second form of restriction is expressed by [("a) Name:"b, Salary:int], which is also a type variable. In addition to being a description type it is *kinded* with the restriction that any instance must contain the fields **Name**:σ and **Salary**:**int** where σ is any instance of "b. Kinded type variables capture the two forms of polymorphism in the definition of **Wealthy**: it is polymorphic with respect to the type of the **Name** field, and it is also polymorphic with respect to the record type containing the **Name** and **Salary** fields. It is because we can extend ML's type inference to deal with description and kinded type variables that Machiavelli has the power to deal with a wide range of database constructs.

In the polymorphism represented by the kinded type variables, there is a close relationship with object-oriented programming. The type scheme {[("a) Name:"b, Salary:int]} can be thought of as a class and functions that are polymorphic with respect to this, such as **Wealthy**, can be thought of as methods of that class. 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 [Car88, CW85]. However, there are important technical differences, the most important of which is that in Machiavelli database values have

unique types, while they have multiple types in Cardelli and Wegner's type systems. Database types in Machiavelli specify the exact structure of values and this property is needed in order to implement various database operations such as equality and *natural join*. (See [BTBO89] for more discussion.) In Machiavelli inheritance is achieved not by *subtyping* but by polymorphic instantiation of kinded type variables. The most important practical difference is that this polymorphism is *inferred*, which means that the programmer does not have to declare and explicitly instantiate the rather complicated forms needed in the Cardelli and Wegner system to capture precisely the polymorphic nature of functions such as `Wealthy`.

Another important extension to these type systems for objects and inheritance is that Machiavelli uniformly integrates *set types* and various database operations including generalized *join* and *projection* in its polymorphic type system. Sets may be constructed on any description type; combined with labeled records, labeled variants and cyclic definitions, the Machiavelli type system allows us to represent most of the structures found in various complex data models [HK87]. Cyclic structures are supported by exploiting the properties of *regular trees* [Cou83]. *Join* and *projection* are generalized to arbitrary, possibly cyclic, structures and are polymorphic functions in Machiavelli's type system. This immediately provides a natural representation of a generalized relational (or complex object) data model within a polymorphic type system of a programming language and achieves a natural integration of databases and a programming language. Although there is some argument about the nature of object identity, we shall assume that it is adequately captured through reference types; then, by representing "objects" as references to complex values, we obtain representations similar to those used in object-oriented databases in a static type system. In particular, by the construction of *views*, functions that preserve identity, we are able to combine naturally two notions of inheritance: subclasses in programming languages and subsets in databases. In fact, it is the availability of generalized relational operations that allows us to do this. Also, as we shall see in section 6 the form of inheritance expressed by Machiavelli's polymorphism can be integrated with *data abstraction*, thereby achieving a basic property of object-oriented programming in a statically typed framework.

The organization of this paper is as follows. Section 2 introduces the basic data structures of Machiavelli including records, variants and sets, and shows how relational queries can be obtained with the operations for these structures. Section 3 contains a definition of the core language itself. It defines the syntax of types and terms, and describes the type inference system. In section 4, the language is extended with relational operations – specifically join and projection – that cannot be derived from basic set operations, and the type inference system is extended to handle them. In section 5 we show how this type system can be used to represent some important aspects of object oriented databases. Section 6 extends the core type system to represent data abstraction and *multiple inheritance*. Again, we should emphasize that Machiavelli is far from a complete database programming language, and while we believe that its type system can be used in a full-fledged language, some care must be taken to ensure that the type system can be used in conjunction with other useful features. Section 7 discusses these problems and the further work that is needed to make the language useful in dealing with external databases.

2 Basic Structures for Data Representation

As we have just mentioned, one of the goal of this study is to develop a polymorphic type system that serves as a medium to represent various database structures. In particular it should be expressive enough to represent various forms of complex objects that violate the "first-normal-form assumption" that underlies most implemented relational database systems and most of the traditional theory of relational databases.

For example we want to be able to deal with structures such as

```
{[Name = [First = "Bridget", Last = "Ludford"],
  Children = {"Jeremy", "Christopher"}],
 [Name = [First = "Ellen", Last = "Gurman"],
  Children = {"Adam", "Benjamin"}]}
```

which is built up out of records and (uniformly typed) sets. This structure is a "non-first-normal-form" relation in which the **Name** field contains a record and the **Children** field contains a set of strings. It is an example of a *description term*, and in this section we shall describe the constructors that enable us to build up such terms from atomic data: records, variants, sets and references. We shall also describe how cyclic structures are created.

Some of the basic syntactic forms of Machiavelli for value and function definition have been borrowed from ML [HMT88]. Knowledge of ML syntax should not be needed provided a few basic forms are understood. In particular, names are bound to values by the use of **val**, as in

```
val four = 2 + 2;
```

functions are defined through the use of **fun**, as in

```
fun f(n) = if eq(n,0) then 0 else n + f(n-1);
```

and there is an function constructor **fn x =>** ... that is used to create functions without naming them, as in

```
(fn x => x + x) (4)
```

which evaluates to 8. In fact, since a fixed point operator is lambda-definable in Machiavelli (using recursive types), recursive function definition can be obtained from value definition and is not essential. It is here for convenience. Finally there is the form **let** $x = e_1$ **in** e_2 **end**, which evaluates e_2 in the environment in which x is bound to e_1. Example:

```
let x = 4 + 5 in x + x*x end
```

which evaluates to 90. In an untyped language, **let** ... **in** ... **end** is also not essential, but the type inference rules are such that this form is treated specially, and is the basis for ML's polymorphism. By implicit or explicit use of **let**, polymorphic functions are bound and used. Polymorphic function definitions such as that of **Wealthy** above are treated as shorthand for a **let** binding whose scope is the rest of the program.

2.1 Labeled Records and Labeled Variants

The syntax for labeled records is:

$$[l_1 = v_1, \ldots, l_n = v_n]$$

where $l_1, \ldots, l_n$ stand for *labels*. A record is a description term if all its fields $v_1, \ldots, v_n$ are description terms. Other than record construction, ([...]), there are two primitives for records. The first, $r.l$, is field selection found in many programming languages, which selects the l field form the record r. The second, **modify**(r,l,e), is field modification, which creates a new record identical to r except on the l field where its value is e. For example,

```
modify([Name = "J. Doe", Age = 21], Age, 22)
```

yields (evaluates to) `[Name = "J. Doe", Age = 22]`. It is important to note that **modify** does *not* have a side-effect. It is a function that returns another record. The syntax (e_1, e_2) for pairs is simply an abbreviation for the record `[first =` e_1`, second =` e_2`]`. Triples and, generally, n-tuples are similarly constructed.

The syntax of labeled variants (injection to labeled disjoint union) is:

<l=v>

A variant is a description term if its component v is a description term. The operation for decomposing a variant is case statement:

case e **of**
  <l_1=x_1> => e_1,
  ⋮
  <l_n=x_n> => e_n,
  else e_0
endcase

where x_i in <l_i=x_i> => e_i is a variable whose scope is in e_i. This operation first evaluates e and if it yields a variant <l_i=v> then binds the variable x_i to the value v and evaluates e_i under this binding. If there is no matching case then the **else** clause is selected. The **else** is optional, and if it is omitted the argument e must be evaluated to a variant labeled with one of $l_1, \ldots, l_n$. This condition is ensured by the type system. Note that **case**...**of**...**endcase** is an expression, and returns a value. The possible results $e_1, \ldots, e_n, e_0$ should all have the same type.

For example,

```
case <Consultant = [Name = "J. Doe", Address = "10 Main St.",
                    Phone = "222-1234"]>
of
   <Consultant = x> => x.Phone,
   <Employee = y> => y.Extension
endcase
```

yields `"222-1234"`.

2.2 Sets

Sets in Machiavelli can only contain description terms and sets themselves are always description terms. This restriction is essential to generalize database operations over structures containing sets. There are four basic operations for sets:

$\{\}$	*empty set,*
$\{x_1, x_2, \ldots, x_n\}$	*set constructor,*
union(s_1, s_2)	*set union,*
hom(f, op, z, s)	*homomorphic extension*

Of these operations, **hom** requires some explanation. This 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

$$\mathbf{hom}(f,op,z,\{\}) = z,$$
$$\mathbf{hom}(f,op,z,\{e_1,\ e_2,\ldots,\ e_n\}) = op(f(e_1),op(f(e_2),\ldots,op(f(e_n),z)\cdots)).$$

for example, a function to add up the members of a set may be defined as

```
fun sum S = hom(fn x => x, +, 0, S)
```

and a function that finds the size of a set is

```
fun card S = hom(fn x => 1, +, 0, S)
```

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 (as is the case in the **sum** and **card** examples) 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. Proper applications of **hom** give rise to deterministic computations and have the property of being computable in parallel. Equality on values that result from improper applications may not be what was intended by the programmer. It is an interesting question to ask when an application of **hom** can be shown, by static analysis of a program, to be proper.

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

$$\mathbf{hom}^*(f,op,\{e_1,\ x_2,\ldots,\ e_n\}) = op(f(e_1),op(f(e_2),\ldots,\ op(f(e_{n-1}),f(e_n))\cdots)).$$

When z is an identity for *op*, **hom** behaves as **hom*** on non-empty sets. **hom*** is useful when the value z for the empty set is difficult to find as in the example:

```
fun max(S) = hom*(fn x=>x, fn (x,y) => if x > y then x else y,S)
```

which computes the maximal element of a non empty set of integers.

The following useful functions can be defined using **hom**:

```
fun map(f,S) = hom(f, union, {}, S)
```

`map`(f,S) applies the function f to each member of S; for example `map(card,{{1,2},{3},{6,5,4}}` evaluates to `{2,1,3}`.

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

`filter`(p,S) extracts those members of S that satisfy property p; for example `filter(odd,{1,2,3,4})` evaluates to `{2,4}`.

In addition to these examples **hom** can be used to define set intersection, membership in a set, set difference, the n-fold cartesian product (denoted by `prod_n` below) of sets and the powerset (the set of subsets) of a set. Also, the form

select E
from x_1 <- S_1,
x_2 <- S_2,
$\vdots$
x_n <- S_n
where P

which is provided in the spirit of relational query languages and the list 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_n(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).

2.3 Cyclic Structures

In many languages, the ability to define cyclic structures depends on the ability to reassign a pointer. In Machiavelli, these two ideas are separated. It is possible to create a structure with cycles through use of the **(rec** $v.e$**)** construct, e.g.

```
val Montana = (rec v.[Name = "Montana", Motto = "Big Sky Country",
                      Capital = [Name = "Billings", State = v]])
```

This record behaves like an infinite tree obtained by arbitrary unfolding by substitution for `v`. For example, the expressions `Montana.Capital`, `Montana.Capital.State`, `Montana.Capital.State.Capital`, etc are all valid. Moreover, equality and other database operations on description terms generalize to those cyclic structures. This uniform treatment is achieved by treating description terms as *regular trees* [Cou83]. The syntax **(rec** $v.e$**)** denotes the regular tree given as the solution to the equation $v = e$, where e may contain the symbol v.

2.4 References

We believe that the notion of "objects" in databases is equivalent to that of references as they are implemented in ML. There are three primitives for references:

new(v**)** *reference creation,*
`!`r *de-referencing,*
r`:=`v *assignment.*

new(v**)** creates a new reference and assigns the value v to it. `!`r returns the value associated with the reference r. r`:=`v changes the value associated with the reference r to v. In a database context, they correspond respectively to the creation of an object with identity, retrieving the value of an object, and changing the associated value of an object without affecting its identity.

When combined with other description term constructors, references represent objects with identity. The uniqueness of identity is guaranteed by the uniqueness of each reference. Two references are equal only if they are the results of the same invocation of **new** primitive. For example if we create the following two *objects* (i.e. references to records):

```
John1 = new([Name="John", Age= 21]);
John2 = new([Name="John", Age= 21]);
```

then `eq(John1,John1)` and `eq(!John1,!John2)` are `true` but `John1 = John2` is `false` even though their associated values are the same. Sharing and mutability are also represented by references. If we define a department object as:

```
SalesDept = new([Name = "Sales", Building = 11]);
```

and from this we define two employee objects as:

```
John = new([Name="John", Age =21, Dept = SalesDept]);
Mary = new([Name="Mary", Age =31, Dept = SalesDept]);
```

then `John` and `Mary` *share* the same object `SalesDept` as the value of `Department` field. An update to the object `SalesDept` as seen from `John`.

```
(!John).Dept := modify(!((!John).Dept), Building, 98)
```

is reflected in the department as seen from `Mary`. After this statement,

```
(!((!Mary).Dept)).Building
```

evaluates to `98`. Unlike many languages references do not have an optional "nil" or "undefined" value. If such an option is required it must be explicitly introduced through the use of a variant.

3 Type Inference and Polymorphism in Machiavelli

Type inference is a method to infer type information that represents the polymorphic nature of a given untyped (or partially typed) program. Hindley established [Hin69] a complete type inference algorithm for untyped lambda terms. Independently, Milner developed [Mil78] a complete type inference algorithm for functional programming language including polymorphic definition (using **let** construct.) This has been successfully used in the ML family of programming languages [Aug84, HMT88] and also been adopted by other functional languages [Tur85, HW89]. Unfortunately this method cannot be used directly with some of the data structures and operations we have described in the previous section. In this section we give an account of the extension to the Hindley-Milner type system that is used in Machiavelli, first through some examples and then through a definition of the "core" language and its type system.

For programs which do not involve field selection, variants and database operations, Machiavelli infers type information similar to those of ML. For example, for the identity function

```
fun id x = x;
```

the type system infers the following type information

```
id :  'a -> 'a
```

where `'a` is a *type variable* intuitively representing an "arbitrary type". This is a *type scheme* which is a representation of the set of all types obtained by substituting its type variables with some types (such as *int*, *bool* or $int \rightarrow int$). This distinction of type schemes from types is crucial to understand Machiavelli's type system. Note that a type is also a type scheme representing the singleton set of itself. The most important property of the ML type system is that for any type consistent expression it infers a *principal type scheme*. This is a type scheme such that all its ground instance are types of the expression and conversely any type of the expression is its instance. This means that the type system infers a type scheme that exactly represents the set of all possible types of an expression. By this mechanism, ML achieves *polymorphism* without explicit type abstraction and type application. The inferred type scheme can be regarded as the polymorphic type of the expression. In the case of `id`, the type scheme `'a -> 'a` represents the set of all possible types of `id` and is therefore regarded as the polymorphic type $\forall t.\, t \rightarrow t$ of `id`.

A more substantial example of type inference is given by the function `map` of the previous section, which has type scheme

```
map : ("a->"b * {"a}) -> {"b}
```

Here `"a` and `"b` are also type variables, but in this case they only represent description types. The type scheme for `map` indicates that it is a function that takes a function of type δ_1 `->` δ_2 and a set of type $\{\delta_1\}$ and returns a set of type $\{\delta_2\}$ where δ_1, δ_2 can be any description types. Thus `map(card, {{1,2,3},{7},{5,2}})` and `map(odd, {9,8,7,6})` are both legitimate applications of `map`. Again, the type scheme `("a->"b * {"a}) -> {"b}` is principal in that any type for `map` is obtained by substituting description types for the type variables `"a` and `"b`. In the example, `({int} -> int * {{int}}) -> {int}` is the type of `map` in `map(card, {{1,2,3},...})`.

Similar examples are possible in ML and its relatives. However it is not possible for ML's type inference method to infer a type scheme for a program involving field selection, variants or the relational database operations that we shall describe later. For example, the simplest function using field selection

```
fun name x = x.Name
```

cannot be typed by ML. (In Standard ML, this function is written `fun name {Name = x,...} = x`, which is rejected by the compiler unless a complete type is specified for the argument.) The difficulty is that the conventional notion of type schemes is not general enough to represent the relationship between the argument type and the result type, which in this case is the inclusion of a field type in a record type.

Wand attempted [Wan87] to solve this problem (with the operation that extends a record with a field) using the notion of *row variables*, which are variables ranging over record fields. The system, however, does not share with ML the property of principal typing (see [OB88, Wan88] for the analysis of the problem and [JM88, Rém89] for the refinement of the system.) Based on Wand's general observation, in [OB88] we developed a type inference method which overcomes the difficulty and extends the method to database operations. Instead of using row variables, we introduced syntactic conditions to control substitution of type variables. For records and variants, the necessary conditions can be refined as *kinded* type variables [Oho90] which have pleasantly simple representation, as we have seen in the example of `Wealthy` in the introduction. For example, the function `name` above is given the following type scheme

```
name : [('a) Name:'b] -> 'b
```

As explained in the introduction, the notation `[('a) Name:'b]` is a kinded type variable representing the set of all record types containing the field `Name:`τ where τ is any instance of `'b`. Substitutions are restricted to

```
-> val joe = [Name="Joe", Age=21,
              Status=<Consultant = [Address="Philadelphia", Telephone=2221234]>];
>> val joe = [Name="Joe", Age=21,
              Status=<Consultant = [Address="Philadelphia", Telephone=2221234]>]
       :  [Name:string, Age:int, Status:<('a)Consultant:[Address:string,Telephone:int]>]
-> fun phone(x) = case x.Status of
                       <Employee = y> => y.Extension,
                       <Consultant = y> => y.Telephone
                  endcase
>> 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

those that respect kind restrictions of type variables. The type scheme above then represents the exact set of all possible types of the function **name** and therefore regarded as a principal (kinded) type scheme for **name**. More examples of type inference for records and variants are shown in figure 1 which shows an interactive session in Machiavelli. Input to the system is prompted by **->**, and output is preceded by **>>**. At the top level input is either a value or function binding; **it** is a name for the result of evaluation of an expression. The output consists of some description of the value that has just been evaluated or bound together with its inferred type.

We now define a small polymorphic functional language by combining the data structures described in the previous section with a functional calculus and giving its type system. This will serve as the polymorphic "core" of Machaivelli.

3.1 Expressions

The syntax of programs or *expressions* of the core language is given by

$$
\begin{aligned}
e ::= {} & c_\tau \mid () \mid x \mid (\mathbf{fn}\ x \Rightarrow e) \mid e(e) \mid \mathbf{let}\ x{=}e\ \mathbf{in}\ e\ \mathbf{end} \mid \\
& \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid \mathbf{eq}(e,e) \mid \\
& [l{=}e,\ldots,l{=}e] \mid e.l \mid \mathbf{modify}(e,l,e) \mid \\
& \langle l{=}e\rangle \mid \mathbf{case}\ e\ \mathbf{of}\ \langle l{=}x\rangle \Rightarrow e,\ldots,\langle l{=}x\rangle \Rightarrow e\ \mathbf{endcase} \mid \\
& \mathbf{case}\ e\ \mathbf{of}\ \langle l{=}x\rangle \Rightarrow e,\ldots,\langle l{=}x\rangle \Rightarrow e\ \mathbf{else} \Rightarrow e\ \mathbf{endcase} \mid \\
& \{e,\ldots,e\} \mid \mathbf{union}(e,e) \mid \mathbf{hom}(e,e,e,e) \mid \mathbf{hom^*}(e,e,e) \mid
\end{aligned}
$$

new(e) | (!e) | e:=e |
(**rec** x.e)

In this, c_τ stands for standard constants including constants of base types and ordinary primitive functions on base types. x stands for the variables of the language. () is the single value of type *unit* and is returned by expressions such as assignment. Examples of the syntax have already been given in section 2 and, in particular, in figure 1. Value binding **val** id = e_1; e_2 is syntactic sugar for **let** id = e_1 **in** e_2. Recursive function definition with multiple argument is also syntactic sugar for expressions constructed from **let**, records, field selection and a fixed point combinator, which is already lambda-definable in Machiavelli using recursive types. Evaluation rules for those expressions are obtained by extending the operational semantics of ML such as the one defined in [Tof88] with the rules for **eq** and the operations on records, sets, and variants and the rules for recursive expressions. The rule for **eq** requires delicate treatment in connection with cyclic structures and sets and we defer it until we discuss database operations in section 4. We have already informally described the evaluation rules for operations on records, sets, and variants. It is not hard to give their formal definitions as reduction rules. In order to handle recursive expressions, we add the following rules. Let $E(x)$ be one of the expressions $e.l$, **modify**(x,l,e), **case** x **of** $\cdots$, **union**(x,e), **union**(x,e), or **hom**(x,e_1,e_2,e_3).

$$E((\textbf{rec } x.e)) \Longrightarrow E(e[(\textbf{rec } x.e)/x])$$

where $e[(\textbf{rec } x.e)/x]$ is the expression obtained form e by substituting x in e for (**rec** x.e). This rule corresponds to "unfolding" of cyclic definitions.

3.2 Types, Description Types and Typing Rules

As explained above, Machiavelli type system is based on type inference. A legal Machiavelli program corresponds to an (untyped) expression associated with a type scheme inferred by the type inference system. As such an implicit type system, the definition of Machiavelli type system requires two steps. The first is to give *typing rules*, which determines when an untyped expression e is considered to have a type τ and therefore considered as a well typed expression. The second step is to develop a type inference algorithm that infers for any type consistent expression a principal type scheme representing the set of all possible types of the expression derivable form the typing rules. In this subsection, we give the complete set of typing rules.

The set of types of Machiavelli is the set of regular trees [Cou83] represented by the following type expressions:

$$\tau \quad ::= \quad unit \mid b \mid b_d \mid \tau \rightarrow \tau \mid [l:\tau,\ldots,l:\tau] \mid \langle l:\tau,\ldots,l:\tau\rangle \mid \{\tau\} \mid ref(\tau) \mid (rec\ v.\ \tau(v))$$

unit is the trivial type whose only value is (). b and b_d range respectively over the base types and base description types of the constants in the language. The other type expressions are: $\tau \rightarrow \tau$ for function types, $[l:\tau,\ldots,l:\tau]$ for record types, $\langle l:\tau,\ldots,l:\tau\rangle$ for variant types, and $\{\tau\}$ for set types. In $(rec\ v.\ \tau(v))$, $\tau(v)$ is a type expression possibly involving the symbol v but not v itself and the entire expression denotes the solution to the equation $v = \tau(v)$, which exists in the set of regular trees. In keeping with our syntax for records we shall use the notation $\tau_1 * \tau_2$ as an abbreviation for the type $[first:\tau_1, second:\tau_2]$. Triples and, generally, n-tuple types are similarly treated. Database examples of Machiavelli types are: a relation type,

```
{[PartNum:int, PartName:string, Color:  <Red:unit, Green:unit, Blue:unit>]}
```

a complex object type,

```
{[Name:[First:string, Last:string], Children:{string}]}
```

and a mutable object type,

```
(rec p. ref([Id#:int, Name:string, Children:{p}]))
```

Note that $(rec\ v.\ \tau(v))$ is not a type constructor but a syntax to denote the solution to the equation $v = \tau(v)$. As a consequence, distinct type expressions may denote the same type. For example, the following type expression denotes the same type as the one above:

```
(rec p. ref([Id#:int, Name:string,
             Children:{ref([id#:int, Name:string, Children:{p}]}]))
```

There is an efficient algorithm [Cou83] to test whether two type expressions denote the same type (i.e. regular tree) or not. We can therefore identify type expressions as the types they denote. Note also that an "infinite" (cyclic) type does not necessarily mean that its values are cyclic. In the last example, while the type is cyclic, a cyclic value of this type presents some biological difficulties.

The set of *description types* is the subset of types represented by the following syntax:

$$\delta \quad ::= \quad unit \mid b_d \mid [l:\delta,\ldots,l:\delta] \mid \langle l:\delta,\ldots,l:\delta\rangle \mid \{\delta\} \mid ref(\tau) \mid (rec\ v.\ \delta(v))$$

where τ ranges over the syntax of all type given previously. This syntax forbids the use of a function type or a base type which is not a description type in a description type unless within a `ref(...)`. Thus `int -> int` is not a description type but

```
ref([x_coord:int, y_coord:int, move_horizontal:int -> ()])
```

is a description type. Note the similarity – and differences – between this type and a class definition in object-oriented languages.

The typing rules are given as a set of rules to derive *typing judgements*. Since the type of an expression depends on the type of its free variables, a typing judgement has the form:

$$\mathcal{A} \rhd e : \tau$$

where $\mathcal{A}$ is a function, called a *type assignment*, from a finite subset of variables to types. We write $\mathcal{A}\{x := \tau\}$ for the function $\mathcal{A}'$ such that $dom(\mathcal{A}) = dom(\mathcal{A}) \cup \{x\}$, $\mathcal{A}'(x) = \tau$ and $\mathcal{A}'(y) = \mathcal{A}(y)$ for $y \neq x$. The typing rules for all the operations we have so far given is shown in figure 2.

Our treatment of polymorphic **let** (the rule LET) differs form Damas-Milner system [DM82] in that it does not use generic types (a type expression of the form $\forall t.\ \tau$) but instead it use syntactic substitution of terms. A naive implementation of this form of typing rule would require recursive unfolding of **let** definitions. This unfolding process always terminates but would prohibit the possibility of incremental type-checking. For the closed raw terms, however, our proof system is equivalent (when restricted to the raw terms ML) to Damas and Milner's system and their technique for inferring type scheme for **let** expressions (their algorithms $\mathcal{W}$ and $\mathcal{J}$) is also applicable to our system. The advantage of our treatment of let is that the type system can be extended to records, variants and database operations. While it is shown that [Oho90] it is still possible to extend Damas-Milner generic type schemes to records and variants using kinded type abstraction, we do not know how to extend them to the conditional typing schemes that we shall require for database operations.

(CONST) $\mathcal{A} \triangleright c_\tau : \tau$

(VAR) $\mathcal{A} \triangleright x : \tau$ if $\mathcal{A}(x) = \tau$

(APP) $$\frac{\mathcal{A} \triangleright e_1 : \tau_1 \to \tau_2 \qquad \mathcal{A} \triangleright e_2 : \tau_1}{\mathcal{A} \triangleright e_1(e_2) : \tau_2}$$

(ABS) $$\frac{\mathcal{A}\{x := \tau_1\} \triangleright e : \tau_2}{\mathcal{A} \triangleright \textbf{fn } x \textbf{ => } e : \tau_1 \to \tau_2}$$

(LET) $$\frac{\mathcal{A} \triangleright e_1[e_2/x] : \tau \qquad \mathcal{A} \triangleright e_2 : \tau'}{\mathcal{A} \triangleright \textbf{let } x \textbf{ = } e_2 \textbf{ in } e_1 : \tau}$$

(IF) $$\frac{\mathcal{A} \triangleright e_1 : \mathit{bool} \qquad \mathcal{A} \triangleright e_2 : \tau \qquad \mathcal{A} \triangleright e_2 : \tau}{\mathcal{A} \triangleright \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}$$

(RECORD) $$\frac{\mathcal{A} \triangleright e_1 : \tau_1, \ \ldots, \ \mathcal{A} \triangleright e_n : \tau_n}{\mathcal{A} \triangleright [l_1{=}e_1, \ldots, l_n{=}e_n] : [l_1 : \tau_1, \ldots, l_n : \tau_n]}$$

(DOT) $$\frac{\mathcal{A} \triangleright e : \tau_1}{\mathcal{A} \triangleright e.l : \tau_2}$$ if τ_1 is a record type containing $l : \tau_2$

(MODIFY) $$\frac{\mathcal{A} \triangleright e_1 : \tau_1 \qquad \mathcal{A} \triangleright e_2 : \tau_2}{\mathcal{A} \triangleright \textbf{modify}(e_1, l, e_2) : \tau_1}$$ if τ_1 is a record type containing $l : \tau_2$

(VARIANT) $$\frac{\mathcal{A} \triangleright e : \tau_1}{\mathcal{A} \triangleright \langle l{=}e \rangle : \tau_2}$$ if τ_2 is a variant type containing $l : \tau_1$

(CASE) $$\frac{\mathcal{A} \triangleright e : \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle \qquad \mathcal{A}\{x_i := \tau_i\} \triangleright e_i : \tau \ (1 \le i \le n)}{\mathcal{A} \triangleright \textbf{case } e \textbf{ of } \langle l_1{=}x_1 \rangle \textbf{ => } e_1, \ldots, \langle l_n{=}x_n \rangle \textbf{ => } e_n \textbf{ endcase} : \tau}$$

(CASE') $$\frac{\mathcal{A} \triangleright e : \langle l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots \rangle \qquad \mathcal{A}\{x_i := \tau_i\} \triangleright e_i : \tau \ (1 \le i \le n) \qquad \mathcal{A} \triangleright e_0 : \tau}{\mathcal{A} \triangleright \textbf{case } e \textbf{ of } \langle l_1{=}x_1 \rangle \textbf{ => } e_1, \ldots, \langle l_n{=}x_n \rangle \textbf{ => } e_n \textbf{ else => } e_0 \textbf{ endcase} : \tau}$$

(SET) $$\frac{\mathcal{A} \triangleright e_1 : \delta \cdots \mathcal{A} \triangleright e_n : \delta}{\mathcal{A} \triangleright \{e_1, \ldots, e_n\} : \{\delta\}}$$

(UNION) $$\frac{\mathcal{A} \triangleright e_1 : \{\delta\} \qquad \mathcal{A} \triangleright e_2 : \{\delta\}}{\mathcal{A} \triangleright \textbf{union}(e_1, e_2) : \{\delta\}}$$

(HOM) $$\frac{\mathcal{A} \triangleright e_1 : \delta \to \tau_1 \qquad \mathcal{A} \triangleright e_2 : (\tau_1 \times \tau_2) \to \tau_2 \qquad \mathcal{A} \triangleright e_3 : \tau_2 \qquad \mathcal{A} \triangleright e_4 : \{\delta\}}{\mathcal{A} \triangleright \textbf{hom}(e_1, e_2, e_3, e_4) : \tau_2}$$

(EQ) $$\frac{\mathcal{A} \triangleright e_1 : \delta \qquad \mathcal{A} \triangleright e_2 : \delta}{\mathcal{A} \triangleright \textbf{eq}(e_1, e_2) : \mathit{bool}}$$

(NEW) $$\frac{\mathcal{A} \triangleright e : \tau}{\mathcal{A} \triangleright \textbf{new}(e) : \mathit{ref}(\tau)}$$

(DEREF) $$\frac{\mathcal{A} \triangleright e : \mathit{ref}(\tau)}{\mathcal{A} \triangleright !e : \tau}$$

(ASSIGN) $$\frac{\mathcal{A} \triangleright e_1 : \mathit{ref}(\tau) \qquad \mathcal{A} \triangleright e_2 : \tau}{\mathcal{A} \triangleright e_1{:=}e_2 : \mathit{unit}}$$

(REC) $$\frac{\mathcal{A}\{v := \delta\} \triangleright e(v) : \delta}{\mathcal{A} \triangleright (\textbf{rec } v.\ e(v)) : \delta}$$

Figure 2: The Proof System for Machiavelli Typings

3.3 Type Inference

The proof system of figure 2 determines which expressions are type correct legal Machiavelli programs. Unlike the simple type discipline, this proof system does not immediately yield a decision procedure for type checking expressions. The second step of the definition of the type system is to give such a decision procedure. Since an expression may have more than one typing, we need to develop a representation for sets of typings and an algorithm which, given any typable expression, infers a representation for the set of all derivable typings for the expression. This is the type inference problem.

In [Hin69, Mil78] this problem was solved by defining a language of type schemes containing type variables and developing an algorithm which, given a typable expression e, computes a *principal typing scheme* $\Sigma \vdash e : \sigma$ satisfying the property that $\mathcal{A} \rhd e : \tau$ is derivable if and only if there is some substitution θ such that $\mathcal{A}(x) = \theta(\Sigma(x))$ for all $x \in dom(\Sigma)$ and $\tau = \theta(\sigma)$. A legal ML program is one with a principal typing scheme with an empty type assignment Σ.

There are two problems in applying this method to our type system. The first one is that the operational semantics for references does not agree with polymorphic type discipline for **let** binding. As pointed out in [Mac88b, Tof88], the straightforward application of the type inference method of [Mil78] to references yield unsound type system. The following example is given in [Mac88b]:

```
let
    val f = new(fn x => x)
in (f:=(fn x=> x + x), (!f)(true))
end
```

If the type system treats the primitive **new** as an ordinary expression constructor then it would infer the type *bool* for the above expression but the expression causes a run time type error if the evaluation of a pair (record) is left-to-right. In [Tof88, Mac88b], solutions have been proposed. They differ in detailed technical treatment but are both based on the idea that the type system prohibits reference values from having a polymorphic type. In what follows, we may assume either of these proposals.

The other problem we need to address is that, in figure 2, some of the rules have associated conditions:

1. a type should be a description type,
2. τ_1 is a record type containing $l : \tau_2$,
3. τ_1 is a variant type containing $l : \tau_2$,
4. a type should be of the form $\langle l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots \rangle$.

The second, third and fourth of these conditions are explicitly required in the the rules (DOT), (MODIFY), (VARIANT) and (CASE'). The first requirement, that a type should be a description type, is indicated by the use of a δ (rather than τ) in the rules (SET), (UNION), (HOM), (EQ) and (REC).

The first condition is handled by introducing a new class of type variables *description type variables*, similar to ML's *eqtype* variables. In order to represent the other three conditions, we refine type schemes as *kinded type schemes* by introducing *kind constraint* on type variables. The set of kinded type schemes appropriate for Machiavelli is given by the following syntax:

$$\sigma ::= t^K \mid d^K \mid unit \mid \sigma \to \sigma \mid [l : \sigma, \ldots, l : \sigma] \mid \langle l : \sigma, \ldots, l : \sigma \rangle \mid \{\sigma\} \mid ref(\sigma) \mid (rec\ v.\ \sigma(v))$$

$\sigma :: U$ for all σ

$t^{[l_1:\sigma_1,\ldots,l_n:\sigma_n,\ldots]} :: [\![l_1 : \sigma_1, \ldots, l_n : \sigma_n]\!]$

$d^{[l_1:\sigma_1,\ldots,l_n:\sigma_n,\ldots]} :: [\![l_1 : \sigma_1, \ldots, l_n : \sigma_n]\!]$

$[l_1 :: \sigma_1, \ldots, l_n : \sigma_n, \ldots] :: [\![l_1 : \sigma_1, \ldots, l_n : \sigma_n]\!]$

$t^{\langle\!\langle l_1:\sigma_1,\ldots,l_n:\sigma_n,\ldots\rangle\!\rangle} :: \langle\!\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n\rangle\!\rangle$

$d^{\langle\!\langle l_1:\sigma_1,\ldots,l_n:\sigma_n,\ldots\rangle\!\rangle} :: \langle\!\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n\rangle\!\rangle$

$[l_1 :: \sigma_1, \ldots, l_n : \sigma_n, \ldots] :: \langle\!\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n\rangle\!\rangle$

Figure 3: The Proof System for Kindings

where t^K stands for type variables with the kind tag K, d^K stands for description type variables with kind tag K. The set of kinds is given by the following syntax:

$$K ::= U \mid [\![l : \sigma, \ldots, l : \sigma]\!] \mid \langle\!\langle l : \sigma, \ldots, l : \sigma\rangle\!\rangle$$

The idea is that U denotes the set of all type schemes, $[\![l : \sigma, \ldots, l : \sigma]\!]$ denotes the set of record type schemes containing the set of all fields $l : \sigma, \ldots, l : \sigma$, and $\langle\!\langle l : \sigma, \ldots, l : \sigma\rangle\!\rangle$ denotes the set of variant type schemes containing the set of all fields $l : \sigma, \ldots, l : \sigma$. Figure 3 gives *kinding rules.*

Kind constraints on type variables are analogous to type constraints on variables. The only legal instantiation of a type variable of the form t^K is a type scheme σ such that we can derive the kinding $\sigma :: K$. This constraint is reflected in the following definition. A *kind preserving substitution* θ is a function from the set of kinded type variables to kinded type schemes such that $\theta(t^K) \neq t^K$ for only finitely many t^K and satisfies the conditions: (1) it maps description type variables to description type schemes and (2) $\theta(t^K) :: \theta(K)$, $\theta(d^K) :: \theta(K)$ for all t^K and d^K. Robinson's unification algorithm [Rob65] can be extended to kinded type schemes. Let E be a set of pairs of kinded type schemes. A kind preserving substitution θ is a *unifier* if $\theta(\sigma_1) = \theta(\sigma_2)$ for all pair $(\sigma_1, \sigma_2) \in E$. It is shown [Oho90] that:

Proposition 1 *There is an algorithm $\mathcal{U}$ which computes a most general unifier for any unifiable set E of pairs of kinded type schemes.* ∎

With these definitions, we can refine the notion of principal typing scheme. Let Σ be an assignment of kinded type schemes to a finite set of variables and σ be a kinded type scheme. A *principal kinded typing scheme* is a formula of the form $\Sigma \rhd e : \sigma$ such that $\mathcal{A} \rhd e : \tau$ is a typing of e if and only if there is a kind preserving substitution θ such that $\mathcal{A}(x) = \theta(\Sigma(x))$ for all $x \in dom(\Sigma)$ and $\tau = \theta(\sigma)$. Using a kind preserving unification algorithm, we can extend ML type inference to the set of Machiavelli's expressions:

Proposition 2 *There is an algorithm, $\mathcal{K}$, which, given any expression e, returns either a pair (Σ, σ) or failure such that if $\mathcal{K}(e) = (\Sigma, \sigma)$ then $\Sigma \rhd e : \sigma$ is a principal kinded typing scheme otherwise e has no typing.* ∎

The detailed proof can be found in [Oho90]. Here we only show the algorithm for the case of field selection $e.l$, which illustrates the use of kinded type variables.

$$
\begin{array}{l}
\mathcal{K}(e.l) = \text{let} \\
\qquad (\Sigma_1, \sigma_1) = \mathcal{K}(e) \\
\qquad \theta = \mathcal{U}(\{(\sigma_1, t_1^{[l:t_2^U]})\}) \text{ (where } t_1, t_2 \text{ are fresh)} \\
\quad \text{in} \\
\qquad \Sigma = \theta(\Sigma_1) \\
\qquad \sigma = \theta(t_2^U) \\
\quad \text{end}
\end{array}
$$

Just as legal ML programs correspond to principal typing schemes with empty type assignment, legal Machaivelli programs correspond to principal kinded typing schemes with empty type assignment. Machiavelli prints kinded type schemes as follows:

$t^U, \ldots$	'a,'b,...
$d^U, \ldots$	"a,"b,...
$t^{[l_1:\sigma_1,\ldots,l_n:\sigma_n]}, \ldots$	[('a)$l_1:\sigma_1,\ldots,l_n:\sigma_n$],...
$d^{[l_1:\sigma_1,\ldots,l_n:\sigma_n]}, \ldots$	[("a)$l_1:\sigma_1,\ldots,l_n:\sigma_n$],...
$t^{\langle\!\langle l_1:\sigma_1,\ldots,l_n:\sigma_n\rangle\!\rangle}, \ldots$	<('a)$l_1:\sigma_1,\ldots,l_n:\sigma_n$>,...
$d^{\langle\!\langle l_1:\sigma_1,\ldots,l_n:\sigma_n\rangle\!\rangle}, \ldots$	<("a)$l_1:\sigma_1,\ldots,l_n:\sigma_n$>,...

as already seen in examples. Thus

```
-> fun name x = x.Name;
>> val name = fn : [('a)Name:'b] -> 'b
```

is a representation of the the following kinded typing scheme:

$$\emptyset, \emptyset \rhd \textbf{fn}\ x \Rightarrow x.Name\ :\ t_1^{[Name:t_2^U]} \rightarrow t_2^U$$

Examples shown in figure 1 are to be similarly understood.

To summarize our progress to this point: we have augmented type schemes of ML with description types (which already exist in ML in a limited form) and kinded type variables. This has provided us with a type system that not only expresses the generic properties of field selection, but also allows sets to be uniformly treated in the language. However relational databases require more than the operations we have so far described, and it is to these that we now turn.

4 Operations for Generalized Relations

We are now going to show how we can extend Machiavelli to include the operations of the relational algebra, *projection* and *natural join* that are not covered by the operations for sets and records that we have so far developed. Before doing this, there are two important points to be made. The first is that, in order to achieve a general definition of these operations we are going to put an ordering on values and on description types. The ordering on types, although somewhat similar to that used by Cardelli [Car88] is in no sense a part of Machiavelli's polymorphism. This should be apparent from the fact that we have already incorporated field selection as a polymorphic operation without having to make use of such an ordering.

The second point is that the introduction of *join* complicates the presentation of the type inference system and increases the complexity of the type inference problem. The typing rule for *join* operation is associated a complex condition which can no longer be represented by some kind. To give a type scheme for *join*, we need to extend the notion of (kinded) typing schemes to *conditional* typing schemes [OB88] by adding syntactic conditions on instantiation of type variables. A similar problem was later observed in [Wan89] if one uses a record concatenation operation rather than join. Since we are primarily concerned with database operations, our inclination is to examine the record joining operation that naturally arises as a result of generalizing the relational algebra.

Our strategy in this section is first to provide a method for generalizing relational algebra over arbitrary description types. We then provide the additional typing rules, which have associated order constraints on the types. Finally we show that although there is no longer a principal typing scheme for a term, we can still provide a principal *conditional* typing scheme which represent the exact set of provable typings. We then describe the method to check the satisfiability of conditions before the evaluation of the term associated with the conditions. In other words, we are still able to guarantee that a program will not cause a type error.

4.1 Generalizing Relational Algebra

Our rationale for wanting to generalize relational operations is that, in keeping with the rest of the language, we would like them to be as "polymorphic" as possible. Since equality is essential to the definition of most of these operations, we cannot expect to generalize them to arbitrary terms of the language. Instead we content ourselves with their effect on description terms, which are those terms that can be typed with a description type. To achieve this end Machiavelli generalizes the following four operations to arbitrary description terms and introduces them as polymorphic functions in its type system:

eq(e_1,e_2)	*equality test,*
join(e_1,e_2)	*database join operation,*
con(e_1,e_2)	*operation for consistency check,*
project(e,δ)	*projection of d onto the type* δ.

The intuition underlying their generalization is the idea exploited in [BJO89] that database objects are *partial descriptions* of real-world entities and can be ordered by *goodness of description.* The polymorphic type system to represent these generalized operations has been developed in [Oho89]. In what follows, we describe how equality, join and projection are generalized to *finite* description terms. For the treatment of cyclic structures as well as the precise semantics of the type system for descriptions, the reader is referred to [Oho89].

We first consider join and equality. We claim that join in the relational model is based on the underlying operation that computes a join of tuples. By regarding tuples as partial descriptions of real-world entities, we can characterize it as a special case of very general operations on partial descriptions that *combines* two consistent descriptions. For example, if we consider the following non-flat tuples

t_1 = `[Name = [First = "Joe"]];`

and

t_2 = `[Name = [Last = "Doe"]]`

as partial descriptions, then the combination of the two should be

$t =$ `[Name = [First= "Joe", Last = "Doe"]]`.

This is characterized by the property that t is the *least upper bound* of t_1 and t_2 under the ordering induced by the inclusion of record fields. Denoting the ordering by $\sqsubseteq$, **join** is defined as:

$$\mathbf{join}(d_1, d_2) = d_1 \sqcup d_2$$

Equality in partial descriptions is an operation which tests the equality on the amount of information and is characterized by the equivalence relation induced by the information ordering, i.e.

$$\mathbf{eq}(d, d') = d \sqsubseteq d' \text{and } d' \sqsubseteq d$$

This approach also provides a uniform treatment of *null values* [Zan84, Bis81], which are essential to database programming involving incomplete information. Join and projection extend smoothly to data containing null values. However care must be taken [Lip79, IL84] to ensure that in using an algebra with these extended operations they provide the required semantics. To represent null values, we also extend the syntax of Machiavelli terms with:

`null(`b`)` *the null value of a base type* b
`<>` *the (polymorphic) null value of variant types*

All other incomplete values are those that are constructed by description term constructors.

The importance of these characterizations is that they do not depend on any particular data structure such as flat records. Once we have defined a (computable) ordering on the set of description terms which represents our intuition of the goodness of description, join and equality is generalized to *arbitrary* complex description terms. To get such an ordering, we first define the pre-order $\preceq$ on description terms. For finite descriptions, $\preceq$ is given as:

$$\begin{aligned}
c^b &\preceq c^b \quad \textit{for all constant } c^b \textit{ of type } b,\\
\mathtt{null}(b) &\preceq c^b \quad \textit{for all constant } c^b \textit{ of type } b,\\
\mathtt{null}(b) &\preceq \mathtt{null}(b) \quad \textit{for any base type } b\\
[l_1 = d_1, \ldots, l_n = d_n] &\preceq [l_1 = d'_1, \ldots, l_n = d'_n, \ldots] \quad \textit{if } d_i \preceq d'_i\ (1 \le i \le n),\\
\mathtt{<>} &\preceq \mathtt{<>},\\
\mathtt{<>} &\preceq \mathtt{<}l = d\mathtt{>} \quad \textit{for any description } d,\\
\mathtt{<}l = d\mathtt{>} &\preceq \mathtt{<}l = d'\mathtt{>} \quad \textit{if } d \preceq d',\\
r &\preceq r \quad \textit{for any reference } r\\
\{d_1, \ldots, d_n\} &\preceq \{d'_1, \ldots, d'_m\} \quad \textit{if } \forall d' \in \{d'_1, \ldots, d'_m\}.\, \exists d \in \{d_1, \ldots, d_n\}.\, d \preceq d'
\end{aligned}$$

The rule for sets is defined to capture the properties of sets in database programming. $\preceq$ fails to be anti-symmetric because of this rule. An ordering is obtained by taking induced equivalence relation and regarding a description term as a representative of an equivalence class. In what follows, we denote by $\sqsubseteq$ the ordering induced by the preorder $\preceq$. Since the ordering relation and the least upper bound are shown to be computable, our characterization of **join** and **eq** immediately gives their definitions on general description terms. The equality (**eq**) is a generalization of *structural equality* to sets and null values. Figure 4 shows an example of a join of complex descriptions. This definition of **join** is a faithful generalization of the join in the relational model. In [BJO89] it is shown that:

```
r1 = {[Pname = "Nut",Supplier = { [Sname = "Smith",City = "London"],
                                  [Sname = "Jones",City = "Paris"],
                                  [Sname = "Blake",City = "Paris"]}],
      [Pname = "Bolt",Supplier = { [Pname = "Blake",City = "Paris"],
                                   [Sname = "Adams",City = "Athens"]}]}

r2 = {[Pname = "Nut",Supplier = {[City = "Paris"]},Qty = 100],
      [Pname = "Bolt",Supplier = {[City="Paris"]},Qty = 200]}

join(r1,r2)= {[Pname = "Nut",Supplier ={[Sname = "Jones",City = "Paris"],
                                       [Sname = "Blake",City = "Paris"]}, Qty = 100],
              [Pname = "Bolt",Supplier ={[Sname = "Blake",City = "Paris"]}, Qty = 200]}
```

Figure 4: Natural join of higher-order relations

Proposition 3 *If r_1, r_2 are first-normal form relations then* $\mathbf{join}(r_1, r_2)$ *is the natural join of r_1 and r_2 in the relational model.* ∎

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.

We turn our attention to projection. In the relational model, it is defined as a projection on a set of labels. We generalize it to an operation which project a complex description onto its "substructure". In a programming language, a structure of data is represented by a *type* and we define projection as an operation specified by its target type. Recall that the syntax of description types is

$$\delta ::= b_d \mid [l:\delta,\ldots,l:\delta] \mid \langle l:\delta,\ldots,l:\delta\rangle \mid \{\delta\} \mid (rec\ v.\ \delta(v))$$

Projection becomes an operation indexed by a description type. `project(`x`,`δ`)` is the operation which, given a description x whose type is "bigger" than δ, returns a description of type δ by 'throwing away" part of its information. The following is a simple projection on flat relation:

```
project({ [Name = "J. Doe", Age = 21, Salary = 21000],
          [Name = "S. Jones", Age = 31, Salary = 31000] },
        {[Name:string, Salary:int]})

= { [Name = "J. Doe", Salary = 21000],
    [Name = "S. Jones", Salary = 31000] }
```

To define such an operation, we use an ordering on description types to model our intuition that the structure represented by one description type "contains" the other. For finite description types, the appropriate ordering is given as:

$$b_d \ll b_d$$

$$
\begin{aligned}
[l_1 : \delta_1, \ldots, l_n : \delta_n] &\ll [l_1 : \delta'_1, \ldots, l_n : \delta'_n, \ldots] \text{ if } \delta_i \ll \delta'_i \ (1 \leq i \leq n) \\
\langle l_1 : \delta_1, \ldots, l_n : \delta_n \rangle &\ll \langle l_1 : \delta'_1, \ldots, l_n : \delta'_n \rangle \text{ if } \delta_i \ll \delta'_i \ (1 \leq i \leq n) \\
\{\delta_1\} &\ll \{\delta_2\} \text{ if } \delta_1 \ll \delta_2
\end{aligned}
$$

By this ordering and the typing relation already etablished in section 3, projection has the following general definition:

$$\mathbf{project}(x, \delta) = \bigsqcup \{d | d \sqsubseteq x, d : \delta\}$$

which is a computable function for any description type δ.

4.2 Extended Expressions and Their Evaluation

The syntax of expressions is extended with the term constructors **join**, **con**, and **project** we have just described:

$$e ::= c_\tau \mid \cdots \mid \mathbf{join(e,e)} \mid \mathbf{con(e,e)} \mid \mathbf{project(e,}\delta\mathbf{)}$$

We extend the evaluation rules for expressions described in section 3 with the rules for the above new term constructors and **eq**. Note that they are only applicable to description terms. A description term d denote an equivalence class of regular trees induced by the ordering we have just described. We write $D(d)$ for the equivalence class denoted by d. The evaluation rules for those term constructors are given as:

$\mathbf{join}(d_1, d_2) \rightarrowtail d_3$	*if* $D(d_3) = D(d_1) \sqcup D(d_2)$
$\mathbf{con}(d_1, d_2) \rightarrowtail \mathtt{true}$	*if* $D(d_1) \sqcup D(d_2)$ *exists*
$\mathbf{con}(d_1, d_2) \rightarrowtail \mathtt{false}$	*if* $D(d_1) \sqcup D(d_2)$ *does not exist*
$\mathbf{project}(d_1, \delta) \rightarrowtail d_2$	*if* $D(d_2)$ *is the leas element of* $\{D(d) \mid D(d) \sqsubseteq D(d_1), d : \delta\}$
$\mathbf{eq}(d_1, d_2) \rightarrowtail \mathtt{true}$	*if* $D(d_1) \sqsubseteq D(d_2)$ *and* $D(d_2) \sqsubseteq D(d_1)$
$\mathbf{eq}(d_1, d_2) \rightarrowtail \mathtt{false}$	*if* $D(d_1) \not\sqsubseteq D(d_2)$ *or* $D(d_2) \not\sqsubseteq D(d_1)$

As we have mentioned, there are generic algorithms to compute these functions.

4.3 Type Inference for Relational Algebra

Figure 5 gives two simple examples of the typing schemes that are inferred by Machiavelli. The type scheme for `join3`, a join of three records is given as a type scheme `("a * "b * "c) -> "d` together with a set of conditions { `"d = jointype("a,"e), "e = jointype("b,"c)` }. There is clearly extra work to be done for we have to infer precise conditions and to verify that there are instances of the type variables that satisfy the conditions.

Figure 6 gives the additional typing rules for the operations **join**, **project**, and **con**, which must be considered in conjunction with those in figure 2. In order to include these operations we explicitly introduce syntactic conditions on substitution of type variables that represent the last three forms of constraint that appear in these rules; they are $\delta_1 \sqcup \delta_2$ exists, $\delta = \delta_1 \sqcup \delta_2$, and $\delta_2 \ll \delta_1$. In fact we only need to consider the last two forms of constraint since $\delta_1 \sqcup \delta_2$ will exist whenever we can find a type $\delta_3 = \delta_1 \sqcup \delta_2$. To represent them we introduce the following syntactic conditions:

```
-> 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]
```

Figure 5: Some Simple Relational Examples

(CON) $$\frac{\mathcal{A} \rhd e_1 : \delta_1 \qquad \mathcal{A} \rhd e_2 : \delta_2}{\mathcal{A} \rhd \mathbf{con}(e_1, e_2) : bool} \quad \text{if } \delta_1 \sqcup \delta_2 \text{ exists}$$

(JOIN) $$\frac{\mathcal{A} \rhd e_1 : \delta_1 \qquad \mathcal{A} \rhd e_2 : \delta_2}{\mathcal{A} \rhd \mathbf{join}(e_1, e_2) : \delta} \quad \text{if } \delta = \delta_1 \sqcup \delta_2$$

(PROJECT) $$\frac{\mathcal{A} \rhd e : \delta_1}{\mathcal{A} \rhd \mathbf{project}(e_1, \delta_2) : \delta_2} \quad \text{if } \delta_2 \ll \delta_1$$

Figure 6: The Typing Rules for Relational Operations

1. $\sigma = jointype(\sigma, \sigma)$, and
2. $lessthan(\sigma, \sigma)$.

Note the difference between $\delta_3 = \delta_1 \sqcup \delta_2$ and $\sigma_3 = jointype(\sigma_1, \sigma_2)$. The former is a property on the relationship between three description types. On the other hand, the latter is a syntactic formula denoting the constraint on substitutions of type variables to represent such property. Similarly for $\delta_1 \ll \delta_2$ and $lessthan(\sigma_1, \sigma_2)$. The following definition provides the meaning of those syntactic conditions. A kind preserving ground substitution θ *satisfies* a condition c if

1. if $c \equiv \sigma_1 = jointype(\sigma_2, \sigma_2)$ then $\theta(\sigma_1), \theta(\sigma_2), \theta(\sigma_3)$ are all description types and $\theta(\sigma_1) = \theta(\sigma_2) \sqcup \theta(\sigma_3)$,
2. if $c \equiv lessthan(\sigma_1, \sigma_2)$ then $\theta(\sigma_1), \theta(\sigma_2)$ are description types and $\theta(\sigma_1) \ll \theta(\sigma_3)$.

θ satisfies a set C of conditions if it satisfies each member of C.

Combining this with the mechanism of kinded type schemes given in section 3, we can extend our inference algorithm. Let C be a set of conditions, Σ be an assignment of kinded type schemes to variables. A *conditional typing scheme* is a formula of the form $C, \Sigma \rhd e : \sigma$ such that if a kind preserving substitution θ satisfies C then $\theta(\Sigma) \rhd e : \theta(\sigma)$ is a derivable typing. A typing $\mathcal{A} \rhd e : \tau$ is an *instance* of a conditional typing scheme $C, \Sigma \rhd e : \sigma$ if there is a kind preserving substitution θ such that θ satisfies C, $\mathcal{A}(x) = \theta(\Sigma(x))$ for all $x \in dom(\Sigma)$, and $\tau = \theta(\sigma)$. A conditional typing scheme $C, \Sigma \rhd e : \sigma$ is *principal* if any derivable typing for e is an instance of it. The following result establishes the complete inference of principal conditional typing schemes.

Proposition 4 *There is an algorithm which, given any raw term e, returns either* failure *or a triple* (C, Σ, σ) *such that if it returns* (C, Σ, σ) *then* $C, \Sigma \rhd e : \sigma$ *is a principal conditional typing scheme, otherwise e has no typing.* ∎

A proof of this, which also gives the type inference algorithm for Machiavelli, is based on the technique we have developed in [OB88] which established the theorem for a sublanguage of Machiavelli. A complete proof and a complete type inference algorithm can be found in [Oho89]. For example, the type `("a * "b * "c) -> "d where { "d = "a lub "e, "e = "b lub "c }` of the three-way join **join3** is the representation of of the principal conditional typing scheme:

$$\{d_1 = jointype(d_2, d_3), d_3 = jointype(d_4, d_5,)\}, \emptyset \rhd \mathbf{fn}(x, y, z) => join(x, join(y, z)) \ : \ (d_2 * d_4 * d_5) \rightarrow d_1$$

It is therefore tempting to identify legal Machiavelli programs with principal conditional typing schemes. There is however one problem in this approach. As we have mentioned at the beggining of this section, the definition of conditional typing schemes does not imply that they have an instance. This happens because the set C of conditions in a typing scheme may not be satisfiable. In such case, the term has no typing and should therefore be regarded as a term with type error. In order to achieve a complete static type-checking, we therefore need to check the satisfiability of the set of conditions. Unfortunately, however, the satisfiability checking cannot be made efficient since it is shown that [OB88] the introduction of *join* makes the type inference problem for the simply typed lambda calculus itself NP-complete, while the construction of a conditional type scheme can be done in polynomial time. A practical solution we adopt here is to *delay* the satisfiability check of a condition until its type variables are fully instantiated. Once the types of all type variables in a condition are known then the satisfiability of the condition can be efficiently checked and the condition can be eliminated. Since the reduction associated with **join** is performed only after actual parameters are supplied, this method also detects all run time type errors. We therefore identify legal Machiavelli programs with principal conditional typing schemes where the only conditions are those that contain type variables.

This strategy 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 7 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 8 shows some simple query processing for the database example in figure 7. Note the use of **join** and other relational operations on "non-flat" relations.

This approach to defining generalized relational operations completely eliminates the problem of "impedance mismatch" between relational databases and a programming language. 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 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 9 shows a simple implementation of a polymorphic transitive closure function. By using renaming operation, this function can be used to compute the transitive closure of any binary relation. Figure 10 shows query processing on the example database using polymorphic functions. The function **cost** taking a part record and a set of such records as arguments computes the total cost of the part. Note that scope of type variables is limited to a single type scheme, so that instantiations of **"a** in the type of **cost** have nothing to do instantiations of **"a** in the type of **expensive-parts**. Also, the apparent

```
-> parts;
>> val it = {[Pname="bolt",P#=1,Pinfo=<Base= [Cost=5]>],
                  ...
             [Pname="engine",P#=2189,
              Pinfo=<Composite = [SubParts={[P#=1,Qty=189], ...},
                                  AssemCost=1000]>],...}
       :  {[Pname:string,P#:int,
            Pinfo:<Base:[Cost:int],
                   Composite:[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 7: A Part-Supplier Database in Generalized Relational Model

```
(* Select all base parts *)
-> join(parts,{[Pinfo=<Base=[]>]});
>> val it = {[Pname="bolt", P#=1, Pinfo=<Base=[Cost=5]>],...}
       :  {[Pname:string,P#:int,
            Pinfo:<Base:[Cost:int],
                   Composite:[SubParts:{[P#:int,Qty:int]}, AssemCost:int]>]}

(* List part names supplied by "Baker" *)
-> select x.Pname
   from x <- join(parts,supplied_by)
   where Join3(x.Suppliers,suppliers,{[Sname="Baker"]}) <> {};
>> {"bolt",...} :  {string}
```

Figure 8: Some Simple Queries

```
-> fun Closure R =
       let val r = select [A=x.A,B=y.B]
                   from x <- R, y <- R
                   where eq(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;
>> val Closure = fn :  {[A:"a,B:"b]} -> {[A:"a,B:"b]}
```

Figure 9: A Simple Implementation of Polymorphic Transitive Closure

complexity of the type of **cost** could be reduced by giving a name to the large repeated sub-expression. Without proper integration of the data model and programming language, defining such a function and checking type consistency is a rather difficult problem. Moreover, the functions **cost** and **expensive_parts** are both parameterized by the relation (**partdb**) and their polymorphism allows them to 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 are uniformly applicable.

5 Manipulation of Object-Oriented Databases

While we make no claim that Machiavelli exhibits all the desirable properties of an object-oriented database language, we believe that the inheritance of methods that is implicit in functions that exploit field selection captures a basic property of object-oriented databases: the ability to describe and manipulate data models that express inheritance. In this section we show how to represent certain important features of object-oriented databases within the type system we have developed.

We shall single out two features of object-oriented data models [LRV88, ABD+89] that set them apart from other data models. The first is the idea of object identity which, as we suggested in subsection 2.4, can be represented by reference types. 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. Even if we allow values to have multiple types [Car88], it is not clear how we generalize this property to sets of values in order to allow heterogeneous sets. This is something we shall discuss in section 7.

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

```
(* a function to compute the total cost of a part *)
-> fun cost(p,partdb) =
       case p.Pinfo of
            <Base = x> => x.Cost,
            <Composite = x> =>
               hom(fn(y)=>y.SubpartsCost,+,x.AssemCost,
                    select [SubpartsCost=cost(z,partdb) * w.Qty,P#=w.P#]
                    from w <- x.SubParts, z <- partdb
                    where eq(z.P#,w.P#))
       endcase;

>> val cost = fn
       :  ([("a) Pinfo:<Base:[("b) Cost:int],
                        Composite:[("c) SubParts:{[("d) P#:"e,Qty:int]},
                                           AssemCost:int]>,
                  P#:"e]
           * {[("a) Pinfo:<Base:[("b) Cost:int],
                           Composite:[("c) SubParts:{[("d) P#:"e,Qty:int]},
                                              AssemCost:int]>,
                     P#:"e]})
          -> int

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

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

Figure 10: Query Processing Using Polymorphic Functions

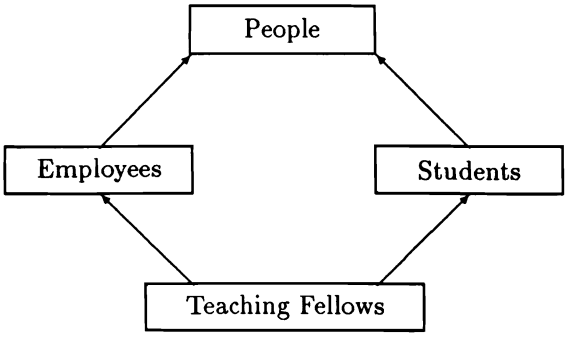

Figure 11: A Simple Class Structure

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 certain types, 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*. A database object will, in general, be a reference to a structure whose type, say *PersonObj*, may rather complicated and will describe all possible states of a Person object including an indicator of whether or not it is an Employee and its attributes as an Employee . A database (or a part of it) will consist of a set `DB` of such objects, i.e. a value of type $\{PersonObj\}$. A *view* of `DB` is a set of relatively simple records in which we "reveal" a part of the structure of each member of `DB` 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 types of possible views of set `DB`. 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 (**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 an *object type* as any record type that contains an `Id` field, which will be assumed to be some reference type. We shall single out object types for special treatment in the language by adding some additional functions that are applicable only to object types.

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 11 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 12. We should emphasize that the definitions in figure 12 are not Machiavelli

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

Figure 12: Some Machiavelli Types

definitions, they are simply shorthands for certain types that we shall use in describing the examples that follow. The 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 teaching 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 to view this as a set of values of type *Person* we can simply write a function **PersonView**, as shown in figure 13, which can be applied to any set of type $\{PersonObj\}$ to extract the **Name** field, which is always available, and produces a set of type $\{Person\}$. The function **EmployeeView** similarly applies to a set of type $\{PersonObj\}$, but selects only those records that have a defined **Salary**, and produces a set of type $\{Employee\}$. Quite general types will be inferred from these definitions; for example the type inferred for **EmployeeView** is

```
{ref([('a) Name:'b, Salary:<('c) Value:'d>])} ->
   {[Name:'b, Salary:'d, Id:ref([('a) Name:'b, Salary:<('c) Value:'d>])]}
```

and the type $\{PersonObj\}$ -> $\{Employee\}$ is an instance of this type.

In the definition of **TFView**, the join of two views provides both the intersection of the two sets of objects (i.e. expressions of object types) and the inheritance of methods. If δ_1, δ_2 are object types, then $\delta_2 \ll \delta_1$ implies that **project**$(View_{\delta_1}(S), \{\delta_2\}) \subseteq View_{\delta_2}(S))$ where $View_{\delta_1}$ and $View_{\delta_2}$ denote the corresponding viewing functions on object types δ_1 and δ_2. 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 **DB** of type $\{PersonObj\}$. Then **join(StudentView(DB), EmployeeView(DB))** always represents the set of objects that are both student and employee. Moreover, methods defined on **StudentView(DB)** and **EmployeeView(DB)** 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(DB)**, is also applicable to **TFView(DB)**. Figure 14 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 sets of object types, the union of sets of object types 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

```
fun PersonView(S) = select [Name=(!x).Name, Id=x]
                    from x <- S
                    where true;
fun EmployeeView(S) =
    hom(case (!x).Salary of
              <Value=y> => {[Name=(!x).Name, Salary=y, Id=x]},
              else => {}
         endcase,union,{},S)

fun StudentView(S) =
    hom(case (!x).Advisor of
              <Value=y> => {[Name=(!x).Name, Advisor=y, Id=x]},
              else => {}
         endcase,union,{},S)

fun TFView(S) =
    hom(case (!(x.Id)).Course of
              <Value=y> => {join(x,[Course=y])}
              else => {}
         endcase,union,{},join(StudentView(S),EmployeeView(S)))
```

Figure 13: Definition of Views

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

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

Figure 14: Using join to find an intersection

$\delta_1 \sqcap \delta_2$ exists. Let s_1, s_2 be two sets having types $\{\delta_1\}$, $\{\delta_2\}$ respectively. Then **unionc**(s_1, s_2) satisfies the following equation:

$$\textbf{unionc}(s_1, s_2) = \textbf{project}(s_1, \{\delta_1 \sqcap \delta_2\}) \cup \textbf{project}(s_2, \{\delta_1 \sqcap \delta_2\})$$

which is reduced to the standard set-theoretic union when $\delta_1 = \delta_2$. This operation can be used to give a union of sets of object types event though their types differ. For example, `unionc(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 other sets 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 sets of object types.

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. "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.

6 Data Abstraction and Inheritance

In the previous section, we haved given example of a simple hierarchy of object types (in figure 11) and showed how Machiavelli's polymorphic type system represents both method inheritance and inclusion of extents. This, however, depends on the explicit types of the implementations of these objects. For example, the type of *Employee* is explicitly defined as

`[Name:string, Age:int, Salary:int, Id:`*PersonObj*`]`

where *PersonObj* is another concrete type given in figure 12. A drawback to this approach is that it does not combine data abstraction with inheritance in the same sense as object-oriented languages do this. Exposing concrete representations is in many cases undesirable. In the above example, the availability of the type of `Id` field is particularly dangerous as the user can access and change any part of the object. As argued in [CDMB90], database views should be integrated with data abstraction mechanism to provide protection mechanism.

A well known data abstraction mechanism in a static type system is to use *abstract data types* which has been implemented in several polymorphic type systems such as Standard ML [HMT88] and Miranda [Tur85]. These type systems, however, do not allow abstract data types to be organized into a class hierarchy. This means that method inheritance achieved by polymorphism does not extend to abstract types. Galileo [ACO85] integrates inheritance and class hierarchy in a static type system by combining the subtype relation and abstract type declarations. However, Galileo supports neither polymorphism nor type inference.

In object-oriented languages [GR83] each data element (object) belongs to a unique member of a user defined class hierarchy. Objects can be manipulated only through *methods* defined in its class and super

classes. This mechanism nicely integrates data abstraction and method inheritance. We would like to extend our polymorphic type system with this feature. Jategaonkar and Mitchell [JM88] suggested the possibility of using their type inference method to extend ML's abstract data types to support inheritance. In [OB89], we have developed a formal system for *parametric classes* that achieves a proper intergration of ML style parametric abstract data types and multiple inheritance in object-oriented programming. Based on this result, we can extend the Machiavelli's type system with data abstraction and multiple inheritance. In the extended system, the programmer can define a hierarchy of classes. A class can be parametric and can contain multiple inheritance declarations. The type correctness of such a class definition (including the type consistency of all inherited methods) is statically checked by the type system. Moreover, apart from the type assertions needed in the definition of a class, the type inference mechanism we have described in section 3 extends to these parametric classes. In [OB89] it is shown that the type system with class definition is sound with respect to the underlying polymorphic type system (i.e. the one we have defined in section 3) and it has a complete type inference algorithm. This section explains this feature through examples. The reader is referred to [OB89] for the full description of the type system and type inference method with class definition, which require a certain amount of mathematical development and is beyond the scope of this paper.

First we must note one design decision we made in developing Machiavelli's classes. Different from ML's abstract types, Machiavelli's classes inherit equality from their implementation types. We adopt this because our main goal of classes is to provide a protection mechanism in database programming involving sets, which require equality. A richer language might, as in Ada [IBH*79], allow a choice of whether equality is inherited from the underlying representation or whether it is to be hidden or redefined.

In the previous section, we have defined the type *PersonObj* and four viewing functions. We will make them abstract by using class definition. We assume that the variable DB of type $\{PersonObj\}$ is defined, which is protected by some form of scoping mechanism. Note that we continue to use *PersonObj* as an abbreviation for the actual type definition. We encapsulate the concrete structure of *PersonObj* by defining the following class:

```
class PObj = PersonObj with
  fun NewPersonView () = select [Name=(!x).Name, Id=x]
                         from x <- DB
                         where true
      :  unit -> {[Name:string, Id:PObj]}
  fun NewEmployeeView S =
      hom(fn x => case (!x).Salary of
                      <Value=y> => {[Name=(!x).Name, Salary=y,Id=x]},
                      else => {}
                  endcase, union(x,y), {}, DB)
      :  unit -> {[Name:string, Salary:int, Id:PObj]}

  fun NewStudentView () =
      hom(fn x => case (!x).Advisor of
                      <Value=y> => {[Name=(!x).Name, Advisor=y,Id=x]},
                      else => {}
                  endcase, union(x,y), {}, DB)
      :  unit -> {[Name:string, Advisor:PObj, Id:PObj]}
```

```
fun NewTFView () =
    hom(case (!x).Course of
              <Values=y> => {join(x,[Course=y])}
              else => {}
         endcase,union,{},join(NewStudentView(),NewEmployeeView()))
    : unit -> {[Name:string, Salary:int, Advisor:PObj, Course:string Id:PObj]}

fun increment_obj_age p = (p:=modify(!p,Age,(!p).Age + 1);p)
    : PObj -> PObj
    ⋮
end
```

Outside of the definition, the actual structure of objects of the type *PersonObj* is hidden and can only be manipulated through the explicitly defined set of interface functions (methods). This is enforced by treating classes and the set of interface functions as if they were base types and primitive operations associated with them. As in Miranda's abstract data types, we require the programmer to specify the type (type-scheme) of each method. Note that the value DB is embedded in this class definition. This technique, exploited in [CDMB90], is necessary to hide the type information of *PersonObj*. For users who have no need for access to the value DB itself, this definition successfully hides the representation type of DB. They can only manipulated the database by explicitly defined viewing functions and any other functions such as **increment_obj_age** within the class definition.

So far Machiavelli's classes behave similar to abstract types found in ML and Miranda. However classes may be organized in a hierarchy connected by multiple inheritance declarations. We demonstrate this feature by defining a hierarchy of views. The class PObj encapsulates the concrete structure of *PersonObj* but not the types that represents views. We also want to encapsulate them to prevent meaningless manipulation on views while maintaining the advantages of method inheritance discussed in the previous section. We start with the class **Person** which is the maximum class in the class hierarchy.

```
class Person = [Name:string, Age:int, Id:PObj] with
    fun persons() = NewPersonView() :  unit -> {Person};
    fun name(p) = p.Name :  sub -> string;
    fun age(p) = p.Age :  sub -> int;
    fun increment_age(p) =
        modify(modify(p, Id, increment_obj_age(p.Id)), Age, p.Age +1)
        :  sub -> sub
end
```

Note that the fourth function **increment_age** increment both the **Age** field in the view and in the actual object. The keyword **sub** in the type specifications of methods is a special type variable representing all possible subclasses of the class which are to be defined later. It is to be regarded as an assertion by the programmer (which may later prove to be inconsistent with a subclass definition) that a method can be applied to values of any subclass. This definition is type consistent and the Machiavelli compiler generates the following bindings:

```
class Person with
    persons :  unit -> {Person}
```

```
name :  ("a < Person) -> string
age :  ("a < Person) -> int
increment_age :  ("a < Person) -> ("a < Person)
```

Note that **Person** in the type schemes is not shorthand but a class name, which is a part of Machiavelli type system. **("a < Person)** is another *kinded type variable* whose range is the set of all subclasses of the class **Person**. At this moment, there is no proper subclass of **Person** and therefore the range of **"a** is the singleton set of **Person** and the above class definition behaves similarly to ML's abstract types. But we can define a number of useful subclasses of **Person**, which inherits method defined in **Person**. The following is the definition for the class **Employee**.

```
class Employee = [Name:string, Age:int, Salary:int, Id:PObj]
isa Person with
  fun employees() = NewEmployeeView() :  unit -> {Employee}
  fun salary(x) = x.Salary :  sub -> int
end
```

which inherits the methods **name**, **age** and **increment_age**, but not **persons** from the class **Person** because there is no **sub** in the type specification of **persons**. ¿From this definition, Machiavelli compiler prints the following information.

```
class Employee isa Person with
     employees :  unit -> {Employee}
     salary :  ("a < Employee) -> int
inherited method
     name :  ("a < Person) -> string
     age :  ("a < Person) -> int
     increment_age :  ("a < Person) -> ("A < Person)
```

In order to preserve complete static type inference, we have given the complete record type required to implement **Employee**, not just the additional fields we need to add to the implementation of **Person**. It is possible that for simple record extensions such as these we could invent a syntactic shorthand that is more in line with object-oriented languages. Continuing in this fashion, we can define the class **Student** and **TeachingFellow** to complete the previous example.

```
class Student = [Name:string, Advisor:PersonObj, Id:PObj]
isa Person with
   fun students () = NewStudentView() :  unit -> {Student}
   fun advisor x = x.Advisor :  sub -> PObj
end

class TeachingFellow = [Name:string, Salary:int, Advisor:PObj,
                        Course:String, Id:PObj]
isa {Employee,Student} with
   fun teaching_fellows () = NewTFView() :  unit -> {TeachingFellow}
   fun Course x = x.Course :  sub -> string
end
```

The second one these illustrate the use of multiple inheritance.

It should be stressed that the method we have developed in [OB89] allows static checking of the type correctness of these class definitions containing multiple inheritance declarations. Moreover, a principal conditional typing scheme is always inferred for expressions containing methods defined in classes. For example, for the following function

```
fun average_salary S = divide(hom(fn x=>salary(x),+,0,S),hom(fn x=> 1,+,0,S));
```

which computes the average salary using the method **salary** defined in the class **Employee**, the type system infers the following principal conditional typing scheme:

```
{("a < Employee)} -> int
```

This function can be applied to any set of type $\{\tau\}$ such that τ is a subclass of **Employee**. In the above example, it can be applied to {**Employee**} and {**TeachingFellow**}. The type correctness of such applications is statically checked.

To demonstrate the use of type parameters, consider how a class for lists might be constructed. We start from a class which defines a "skeletal" structure for lists.

```
class pre_list = (rec t.<Empty:unit, List:[Tail:t]>)
with
     nil = <Empty = ()> :  sub;
     fun tl(x) = case x of
                    <Empty = y> => ... error ...;
                    <List = z> => z.Tail;
                endcase :  sub -> sub
     fun null(x) = case x of
                      <Empty = y> => true;
                      <List = z> => false;
                  endcase :sub -> bool;
end
```

By itself, the class **pre_list** is useless for it provides no method for constructing non-empty lists. We may nevertheless derive a useful subclass from it.

```
class list('a) =
     (rec t. <Empty:unit,List:[Head:'a,Tail:t]>
isa pre_list
with
     fun cons(h,t) = <List=[Head=h,Tail=t]>
         :  ('a*sub) -> sub;
     fun hd(x) = case x of
                     <Empty=y> => ... error ...;
                     <List=z> => z.Head;
                end :  sub -> 'a;
end
```

which is a class for polymorphic lists much as they appear in ML. Separating the definition into two parts may seem pointless here but we may be able to define other useful subclasses of *pre_list*. Moreover, since a may itself be a record type, we may be able to define further useful subclasses of *list*. For example, we could construct a class

```
class genintlist('b) =
  (rec t. <Empty:unit,
           List:[Head:[Ival:int, Cont:'b],
                 Tail:t]>)
isa list([Ival:int, Cont:'b])
with
  ⋮
end
```

which could be used, say, as the implementation type for a "bag" of values of type `'b`. In this case all the methods of `pre_list` and `list` are inherited.

7 Conclusions and Directions for Further Investigations

Throughout this paper we have stressed the fact that we only have an experimental version of Machiavelli, which lacks many of the useful features of other programming languages. While we believe that the type system of Machiavelli can be used as the basis for a full-blown programming language, this claim can only be proved by a careful analysis of the addition of new features and, ultimately, by a full-blown implementation. Let us briefly mention some of the additions.

Standard ML of New Jersey [Mac88b] incorporates a number of features that we have not mentioned here. It exploits *pattern matching* to bind variables and has a system of *exceptions*. We believe that both of these could safely be added to Machiavelli and would be useful in many of the examples in this paper. More problematic is the system of *modules* [Mac86] in this language. Modules bear some relationship to the classes (or abstract data types) described in this paper. However it remains to be seen whether the sophisticated schemes for defining and instantiating modules that are available in SML can be combined with the typechecking for classes with inheritance that we have described here.

Turning to object-oriented databases or, more generally, database programming languages, all such languages have a layer that supports some kind of persistent database. While the implementation of such storage systems is a serious technical problem, we see no difficulty in exploiting such a system to provide a real database manager for Machiavelli. However, operating systems do not respect the type systems of programming languages and, as with files in most programming languages, on opening a database one must either do a dynamic type check or trust that the declared type of the database conforms to that specified in the program. We believe the former is the only satisfactory option and it is therefore essential to find ways of encapsulating the type of a database with the database and to incorporate dynamic type-checking into the language at certain points. In order to do this we must study the us of *dynamic* types [ACPP89].

Other issues raised by object-oriented languages include late binding and overloading. The former should not present a problem for the type system if we are able to constrain a given method to just one type. However some object-oriented languages allow overloading of methods. The type of the result of a method may depend in an *ad hoc* way on the type of some input parameter or on the class of the instance. Some recent results [AKW90] show the undecidability of type checking (let alone type inference) for a rather general form of overloading. It remains to be seen whether the type system of Machiavelli can be used in conjunction with some more restricted form of overloading.

The problem of *heterogeneous* structures is not, to our knowledge, addressed in any statically typed language, and yet it is common in database work to want to deal with a collection of (say) records of different types. Consider, for example, the set

```
{[Name = "Joe", Age = 21],
 [Name="John", Age = 23, Dept = "Sales"],
 [Name="Mary", Dept = "Research"]}
```

which is not a legal expression in Machiavelli, nor is a bulk structure of this form possible in most statically typed languages. Yet there are some properties of the members of this set, for example they are all of kind $[Name : string]$. If this information could be represented in the type system, then it might be possible to make such expressions legal and justify the apparently reasonable selection of the **Name** field from each member of this set. Recent investigations by the authors indicate that the right way to approach the problem of heterogeneity is to exploit a form of dynamic value whose type is "partially abstract". The advantage to dealing with heterogeneous structures is that it appears to provide a more general solution to the subset/subtype paradox mentioned at the beginning of section 5. Here, the inclusion ordering is derived from an ordering on kinds rather than one on types which, as we have observed, is not needed to express the generic properties of field selection.

8 Acknowledgements

Val Breazu-Tannen deserves our special thanks. He has contributed to many of the ideas in this paper and has greatly helped us in our understanding of type systems. We would also like to acknowledge helpful conversations with Serge Abiteboul, Malcolm Atkinson, Luca Cardelli, John Mitchell, Rick Hull and Aaron Watters.

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.

[ABD+89] M.P. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrick, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proceedings of the First Deductive and Object-Oriented Database Conference*, Kyoto, Japan, DEcember 1989.

[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.

[ACPP89] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, 1989.

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

[AKW90] S. Abiteboul, P. Kanellakis, and E. Waller. Method schemas (preliminary report). In *Proceedings of ACM Symposium on Principles of Database Systems*, 1990.

[Aug84] L. Augustsson. A compiler for lazy ML. In *Symposium on LISP and Functional Programming*, pages 218–227. ACM, 1984.

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

[Bis81] J. Biskup. A formal approach to null values in database relations. In *Advances in Data Base Theory Vol 1*, Prenum Press, New York, 1981.

[BJO89] P. Buneman, A. Jung, and A. Ohori. Using powerdomains to generalize relational databases. *Theoretical Computer Science*, To appear. Available as a technical report from Department of Computer and Information Science, University of Pennsylvania, 1989.

[BTBO89] V. Breazu-Tannen, P. Buneman, and A. Ohori. Can object-oriented databases be statically typed? In *Proc. 2^{nd} International Workshop on Database Programming Languages*, pages 226 – 237, Gleneden Beach, Oregon, June 1989. Morgan Kaufmann Publishers.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. (Special issue devoted to Symp. on Semantics of Data Types, Sophia-Antipolis, France, 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.

[CDMB90] R. Connor, A. Dearle, R. Morrison, and F. Brown. Existentially quantified types as a database viewing mechanism. In *Proceedings of 2^{nd} International Conference on Extending Data Base Technology*, Venice, Italy, March 1990.

[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.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley, 1983.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. American Mathematical Society*, 146:29–60, December 1969.

[HK87] R. Hull and R. King. Semantic database modeling: Survey, applications and research issues. *Computing Surveys*, 19(3), September 1987.

[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.

[HW89] P. Hudak and P. (editors) Wadler. Report on the programming language Haskell, a non-strict, purely functional language, version 1.0. Technical report, University of Glasgow, 1989. Pre-release Draft for FPCA '89.

[IBH*79] J.H. Ichbiah, J.G.P. Barnes, J.C. Heliard, B. Krieg-Bruckner, O. Roubine, and B.A. Wichmann. Rationale of the design of the programming language Ada. *SIGPLAN Notices*, 14(6), 1979.

[IL84] T. Imielinski and W. Lipski. Incomplete information in relational databases. *Journal of ACM*, 31(4):761–791, October 1984.

[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.

[Lip79] W. Lipski. On semantic issues connected with incomplete information databases. *ACM Transactions on Database Systems*, 4(3):262–296, September 1979.

[LRV88] C. Lecluse, P. Richard, and F. Velez. O_2, an object-oriented data model. In *Proceedings of ACM SIGMOD Conference*, pages 424–434, 1988.

[Mac86] D. B. MacQueen. Using dependent types to express modular structure. In *Conf. Record Thirteenth Ann. Symp. Principles of Programming Languages*, pages 277–286. ACM, January 1986.

[Mac88a] D. MacQueen. An implementation of Standard ML modules. In *Proc. ACM Conference on LISP and Functional Programming*, pages 212–243, Snowbird, Utah, July 1988.

[Mac88b] D. MacQueen. References and weak polymoprhism. Note in Standard ML of New Jersey Distribution Package, 1988.

[MBCD89] R. Morrison, A.L. Brown, R.C.H. Connor, and A. Dearle. Napier88 reference manual. Technical report, Department of Computational Science, University of St Andrews, 1989.

[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.

[OB89] A. Ohori and P. Buneman. Static type inference for parametric classes. In *Proceedings of ACM OOPSLA Conference*, pages 445–456, New Orleans, Louisiana, October 1989.

[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.

[Oho89] A. Ohori. Semantics of types for database objects. *Theoretical Computer Science*, To appear. (Special issue devoted to 2^{nd} International Conference on Database Theory,) Available as a technical report form University of Pennsylvania, 1989.

[Oho89] A. Ohori. *A Study of Types, Semantics and Languages for Databases and Object-oriented Programming.* PhD thesis, University of Pennsylvania, 1989.

[Oho90] A. Ohori. Extending polymorphism to records and variants. Unpublished manuscript, Preliminary abstract presented at 6^{th} Workshop on Mathematical Foundation of Programming Semantics, 1990.

[Rém89] D. Rémy. Typechecking records and variants in a natural extension of ML. In David MacQueen, editor, *ACM Conference on Principles of Programming Languages*, 1989.

[Rey74] J.C. Reynolds. Towards a theory of type structure. In *Paris Colloq. on Programming*, pages 408–425. Springer-Verlag, 1974.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, March 1965.

[Sch77] J.W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–261, 1977.

[Tof88] M. Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, Department of Computer Science, University of Edinburgh, 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 Annual Symposium on Logic in Computer Science*, pages 37–44, Ithaca, New York, June 1987.

[Wan88] M. Wand. Corrigendum : Complete type inference for simple object. In *Proceedings of the Third Symposium on Logic in Computer Science*, 1988.

[Wan89] M. Wand. Type inference for records concatenation and simple objects. In *Proceedings of 4th IEEE Symposim on Logic in Computer Science*, pages 92–97, 1989.

[Zan84] C. Zaniolo. Database relation with null values. *Journal of Computer and System Sciences*, 28(1):142–166, 1984.