September 1992

Polymorphism and Inference in Database Programming

Peter Buneman
University of Pennsylvania

Atsushi Ohori
Oki Electric

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-92-72.

Polymorphism and Inference in Database Programming

Abstract

The polymorphic type system of ML can be extended in two ways to make it the appropriate 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 involving field selection and generalized relational operators. These extensions may also be used to provide static polymorphic typechecking in object-oriented languages and databases. A problem that arises with object-oriented databases is the apparent need for dynamic typechecking when dealing with queries on heterogeneous collections of objects. An extension of the type system needed for generalized relational operations can also be used for manipulating collections of dynamically typed values in a statically typed language. A prototype language based on these ideas has been implemented. While it lacks a proper treatment of persistent data, it demonstrates that a wide variety of database structures can be cleanly represented in a polymorphic programming language.

Comments

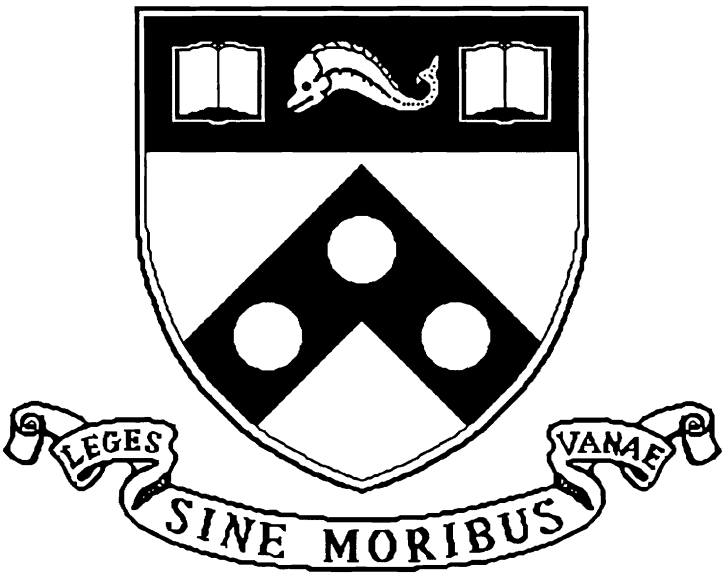
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-92-72.

Polymorphism and Type Inference In Database Programming

MS-CIS-92-72
LOGIC & COMPUTATION 52

Peter Buneman
(University of Pennsylvania)

Atsushi Ohori
(Oki Electric)



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

September 1992

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†
Oki Electric, Kansai Laboratory
Crystal Tower, 1-2-27 Shiromi
Chuo-ku, Osaka 540, JAPAN

Abstract

The polymorphic type system of ML can be extended in two ways to make it the appropriate 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 involving field selection and generalized relational operators.

These extensions may also be used to provide static polymorphic typechecking in object-oriented languages and databases. A problem that arises with object-oriented databases is the apparent need for dynamic typechecking when dealing with queries on heterogeneous collections of objects. An extension of the type system needed for generalized relational operations can also be used for manipulating collections of dynamically typed values in a statically typed language. A prototype language based on these ideas has been implemented. While it lacks a proper treatment of persistent data, 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* — applications of primitive operations 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
```

*Supported by research grants NSF IRI86-10617, ARO DAA6-29-84-k-0061 and ONR NOOO-14-88-K-0634

†This work was performed in part while the second author was supported by a Royal Society Research Fellowship at University of Glasgow, Scotland

is a straightforward process that is routinely carried out by the compiler, not only as a partial check on the correctness of the program, but also as an essential part of the optimization process However, once we add some form of procedural abstraction to the language, typechecking is no longer straightforward. For example, how do we check the type correctness of a program containing the function definition

```
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 with 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, Gir71] and persistence, but its polymorphism does not extend to operations on 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, and programs that violate the intended types can have disastrous consequences. See [AB87] for a survey of various approaches to type-checking in database programming.

The language ML [MTH90] has a *type inference system* which infers, if it exists, a most general polymorphic type for a program [Mil78, DM82]. Because of this, ML enjoys much of the flexibility of untyped (or dynamically typed) languages without sacrificing the advantages of static type checking. Unfortunately the polymorphism in ML is not general enough to express the generic nature of field selection, which occurs in functions such as such as Wealthy and quite generally in database programming. Our goal in this paper is to show that an extension to ML's type system can express the polymorphic nature of the data types and operations that are used in relational and object-oriented databases and is therefore an appropriate basis for a general-purpose database programming language. These ideas are embodied in Machiavelli [OBBT89], an experimental programming language based on 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 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. This function takes a set of records (i.e. a relation) with Name and Salary information and returns the set of all Name values that occur in records with 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 largely 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 types are mentioned in the code, Machiavelli *infers* the type information

$$\mathsf{Wealthy} : \{d :: [\![Name : d', Salary : int]\!]\} \rightarrow \{d'\}.$$

To understand what this means, consider first the type given to the function cons, the function that adds an element to a list, by ML. It is the type expression $t * list(t) \rightarrow list(t)$ in which t is a *type variable.* This represents the polymorphic type $\forall t.t * list(t) \rightarrow list(t)$ where t in $t * list(t) \rightarrow list(t)$ is universally quantified over types. This means that the valid types for cons may be obtained by substituting any type for t. Thus $int * list(int) \rightarrow list(int)$, $string * list(string) \rightarrow list(string)$, and $list(int) * list(list(int)) \rightarrow list(list(int))$ are all valid types for cons. Now in the type for Wealthy above d and d' are also type variables, but unlike the variable t in the previous example we cannot perform arbitrary substitutions of types for these variables. There are two restrictions. The first is indicated by the decoration "$:: [\![Name : d', Salary : int]\!]$" on the type variable d. This allows only certain record types to be substituted for d, i.e. those with a $Salary : int$ field, a $Name : \delta$ field (where δ is obtained by substituting some type for d'), and possibly other fields. This represents polymorphic type of the form $\forall d'.\forall d :: [\![Name : d', Salary : int]\!].\{d\} \rightarrow \{d'\}$ where the type variable d is quantified over only those record types that contain $Name$ and $Salary$ fields of appropriate types. Thus

$$\{[Name : string,\ Salary : int]\} \rightarrow \{string\}$$
$$\{[Name : string,\ Age : int,\ Salary : int]\} \rightarrow \{string\}$$
$$\{[Name : [First : string,\ Last : string],\ Weight : int,\ Salary : int]\} \rightarrow \{[First : string,\ Last : string]\}$$

are allowable instances of the type of wealthy, while

$$\{[Name : string]\} \rightarrow \{string\}$$
$$\{[Name : string,\ Age : int,\ Salary : string]\} \rightarrow \{string\}$$
$$\{int\} \rightarrow \{string\}$$

are not allowable instances, for the substitutions for d that generate them do not match with the constraints imposed by the decoration $[\![Name : d', Salary : int]\!]$. Type variables whose instantiation is controlled by such a decoration are called *kinded* type variables.

The second constraint we place on the type variables d and d' is that they can only be instantiated with *description types.* Some of the essential operations on databases require computable equality, and this is not available on function types and, may be unavailable on 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 that appears outside the scope of a reference type. Equality is always available on references regardless of their associated values. We therefore allow description types to contain function types inside of reference type constructor. ML recognizes a similar constraint on type variables.

In order to display type variables using conventional programming fonts we follow the ML convention of displaying ordinary type variables as 'a, 'b, ...and description type variables as "a, "b etc. Thus the type $\{d :: [\![Name : d', Salary : int]\!]\} \rightarrow \{d'\}$ will be displayed in examples as {"a::[Name : "b, Salary : int]} -> {"b}.

The typing Wealthy: {"a::[Name : "b, Salary : int]} -> {"b} places restrictions on how Wealthy may be used. For example, all of the following

```
Wealthy({[Name = "Joe"], [Name = "Fred"]})
Wealthy({[Name = "Joe", Salary = "nonsense"]})
sum(Wealthy({[Name = "Fred", Salary = 30000],
             [Name = "Joe", Salary = 200000]}))
```

will be rejected by the compiler. In the first application the Salary field is missing; in the second it has the wrong type. In neither case can we find a suitable instantiation for the kinded type variable "a::[Name : "b, Salary : int]. In the third case we can find such an instantiation, but this results in the variable "b being bound to string, so that the result of Wealthy is of type {string} — an inappropriate argument for sum.

There is a close relationship between the polymorphism represented by the kinded type variables the generic nature of 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.) Inheritance is thus 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. "Complex object" or "non-first-normal-form" relations are usually taken as relations whose entries are not restricted to being atomic values, but may themselves be relations. The structures we shall describe are more general in that they can also include variants and cyclic structures. Thus Machiavelli 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 data types.

The attempt to understand the nature of object-oriented databases has centered more on a discussion of features[ABD+89] than on any principled attempt to provide a formal semantics. However, looking at these features, there are some that are not directly captured in a functional language with the relational extensions

we have described above. First, the class structure of object-oriented languages provides a form of abstraction and inheritance that does not immediately fall out of an ML-style type system. Second, object identity is not provided in the relational model (though it is an open issue as to whether it requires more than the addition of a reference type, as in ML.) Third, and perhaps most interesting from the standpoint of object-oriented databases, there is an implicit requirement that *heterogeneous* collections should be representable in the language. We believe that these issues can be satisfactorily resolved in the context of the type system we are advocating. In particular, the heterogeneous collections – which would appear to be inconsistent with static type-checking – can be satisfactorily represented using essentially the same apparatus developed to handle relational data types. This is discussed in section 5.

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. Section 3 also presents the type inference process in some detail for the basic operations required for records, sets and variants. 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 discuss how this type system can be used to capture an important aspect of object oriented databases, the manipulation of heterogeneous collections. Section 6 concludes with a brief discussion of further applications of these ideas to object-oriented languages and databases.

2 Basic Structures for Data Representation

As we have just mentioned, the main goal of this study is to develop a polymorphic type system that serves as a medium in which 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. As we describe each constructor, we shall say under what conditions it constructs a description term. For example, a record whose fields contain functions can be very useful, but such a value cannot be placed directly in a set. This would give rise to a type error.

We start with the basic syntactic forms of Machiavelli for value and function definition, which are exactly those of ML. 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,1) then 1 else n * f(n-1)
```

and there is a 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 used 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 it 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 our Wealthy example 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, $\_.l$ is field selection; $r.l$ selects the l field form the record r. The second, modify(_,l,_), is field modification in which modify(r,l,e) 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)
```

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. This construct enables us to modify a record field that is not a reference. With the polymorphic typing of Machiavelli presented later, it achieves added flexibility in programming with records.

We shall make frequent use of the syntax (e_1, e_2) for pairs. This is simply an abbreviation for the record [first = e_1, second = e_2]. Triples and, generally, n-tuples are similarly constructed.

Variants are used to "tag" values in order to treat them uniformly. For example, the values <Int = 7> and <Real = 3.0> could both be treated as numbers, and the tags used to indicate how the value is to be interpreted (e.g. real or integer.) A program may use these tags in deciding what operations to perform on the tagged values (e.g. real or integer arithmetic.) The syntax for constructing a variant is:

<l=v>

The operation for analyzing a variant is a case expression:

```
case e of
    <l1=x1> => e1,
      ⋮
    <ln=xn> => en,
    else       e0
endcase
```

where each 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 omitted, the argument e must be evaluated to a variant labeled with one of $l_1, \ldots, l_n$. It is a property of the type system that this condition can be statically checked.

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".

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. A variant <l = v> is a description term if v is a description term.

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,*
$\{x\}$	*singleton set constructor,*
union(s_1,s_2)	*set union,*
hom(f,op,z,s)	*homomorphic extension*

The syntax $\{x_1, x_2, \ldots, x_n\}$ is syntactic shorthand for union($\{x_1\}$, union($\{x_2\}$, union(...,$\{x_n\}$)))

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. Its definition is

hom(f,op,z,$\{\}$) = z,
hom(f,op,z,$\{e\}$) =$f(e)$
hom(f,op,z,union(e_1,e_2)) = op(hom(f,op,z,e_1),hom(f,op,z,e_2))

for example, a function to check if there is at least one element satisfying property P in a set can be defined as

```
fun exists P S = hom(P, or, false, S)
```

and a function that finds the largest member of a set of non-negative integers is

```
fun max S = hom( fn x => x, fn(x,y) => if x > y then x else y, 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 and idempotent operation with identity z and f has no side-effects (as is the case in the exists and max examples) then the result of hom will be independent of the order of this evaluation. Now one would also like to use hom on operations that are not idempotent, for example

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

However + is not idempotent, and it is easy to construct programs with ambiguous outcomes if evaluated according to the rules above and a further rule that says union(s, s) = s. For example[1]

```
2 = hom(fn x => x, +, 0, {1, 1}) = hom(fn x => x, +, 0, {1}) = 1
```

Now it is easy enough to remove such ambiguous outcomes by insisting — as we have done in our implementation — that, in the representation of sets, we do not have duplicated elements. This is equivalent to putting a condition on the third line of the definition of hom that the expressions e_1 and e_2 denote disjoint sets. Unfortunately this considerably complicates the operational semantics of the language, and it precludes the possibility of lazy evaluation. For a resolution of this issue, see [BTS91, BTBN91], which disuss the semantic properties of programs with sets and other collection types. In this paper we shall occasionally make use of "incorrect" applications of hom; however we are confident that the adoption of an alternative semantics will not affect typing issues, which are the main concern here.

Various useful functions can be defined using correct applications of hom. A function map(f, S), which applies the function f to each member of S is:

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

For example map(max,{{1,2},{3},{6,5,4}}) evaluates to {2,3,6}.

A selection function is defined by

```
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(even,{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 x1 <- S1,
     x2 <- S2,
      ⋮
     xn <- Sn
where P
```

in which $x_1, x_2, \ldots, x_n$ may occur free in E and P, is provided in the spirit of relational query languages and the list comprehensions of Miranda [Tur85]. This can be implemented as

[1] We are grateful to Val Tannen for this example and for much of the ensuing discussion.

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

in which map, filter and prod_n are the functions we have just described, and (E,P) is a pair of values (implemented in Machiavelli as records). See [Wad90] for a related discussion of syntax for programming with lists.

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 = "Helena", 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 but not v itself. To ensure that the equation $v = e$ has a proper solution, we place the restriction that if e contains a new constructor then the argument of new may not contain x.

2.4 References

We believe – though we shall comment more on this in section 6 – that the notion of "object identity" 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, and $r{:=}v$ changes the value associated with the reference r to v. In a database context, they correspond respectively to creating an object with identity, retrieving the value of an object, and changing the associated value of an object without affecting its 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 John1 = John1 and !John1 = !John2 are true but John1 = John2 is false even though their associated values are the same. Sharing and mutability are captured 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 Dept field. Thus, 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 [Hin69] established a complete type inference algorithm for untyped lambda expressions. Independently, Milner [Mil78] developed a complete type inference algorithm for a functional programming language including polymorphic definition (using let construct.) Damas and Milner [DM82] formulated its type system and showed the completeness of Milner's type inference algorithm. This has been successfully used in the ML family of programming languages [Aug84, MTH90] and also been adopted by other functional languages [Tur85, HPJW+92]. 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 for the extension to the Damas-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. The extension is a departure from that given in our original outline of Machiavelli [OBBT89] in that the notion of kinded types allows us to obtain a "principal type" result for expressions in a core language. This significantly simplifies the presentation of the type inference algorithm.

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". The notation 'a -> 'a is a type representing the set of types that can be obtained by substituting its type variables with some types (such as int, bool or int $\rightarrow$ int). This type can be understood as a representation of a polymorphic type of the form

$\forall t.t \rightarrow t$ in the second-order polymorphic lambda calculus [Rey74, Gir71]. The most important property of the ML type system is that for any type consistent expression it infers a *principal type*. This is a type such that all its instances are types of the expression and conversely any type of the expression is its instance. This means that the type system infers a type that exactly represents the set of all possible types of an expression. In the example of id above, the set of instances of 'a -> 'a is the set of all types of the form $\tau \rightarrow \tau$ and is exactly the set of all possible types of id. By this mechanism, ML achieves *polymorphism* without explicit type abstraction and type application.

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

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

Here "a and "b are also type variables, but in this case they only represent description types. The type for map indicates that it is a function that takes a function of type $\delta_1 \rightarrow \delta_2$ 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(max, {{1,2,3},{7},{5,2}}) is a legitimate application of map. Again, the type ("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(max, {{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 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 x = (#Name x), which is rejected by the compiler unless a complete type is specified for the argument x.) The difficulty is that the conventional notion of types in ML 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 finite sets of record fields. His system, however, does not share with ML the property of principal typing (see [OB88, Wan88] for the analysis of the problem and [JM88, Rem89] for the refinements 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 represented as *kinded* type variables [Oho92], as we have seen in the example of Wealthy in Introduction. For example, the function name above is given the following type

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

As explained in the introduction, the notation all record types containing the field Name : τ where τ is any instance of 'b. Substitutions are restricted to those that respect kind restrictions of type variables. The type above then represents the exact set of all possible types of the function name and is therefore regarded as a principal (kinded) type 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 >> . The top level input is either a value or function binding; it is a name for the

```
-> 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

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 Machiavelli.

3.1 Expressions

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

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

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. The set-valued expression $\{e_1,\ldots,e_n\}$ is shorthand for $\mathsf{union}(\{e_1\},\mathsf{union}(\ldots,\{e_n\})\ldots)$.

The binding val $id = e_1$; e_2 is syntactic sugar for let $id = e_1$ in e_2 end. 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, 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 how operations on records, sets and variants are evaluated, and these can readily be formulated 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(e,x), or hom(e_1,e_2,e_3,x).

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

where $e[(\text{rec } x.e)/x]$ is the expression obtained form e by substituting (rec $x.e$) for all free occurrences of x in e (with necessary renaming of bound variables.) This rule corresponds to "unfolding" of cyclic definitions.

3.2 Types and Description Types

The set of types of Machiavelli, ranged over by τ, is the set of regular trees [Cou83] represented by the following type expressions [2]:

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

t stands for type variables. unit is the trivial type whose only value is (). b and b_d range respectively over the base types and base description types 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, other than v itself, in which the type variable v may occur free, and the entire expression denotes the solution to the equation $v = \tau(v)$, which exists as a 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 : τ_1, second : τ_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 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:

[2] While most of the ideas in this paper related to type-checking can be generalized to work for regular trees, we have not always given this generalization. It is often enough to think of the types in Machiavelli as simply the expressions defined by this syntax

```
(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*, ranged over by δ, is the subset of types represented by the following syntax:

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

d stands for description type variables, i.e. those type variables whose instances are restricted to description types. τ in $\mathsf{ref}(\tau)$ ranges over the syntax of all types 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.

3.3 Type Inference without Records and Variants

As we have already indicated, the Machiavelli type system is based on type inference. A legal program corresponds to an (untyped) expression associated with a type inferred by the type inference system. As such, the definition of this implicit system requires two steps: first we give the *typing rules*, which determine when an untyped expression e is considered to have a type τ and is therefore considered as a well typed expression; second, we develop a type inference algorithm that infers, for any type consistent expression, a principal type. In order to increase readability, we develop the description of the type system, in two stages: in the rest of this subsection and the following subsection, we describe the type system for expressions that do not involve records and variants; then, in subsection 3.4 we extend the system to records and variants by introducing kinding.

The typing rules are given as a set of rules to derive *typing judgments*. Since, in general, an expression e contains free variables and the type of e depends on the types assigned to those variables, a typing judgment is defined relative to a type assignment of free variables. We let $\mathcal{A}$ range over type assignments, which are functions from a finite subset of variables to types. We write $\mathcal{A}(x,\tau)$ for the function $\mathcal{A}'$ such that $domain(\mathcal{A}') = domain(\mathcal{A}) \cup \{x\}$, $\mathcal{A}'(x) = \tau$ and $\mathcal{A}'(y) = \mathcal{A}(y)$ for $y \neq x$. A typing judgment is a formula of the form:

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

expressing the fact that expression e has type τ under type assignment $\mathcal{A}$. The typing rules for those operations in Machiavelli that do not involve records are shown in Figure 2. Note that in some of them such as (UNION), types are restricted to description types, which is indicated by the use of δ instead of τ.

In (LET), the notation $e_1[e_2/x]$ denotes the expression obtained from e_1 by substituting e_2 for all free occurrences of x. This rule for polymorphic let differs form that of Damas-Milner system [DM82] in that it does not use generic types (a type expression of the form $\forall t.\ \tau$) but instead it uses syntactic substitution of expressions. It is shown in [Oho89a] that this proof system is equivalent to that of Damas-Milner. The

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

(UNIT) $\mathcal{A} \rhd () : \mathsf{unit}$

(VAR) $\mathcal{A} \rhd x : \tau \quad \text{if } \mathcal{A}(x) = \tau$

(ABS) $$\frac{\mathcal{A}(x, \tau_1) \rhd e : \tau_2}{\mathcal{A} \rhd \mathsf{fn}\ x \Rightarrow e : \tau_1 \to \tau_2}$$

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

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

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

(EQ) $$\frac{\mathcal{A} \rhd e_1 : \delta \qquad \mathcal{A} \rhd e_2 : \delta}{\mathcal{A} \rhd \mathsf{eq}(e_1, e_2) : \mathsf{bool}}$$

(SINGLETON) $$\frac{\mathcal{A} \rhd e : \delta}{\mathcal{A} \rhd \{e\} : \{\delta\}}$$

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

(HOM) $$\frac{\mathcal{A} \rhd e_1 : \delta \to \tau_1 \quad \mathcal{A} \rhd e_2 : (\tau_1 * \tau_2) \to \tau_2 \quad \mathcal{A} \rhd e_3 : \tau_2 \quad \mathcal{A} \rhd e_4 : \{\delta\}}{\mathcal{A} \rhd \mathsf{hom}(e_1, e_2, e_3, e_4) : \tau_2}$$

(NEW) $$\frac{\mathcal{A} \rhd e : \tau}{\mathcal{A} \rhd \mathsf{new}(e) : \mathsf{ref}(\tau)}$$

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

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

(REC) $$\frac{\mathcal{A}(v, \delta) \rhd e(v) : \delta}{\mathcal{A} \rhd (\mathsf{rec}\ v.\ e(v)) : \delta}$$

Figure 2: Typing Rules for Expressions Without Records and Variants

advantage of our treatment of let is that it yields simpler proofs of various properties of the type system and that the type system can be extended to records, variants and database operations. While it is still possible to extend Damas-Milner generic types to records and variants using kinded type abstraction [Oho92], we do not know how to extend them to the conditional typing that we shall require for database operations. However, a naive implementation of a type inference algorithm based on this typing rule would require recursive unfolding of **let** definitions. This unfolding process always terminates but would decrease efficiency and prohibit the possibility of incremental type-checking. This problem is overcome by adding an extra parameter to a type inference algorithm to maintain principal types for let-bound variables. We will comment on this when we describe the type inference algorithm.

The proof system of Figure 2 determines which expressions are type correct Machiavelli programs (not involving operations on records and variants.) 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.

Following [Hin69, Mil78], we solve this problem by developing an algorithm that always infers a principal type for any type consistent expressions. A *substitution* S is a function from type variables to types. A substitution may be extended to type expressions, and we identify a substitution and its extension, i.e. we shall write $S(\tau)$ for the expression obtained by replacing each type variable t in τ with $S(t)$. A typing $\mathcal{A}_1 \rhd e : \tau_1$ is *more general* than $\mathcal{A}_2 \rhd e : \tau_2$ if $domain(\mathcal{A}_1) \subseteq domain(\mathcal{A}_2)$ and there is some substitution S such that $\tau_2 = S(\tau_1)$ and $\mathcal{A}_2(x) = S(\mathcal{A}_1(x))$ for all $x \in domain(\mathcal{A}_1)$. A typing $\mathcal{A} \rhd e : \tau$ is *principal* if it is more general than any other derivable typing of e.

Figure 3 shows an algorithm to compute a principal typing for any untyped expression of Machiavelli that does not contain records, variants and database operations. The algorithm consists of a set of functions, one for each typing rule, together with the main function *Typing*. Based on the typing rule (RULE), P_{RULE} synthesizes a principal typing for an expression e from those of its subexpressions. It generates the equations that make the typings of the subexpressions conform to the premises of the rule, solves the equations and generates the typing corresponding to the conclusion of the rule. *Unify* used in these functions is a unification algorithm. $allpairs(\{\mathcal{A}_1, \ldots, \mathcal{A}_n\})$ denotes the set of pairs $\{(\mathcal{A}_i(x), \mathcal{A}_j(x)) | x \in domain(\mathcal{A}_i) \cap domain(\mathcal{A}_j), i \neq j\}$. The notation $F{\restriction}^X$ denotes the restriction of the function F to the set $X \subseteq domain(F)$.

For example, consider the function P_{APP}, which takes principal typings of e_1 and e_2, and synthesizes a principal typing of $e_1(e_2)$. It first generates the equations that require the common variables of e_1 and and e_2 to have the same type assignment, together with the equation that makes the type of e_2 to be the domain type of the type of e_1. They are respectively the set of equations $allpairs(\{\mathcal{A}_1, \mathcal{A}_2\})$ and the equation $(\tau_1, \tau_2 \rightarrow t)$. It then solves these equations by *Unify* which always finds a most general solution to the equations (if it exists) in the form of a substitution S. Finally, it returns the type assignment $S(\mathcal{A}_1 \cup \mathcal{A}_2)$ and a type $S(t)$, corresponding to the conclusion of the rule APP.

The main function *Typing* is presented in the style of [Mit90]. It analyzes the structure of the given expression, recursively calls itself on its subexpressions to get their principal typings and then calls an appropriate function P that corresponds to the outermost constructor of the expression. The extra parameter L to *Typing* is an environment that records the principal typings of let-bound variables. By maintaining this environment, the algorithm avoids repeated computation of a principal type of e_1 in inferring a typing of expressions of the form **let** x=e_1 **in** e_2 **end**, and it also enables incremental compilation. Renaming type variables in the case of $x \in domain(L)$ effectively achieves the same effect of computing the principal typing

of e_1 for each occurrence of x in e_2.

As an example of type inference, let us use the algorithm to compute a principal typing of the function insert and of its application:

```
val insert = fn x => fn S => union({x}, S);
insert 2 {};
```

Figure 4 shows the sequence of the function calls and their results during the computation. Line 1 is the top level call of the algorithm on fn x => fn S => union({x}, S). Line 3 is the first recursive call on its only subexpression, whose result is shown on line 15. Line 9 and 12 contain a call of *Typing* on a variable which immediately returns a principal typing. In $P_{\text{SINGLETON}}$ on line 10 and 11, type variable t_1 is unified with a fresh description type variable d_1. In line 13 and 14, P_{UNION} unifies type variable t_2 with type $\{d_1\}$ and takes the union of type assignments. Line 17 shows a principal typing of insert. Line 18 – 35 shows an inference process for insert 2 {}, which is a shorthand for let insert = fn x => fn S => union({x}, S) in insert 2 {} end.

It requires some work to show that the algorithm we have described has the desired properties. We have also glossed over some important details such as the treatment of description type variables, recursive types and references. Before dealing with these issues let us first show how the typing rules and the inference system may be extended to handle records and variants.

3.4 Kinded Type Inference for Records and Variants

To extend the type system to records and variants, we need to introduce kind constraints on type variables. The set of kinds in Machiavelli is given by the syntax:

$$K ::= \mathsf{U} \mid [\![l{:}\tau,\ldots,l{:}\tau]\!] \mid \langle\!\langle l{:}\tau,\ldots,l{:}\tau \rangle\!\rangle$$

The idea is that U denotes the set of all types, $[\![l_1{:}\tau_1,\ldots,l_n{:}\tau_n]\!]$ denotes the set of record types containing the set of all fields $l_1 : \tau_1, \ldots, l_n : \tau_n$, and $\langle\!\langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n \rangle\!\rangle$ denotes the set of variant types containing the set of all fields $l_1 : \tau_1, \ldots, l_n : \tau_n$.

In the extended type system, type variables must be kinded by a *kind assignment* $\mathcal{K}$, which is a mapping from type variables to kinds. We write $\{t_1 :: k_1, \ldots, t_n :: k_n\}$ for a kind assignment $\mathcal{K}$ that maps t_i to k_i $(1 \leq i \leq n)$. A type τ has a kind k under a kind assignment $\mathcal{K}$, denoted by $\mathcal{K} \vdash \tau :: k$, if it satisfies the conditions shown in Figure 5. For example, the following is a legal kinding:

$$\{t_1 :: \mathsf{U}, t_2 :: [\![\mathsf{Name} : t_1, \mathsf{Age} : \mathsf{int}]\!]\} \vdash t_2 :: [\![\mathsf{Name} : t_1]\!]$$

A typing judgment is now refined to incorporate kind constraints on type variables:

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

Typing judgments of the form $\mathcal{A} \rhd e : \tau$ described in the previous subsection should now be taken as judgments of the form $\mathcal{K}_0, \mathcal{A} \rhd e : \tau$ where $\mathcal{K}_0$ is the kind assignment mapping all the type variables appearing in $\mathcal{A}, \tau$ to the universal kind U. The typing rules for records and variants in the extended type system are given in Figure 6. The rules for other constructors are the same as before except that they should be reinterpreted by adding the universal kinding stated above. Note that the kinding constraints in the rules

$P_{\text{APP}}((\mathcal{A}_1, \tau_1), (\mathcal{A}_2, \tau_2)) =$
  let $S = \mathit{Unify}(\mathit{allpairs}(\{\mathcal{A}_1, \mathcal{A}_2\}) \cup \{(\tau_1, \tau_2 \rightarrow t)\})$ (t fresh)
  in $(S(\mathcal{A}_1) \cup S(\mathcal{A}_2), S(t))$
  end

$P_{\text{ABS}}((\mathcal{A}, \tau), x) =$
  if $x \in \mathsf{domain}(\mathcal{A})$ *then* $(\mathcal{A}\!\uparrow^{\mathit{domain}(\mathcal{A})\setminus\{x\}}, \mathcal{A}(x) \rightarrow \tau)$
  else $(\mathcal{A}_1, t \rightarrow \tau)$ (t fresh)

$P_{\text{LET}}((\mathcal{A}_1, \tau_1), (\mathcal{A}_2, \tau_2)) =$
  let $S = \mathit{Unify}(\mathit{allpairs}(\{\mathcal{A}_1, \mathcal{A}_2\}))$
  in $(S(\mathcal{A}_1 \cup \mathcal{A}_2), S(\tau_2))$
  end

$P_{\text{SINGLETON}}(\mathcal{A}, \tau) = \mathit{let}\ S = \mathit{Unify}(\{(\tau, d)\})\ \mathit{in}\ (S(\mathcal{A}), \{S(d)\})\ \mathit{end}$ (d fresh)

$P_{\text{UNION}}((\mathcal{A}_1, \tau_1), (\mathcal{A}_2, \tau_2)) =$
  let $S = \mathit{Unify}(\mathit{allpairs}(\{\mathcal{A}_1, \mathcal{A}_2\}) \cup \{(\tau_1, \tau_2), (\tau_1, \{t\})\})$ (t fresh)
  in $(S(\mathcal{A}_1 \cup \mathcal{A}_2), S(\{t\}))$
  end
  ⋮

$\mathit{Typing}(e, L) =$
  case e of:

c_τ	$\Longrightarrow$	$(\emptyset, \tau)$
x	$\Longrightarrow$	if $x \in \mathit{domain}(L)$ then $L(x)$ with all type variables renamed else $(\{x : t\}, t)$ (t fresh)
fn x => e	$\Longrightarrow$	$P_{\text{ABS}}(\mathit{Typing}(e, L), x)$
$e_1(e_2)$	$\Longrightarrow$	$P_{\text{APP}}(\mathit{Typing}(e_1, L), \mathit{Typing}(e_2, L))$
let $x = e_1$ in e_2	$\Longrightarrow$	let $(\mathcal{A}_1, \tau_1) = \mathit{Typing}(e_1, L)$ $L' = L(x, (\mathcal{A}_1, \tau_1))$ in $P_{\text{LET}}((\mathcal{A}_1, \tau_1), \mathit{Typing}(e_2, L'))$
$\{e\}$	$\Longrightarrow$	$P_{\text{SINGLETON}}(\mathit{Typing}(e, L))$
union(e_1,e_2)	$\Longrightarrow$	$P_{\text{UNION}}(\mathit{Typing}(e_1, L), \mathit{Typing}(e_2, L))$
⋮		

  endcase

Figure 3: Type Inference Algorithm without Records, Variants

$$
\begin{array}{ll}
1 & \mathit{Typing}(\mathsf{fn\ x\ =>\ fn\ S\ =>\ union(\{x\},S)},\emptyset) \\
2 & \quad = P_{\text{ABS}}(\mathit{Typing}(\mathsf{fn\ S\ =>\ union(\{x\},S)},\emptyset),\mathsf{x}) \\
3 & \qquad \rangle \mathit{Typing}(\mathsf{fn\ S\ =>\ union(\{x\},S)},\emptyset) \\
4 & \qquad \rangle \quad = P_{\text{ABS}}(\mathit{Typing}(\mathsf{union(\{x\},S)},\emptyset),S) \\
5 & \qquad \rangle \qquad \rangle \mathit{Typing}(\mathsf{union(\{x\},S)},\emptyset) \\
6 & \qquad \rangle \qquad \rangle \quad = P_{\text{UNION}}(\mathit{Typing}(\{\mathsf{x}\},\emptyset),\mathit{Typing}(\mathsf{S},\emptyset)) \\
7 & \qquad \rangle \qquad \rangle \qquad \rangle \mathit{Typing}(\{\mathsf{x}\},\emptyset) \\
8 & \qquad \rangle \qquad \rangle \qquad \rangle \quad = P_{\text{SINGLETON}}(\mathit{Typing}(\mathsf{x},\emptyset)) \\
9 & \qquad \rangle \qquad \rangle \qquad \rangle \qquad \rangle \mathit{Typing}(\mathsf{x},\emptyset) = (\{\mathsf{x}\colon t_1\},t_1) \\
10 & \qquad \rangle \qquad \rangle \qquad \rangle \quad = P_{\text{SINGLETON}}((\{\mathsf{x}\colon t_1\},t_1)) \\
11 & \qquad \rangle \qquad \rangle \qquad \rangle \quad = (\{\mathsf{x}\colon d_1\},\{d_1\}) \\
12 & \qquad \rangle \qquad \rangle \qquad \rangle \mathit{Typing}(\mathsf{S},\emptyset) = (\{\mathsf{S}\colon t_2\},t_2) \\
13 & \qquad \rangle \qquad \rangle \quad = P_{\text{UNION}}((\{\mathsf{x}\colon d_1\},\{d_1\}),(\{\mathsf{S}\colon t_2\},t_2)) \\
14 & \qquad \rangle \qquad \rangle \quad = (\{\mathsf{x} : d_1,\mathsf{S} : \{d_1\}\},\{d_1\}) \\
15 & \qquad \rangle \quad = (\{\mathsf{x} : d_1\},\{d_1\} \to \{d_1\}) \\
16 & \quad = P_{\text{ABS}}((\{\mathsf{x} : d_1\},\{d_1\} \to \{d_1\}),\mathsf{x}) \\
17 & \quad = (\emptyset, d_1 \to \{d_1\} \to \{d_1\}) \\
 & \\
18 & \mathit{Typing}(\mathsf{let\ insert\ =\ fn\ x\ =>\ fn\ S\ =>\ union(\{x\},S)\ in\ insert\ 2\ \{\}\ end},\emptyset) \\
19 & \quad = P_{\text{LET}}((\emptyset, d_1 \to \{d_1\} \to \{d_1\})),\mathit{Typing}(\mathsf{insert\ 2\ \{\}},\{(\mathsf{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\}) \\
20 & \qquad \rangle \mathit{Typing}(\mathsf{insert\ 2\ \{\}},\{(\mathsf{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\}) \\
21 & \qquad \rangle \quad = P_{\text{APP}}(\mathit{Typing}(\mathsf{insert\ 2},\{(\mathsf{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\}), \\
 & \qquad\qquad\qquad \mathit{Typing}(\{\},\{(\mathsf{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\})) \\
22 & \qquad \rangle \qquad \rangle \mathit{Typing}(\mathsf{insert\ 2},\{(\mathsf{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\}) \\
23 & \qquad \rangle \qquad \rangle \quad = P_{\text{APP}}(\mathit{Typing}(\mathsf{insert},\{(\mathsf{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\}), \\
 & \qquad\qquad\qquad\qquad \mathit{Typing}(2,\{(\mathsf{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\})) \\
24 & \qquad \rangle \qquad \rangle \qquad \rangle \mathit{Typing}(\mathsf{insert},\{(\mathsf{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\}) \\
25 & \qquad \rangle \qquad \rangle \qquad \rangle \quad = (\emptyset, d_2 \to \{d_2\} \to \{d_2\}) \\
26 & \qquad \rangle \qquad \rangle \qquad \rangle \mathit{Typing}(2,\{(\mathsf{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\}) \\
27 & \qquad \rangle \qquad \rangle \qquad \rangle \quad = (\emptyset, int) \\
28 & \qquad \rangle \qquad \rangle \quad = P_{\text{APP}}((\emptyset, d_2 \to \{d_2\} \to \{d_2\}),(\emptyset, int)) \\
29 & \qquad \rangle \qquad \rangle \quad = (\emptyset,\{int\} \to \{int\}) \\
30 & \qquad \rangle \qquad \rangle \mathit{Typing}(\{\},\{(\mathsf{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\}) \\
31 & \qquad \rangle \qquad \rangle \quad = (\emptyset,\{d_3\}) \\
32 & \qquad \rangle \quad = P_{\text{APP}}((\emptyset,\{int\} \to \{int\}),(\emptyset,\{d_3\})) \\
33 & \qquad \rangle \quad = (\emptyset,\{int\}) \\
34 & \quad = P_{\text{let}}((\emptyset, d_1 \to \{d_1\} \to \{d_1\}),(\emptyset,\{int\})) \\
35 & \quad = (\emptyset,\{int\})
\end{array}
$$

Figure 4: Computing a Principal Typing

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

$$\mathcal{K} \vdash t :: [\![l_1{:}\tau_1,\ldots,l_n{:}\tau_n]\!] \quad \text{if } t \in \mathit{domain}(\mathcal{K}), \mathcal{K}(t) = [\![l_1{:}\tau_1,\ldots,l_n{:}\tau_n,\ldots]\!]$$

$$\mathcal{K} \vdash [l_1{:}\tau_1,\ldots,l_n{:}\tau_n,\ldots] :: [\![l_1{:}\tau_1,\ldots,l_n{:}\tau_n]\!]$$

$$\mathcal{K} \vdash t :: \langle\!\langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n \rangle\!\rangle \quad \text{if } t \in \mathit{domain}(\mathcal{K}), \mathcal{K}(t) = \langle\!\langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n,\ldots \rangle\!\rangle$$

$$\mathcal{K} \vdash \langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n,\ldots \rangle :: \langle\!\langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n \rangle\!\rangle$$

Figure 5: Kinding Rules

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

(DOT) $$\frac{\mathcal{K},\mathcal{A} \rhd e : \tau_1 \qquad \mathcal{K} \vdash \tau_1 :: [\![l : \tau_2]\!]}{\mathcal{K},\mathcal{A} \rhd e.l : \tau_2}$$

(MODIFY) $$\frac{\mathcal{K},\mathcal{A} \rhd e_1 : \tau_1 \qquad \mathcal{K},\mathcal{A} \rhd e_2 : \tau_2 \qquad \mathcal{K} \vdash \tau_1 :: [\![l : \tau_2]\!]}{\mathcal{K},\mathcal{A} \rhd \mathsf{modify}(e_1,l,e_2) : \tau_1}$$

(VARIANT) $$\frac{\mathcal{K},\mathcal{A} \rhd e : \tau_1 \qquad \mathcal{K} \vdash \tau_2 :: \langle\!\langle l{:}\tau_1 \rangle\!\rangle}{\mathcal{K},\mathcal{A} \rhd \langle l{=}e \rangle : \tau_2}$$

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

(CASE') $$\frac{\mathcal{K},\mathcal{A} \rhd e : \tau_0 \ \ \mathcal{K},\mathcal{A}(x_i,\tau_i) \rhd e_i : \tau \ (1 \le i \le n) \ \ \mathcal{K},\mathcal{A} \rhd e_0 : \tau \ \ \mathcal{K} \vdash \tau_0 :: \langle\!\langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n \rangle\!\rangle}{\mathcal{K},\mathcal{A} \rhd \mathsf{case}\ e\ \mathsf{of}\ \langle l_1{=}x_1 \rangle \Rightarrow e_1,\ldots,\ \langle l_n{=}x_n \rangle \Rightarrow e_n\ \mathsf{else} \Rightarrow e_0\ \mathsf{endcase} : \tau}$$

Figure 6: Typing Rules for Records and Variants

(DOT) and (VARIANT) exactly capture the conditions for the expressions to have a typing. The following is an example of legal typing:

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

which says that the function fn x => x.Name can be applied to any record type t_2 which contains the field Name:t_1 and returns a value of type t_1.

To refine the type inference algorithm, we need to refine an unification algorithm to *kinded unification.* The strategy is to add a kind assignment to each component in unification and to check the condition that unification respects the constraints specified by kind assignments. A *kinded substitution* is a pair $(\mathcal{K}, S)$ consisting of a kind assignment $\mathcal{K}$ and a substitution S. Intuitively, the kind assignment $\mathcal{K}$ is the kind constraints that must be satisfied by the results of applying the substitution S. We write $[t_1 \mapsto \tau_1, \ldots, t_n \mapsto \tau_n]$ for the substitution which maps x_i to τ_i $(1 \le i \le n)$. We say that a kinded substitution $(\mathcal{K}_1, S)$ *respects* a kind assignment $\mathcal{K}_2$ if, for all $t \in \mathit{domain}(\mathcal{K}_2)$, $\mathcal{K}_1 \vdash S(t) :: S(\mathcal{K}_2(t))$ is a legal kinding. For example, a kind substitution

$$(\{t_1 :: \mathsf{U}\}, [t_2 \mapsto [\mathsf{Name} : t_1, \mathsf{Age} : \mathsf{int}]])$$

respects the kind constraints $\{t_1 :: \mathsf{U}, t_2 :: [\![\mathsf{Name} : t_1]\!]\}$ and can be applied to type t_2 under this constraint. A kinded substitution $(\mathcal{K}_1, S_1)$ is *more general* than $(\mathcal{K}_2, S_2)$ if $S_2 = S_3 \circ S_1$ for some S_3 such that $(\mathcal{K}_2, S_3)$ respects $\mathcal{K}_1$, where $S \circ S'$ is the composition of substitutions S, S' defined as $S \circ S'(t) = S(S'(t))$. A ***kinded set of equations*** is a pair consisting of a kind assignment and a set of pairs of types. A kinded substitution $(\mathcal{K}_1, S)$ is a ***unifier*** of a kinded set of equations $(\mathcal{K}_2, E)$ if it respects $\mathcal{K}_2$ and $S(\tau_1) = S(\tau_2)$ for all $(\tau_1, \tau_2) \in E$. We can then obtain the following result, a refinement of Robinson's [Rob65] unification algorithm.

Theorem 1 *There is an algorithm Unify which, given any kinded set of equations, computes a most general unifier if one exists and reports failure otherwise.*

We provide here a description of the algorithm; a sketch of its correctness proof is to be found in [Oho92]. The algorithm *Unify* is presented in the style of [GS89] by a set of transformation rules on triples $(\mathcal{K}, E, S)$ consisting of a kind assignment $\mathcal{K}$, a set E of type equations and a set S of "solved" type equations of the form (t, τ) such that $t \notin FTV(\tau)$. Let $(\mathcal{K}, E)$ be a given kinded set of equations. The algorithm *Unify* first transforms $(\mathcal{K}, E, \emptyset)$ to $(\mathcal{K}', E', S')$ until no more rules can apply. It then returns $(\mathcal{K}', S')$ if E' is empty; otherwise it reports failure.

Let F range over functions from a finite set of labels to types. We write $[F]$ and $[\![F]\!]$ respectively to denote the record type identified by F and the record kind identified by F. Figure 7 gives the set of transformation rules for record types and function types. The rules for variant types are obtained from those of record types by replacing record type constructor $[F]$, record kind constructor $[\![F]\!]$ with variant type constructor $\langle F \rangle$, and variant kind constructor $\langle\!\langle F \rangle\!\rangle$, respectively. Rules I, II, V and VI are same as those in ordinary unification. Rule I eliminates an equation and is always valid. Rule II is the case for variable elimination; if occur-check (the condition that t does not appear in τ) succeeds then it generates one point substitution $[t \mapsto \tau]$, applies it to all the type expressions involved and then moves the equation (t, τ) to the solved position. Rules V and VI decompose an equation of complex types into a set of equations of the corresponding subcomponents. Rules III and IV are cases for variable elimination similar to rule II except that the variables have non trivial kind constraint. In addition to eliminating a type variable as in rule II, these rules check the consistency of kind constraints and, if they are consistent, generates a set of new equations equivalent to the kind constraints.

Using this refined unification algorithm, we can now extend the type inference system. First, we refine the notion of principal typings. A typing $\mathcal{K}_1, \mathcal{A}_1 \rhd e : \tau_1$ is *more general than* $\mathcal{K}_2, \mathcal{A}_2 \rhd e : \tau_2$ if $domain(\mathcal{A}_1) \subseteq domain(\mathcal{A}_2)$, and there is a substitution S such that the kinded substitution $(\mathcal{K}_2, S)$ respects $\mathcal{K}_1$, $\mathcal{A}_2(t) = S(\mathcal{A}_1(t))$ for all $t \in domain(\mathcal{A}_1)$, and $\tau_2 = S(\tau_1)$. A typing $\mathcal{K}, \mathcal{A} \rhd e : \tau$ is *principal* if it is more general than all the derivable typings for e. The type inference algorithm is extended by adding the new functions to compose a principal type for record and variant operations and to extend the main algorithm by adding the cases for records and variants. Figure 8 shows the new composition functions corresponding to the typing rules for records and variants. The functions we have defined in Figure 3 remain unchanged except that they take kinded typings of the form $(\mathcal{K}, \mathcal{A}, \tau)$ and the appropriate kind assignments must be added as component of the the parameter of the unification algorithm and of its result. Figure 9 shows the necessary changes to the main algorithm.

Figure 10 shows the type inference process for the function fn x => (x.Name, x.Sal > 10000), a function that is used in the implementation of Wealthy, which was described earlier. In this example, the pairing function $(\_,\_)$ and the product type are respectively shorthand for a standard binary record constructor and binary record type.

I $(\mathcal{K}, E \cup \{(\tau, \tau)\}, S) \Rightarrow (\mathcal{K}, E, S)$

II $(\mathcal{K} \cup \{t \mapsto U\}, E \cup \{(t, \tau)\}, S) \Rightarrow ([t \mapsto \tau](\mathcal{K}), [t \mapsto \tau](E), \{(t, \tau)\} \cup [t \mapsto \tau](S))$ if t does not appear in τ

III $(\mathcal{K} \cup \{t_1 \mapsto [\![F_1]\!], t_2 \mapsto [\![F_2]\!]\}, E \cup \{(t_1, t_2)\}, S) \Rightarrow$
$([t_1 \mapsto t_2](\mathcal{K} \cup \{t_2 \mapsto [\![F]\!]\}),$
$[t_1 \mapsto t_2](E \cup \{(F_1(l), F_2(l)) | l \in domain(F_1) \cap domain(F_2)\}),$
$\{(t_1, t_2)\} \cup [t_1 \mapsto t_2](S))$
where $F = \{(l, \tau_l) | l \in domain(F_1) \cup domain(F_2), \tau_l = F_1(l)$ if $l \in domain(F_1)$ otherwise $\tau_l = F_2(l)\}$
if t_1 not appears in F_2 and t_2 not appears in F_1.

IV $(\mathcal{K} \cup \{t_1 \mapsto [\![F_1]\!]\}, E \cup \{(t_1, [F_2])\}, S) \Rightarrow$
$([t_1 \mapsto [F_2]](\mathcal{K}),$
$[t_1 \mapsto [F_2]](E \cup \{(F_1(l), F_2(l)) | l \in domain(F_1) \cap domain(F_2)\}),$
$\{(t_1, [F_2])\} \cup [t_1 \mapsto [F_2]](S))$
if $domain(F_1) \subseteq domain(F_2)$ and $t \notin FTV([F_2])$

V $(\mathcal{K}, E \cup \{(\tau_1^1 \to \tau_1^2, \tau_2^1 \to \tau_2^2)\}, S) \Rightarrow (\mathcal{K}, E \cup \{(\tau_1^1, \tau_2^1), (\tau_1^2, \tau_2^2)\}, S)$

VI $(\mathcal{K}, E \cup \{[F_1], [F_2]\}, S) \Rightarrow (\mathcal{K}, E \cup \{(F_1(l), F_2(l)) | l \in domain(F_1)\}, S)$
if $domain(F_1) = domain(F_2)$

Figure 7: Some of the Transformation Rules for Kinded Unification

$P_{\text{RECORD}}([l_1 = (\mathcal{K}_1, \mathcal{A}_1, \tau_1), \ldots, l_n = (\mathcal{K}_n, \mathcal{A}_n, \tau_n)]) =$
let $(\mathcal{K}, S) = \mathit{Unify}(\mathcal{K}_1 \cup \cdots \cup \mathcal{K}_n, \mathit{allpairs}(\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}))$ (t fresh)
in $(\mathcal{K}, S(\mathcal{A}_1) \cup \cdots \cup S(\mathcal{A}_n), S([l_1 : \tau_1, \ldots, l_n : \tau_n]))$
end

$P_{\text{DOT}}((\mathcal{K}, \mathcal{A}, \tau), l) =$
let $(\mathcal{K}', S) = \mathit{Unify}(\mathcal{K} \cup \{t_1 :: U, t_2 :: [\![l : t_1]\!]\}, \{(t_2, \tau)\})$ (t_1, t_2 fresh)
in $(\mathcal{K}', S(\mathcal{A}), S(t_1))$
end

$P_{\text{MODIFY}}((\mathcal{K}_1, \mathcal{A}_1, \tau_1), (\mathcal{K}_2, \mathcal{A}_2, \tau_2), l) =$
let $(\mathcal{K}, S) = \mathit{Unify}(\mathcal{K}_1 \cup \mathcal{K}_2 \cup \{t_1 :: U, t_2 :: [\![l : t_1]\!]\}, \mathit{allpairs}(\{\mathcal{A}_1, \mathcal{A}_2\}) \cup \{(t_2, \tau_1), (t_1, \tau_2)\})$ (t_1, t_2 fresh)
in $(\mathcal{K}, S(\mathcal{A}), S(t_2))$
end

$P_{\text{VARIANT}}((\mathcal{K}, \mathcal{A}, \tau), l) =$
let $(\mathcal{K}', S) = \mathit{Unify}(\mathcal{K} \cup \{t_1 :: U, t_2 :: \langle\!\langle l : t_1 \rangle\!\rangle\}, \{(t_1, \tau)\})$ (t_1, t_2 fresh)
in $(\mathcal{K}', S(\mathcal{A}), S(t_2))$
end

$P_{\text{CASE1}}((\mathcal{K}_0, \mathcal{A}_0, \tau_0), [l_1 = (\mathcal{K}_1, \mathcal{A}_1, \tau_1), \ldots, l_n = (\mathcal{K}_n, \mathcal{A}_n, \tau_n)]) =$
let $(\mathcal{K}, S) =$
$\mathit{Unify}(\mathcal{K}_0 \cup \cdots \cup \mathcal{K}_n \cup \{t :: U, t_1 :: U, \ldots, t_n :: U\},$
$\mathit{allpairs}(\{\mathcal{A}_0, \ldots, \mathcal{A}_n\}) \cup \{(\tau_i, t_i \rightarrow t) | 1 \leq i \leq n\} \cup \{(\tau_0, <l_1 : t_1, \ldots, l_n : t_n>)\})$ ($t, t_1, \ldots, t_n$ fresh)
in $(\mathcal{K}, S(\mathcal{A}_1) \cup \cdots \cup S(\mathcal{A}_n), S(t))$
end

$P_{\text{CASE2}}((\mathcal{K}_0, \mathcal{A}_0, \tau_0), [l_1 = (\mathcal{K}_1, \mathcal{A}_1, \tau_1), \ldots, l_n = (\mathcal{K}_n, \mathcal{A}_n, \tau_n)], (\mathcal{K}_{n+1}, \mathcal{A}_{n+1}, \tau_{n+1})) =$
let $(\mathcal{K}, S) =$
$\mathit{Unify}(\mathcal{K}_0 \cup \cdots \cup \mathcal{K}_{n+1} \cup \{t :: U, t_1 :: U, \ldots, t_n :: U, t_0 :: \langle\!\langle l_1 : t_1, \ldots, l_n : t_n \rangle\!\rangle\},$
$\mathit{allpairs}(\{\mathcal{A}_0, \ldots, \mathcal{A}_n\}) \cup \{(\tau_i, t_i \rightarrow t) | 1 \leq i \leq n\} \cup \{(\tau_0, t_0), (\tau_{n+1}, t)\})$ ($t, t_0, t_1, \ldots, t_n$ fresh)
in $(\mathcal{K}, S(\mathcal{A}_1) \cup \cdots \cup S(\mathcal{A}_n), S(t))$
end

Figure 8: New Functions to Synthesize Principal Typings

$Typing(e, L) =$
case e of

c_τ	$\Longrightarrow$	$(\emptyset, \emptyset, \tau)$
x	$\Longrightarrow$	if $x \in domain(L)$ then $L(x)$ with all type variables renamed else $(\{t :: U\}, \{x : t\}, t)$ (t fresh)
$\vdots$		
$[l_1{=}e_1, \ldots, l_n{=}e_n]$	$\Longrightarrow$	$P_{\text{RECORD}}([l_1 = Typing(e_1, L), \ldots, l_n = Typing(e_n, L)])$
$e.l$	$\Longrightarrow$	$P_{\text{DOT}}(Typing(e, L).\, l)$
modify(e_1,l,e_2)	$\Longrightarrow$	$P_{\text{MODIFY}}(Typing(e_1, L), Typing(e_2, L), l)$
<l=e>	$\Longrightarrow$	$P_{\text{VARIANT}}(Typing(e, L), l)$
case e of <l_1=x_1> => e_1,..., <l_n=x_n> => e_n endcase	$\Longrightarrow$	$P_{\text{CASE1}}(Typing(e, L),$ $[l_1 = P_{\text{ABS}}(Typing(e_1, L), x_1), \ldots, l_n = P_{\text{ABS}}(Typing(e_n, L), x_n)])$
case e of <l_1=x_1> => e_1,..., <l_n=x_n> => e_n else e_0 endcase	$\Longrightarrow$	$P_{\text{CASE2}}(Typing(e, L),$ $[l_1 = P_{\text{ABS}}(Typing(e_1, L), x_1), \ldots, l_n = P_{\text{ABS}}(Typing(e_n, L), x_n)],$ $Typing(e_0, L))$

endcase

Figure 9: The Main Algorithm for Type Inference with Records and Variants

```
1   Typing(fn x => (x.Name, x.Sal > 10000), ∅)
2     = P_ABS(Typing((x.Name, x.Sal> 10000), ∅), x)
3       )Typing((x.Name, x.Sal > 10000), ∅)
4       )   = P_RECORD((Typing(x.Name, ∅), Typing(x.Sal > 10000, ∅)))
5       )     )Typing(x.Name, ∅)
6       )     )   = P_DOT(Typing(x, ∅), Name)
7       )     )     )Typing(x, ∅) = ({t1 :: U}, {x : t1}, t1)
8       )     )   = ({t2 :: U, t1 :: [[Name : t2]]}, {x : t1}, t2)
9       )     )Typing(x.Sal > 10000, ∅)
10      )     )   = P_>(Typing(x.Sal, ∅), Typing(10000, ∅))
11      )     )     )Typing(x.Sal, ∅) = ({t3 :: U, t4 :: [[Sal : t3]]}, {x : t4}, t3)
12      )     )     )Typing(10000, ∅) = (∅, ∅, int)
13      )     )   = ({t4 :: [[Sal : int]]}, {x : t4}, bool)
14      )   = ({t2 :: U, t1 :: [[Name : t2, Sal : int]]}, {x : t1}, (t2, bool))
15    = ({t2 :: U, t1 :: [[Name : t2, Sal : int]]}, ∅, t1 → (t2, bool))
```

Figure 10: Examples of Type Inference with Records

3.5 Further Refinement and the Correctness of the Type Inference System

In the explanation of type inference algorithm so far, we have ignored the constraint that some type variables should only denote description types. The necessary extension is to introduce description kind constructors D, $[\![l : \delta, \ldots, l : \delta]\!]_d$ and $\langle\!\langle l : \delta, \ldots, l : \delta \rangle\!\rangle_d$ respectively denoting the set of all description types, description record types, and description variant types. Although it increases the notational complexity, these extension can be easily incorporated with the unification algorithm and the type inference.

Another simplification we made in the description of the type inference algorithm is our assumption that types are all non cyclic. To extend the type inference algorithm to recursive types, we only need to extend the kinded unification algorithm to infinite regular trees. The necessary extension is similar to the one needed to extend an ordinary unification algorithm to regular trees [Cou83], which involves: (1) defining a data structure to represent regular trees. (2) changing the cases for variable elimination (cases of II and IV) by eliminating occur-check and replacing the one point substitution $[t \mapsto \tau]$ by the substitution $[t \mapsto \overline{(\mathsf{rec}\ v.\tau[v/t])}]$ where $\overline{(\mathsf{rec}\ v.\tau[v/t])}$ is a regular tree that is a solution to $v = \tau[v/t]$, and (3) changing the cases for decomposition (cases V and VI) so that they generate the equations for the set of pairs of corresponding subtrees of the given regular trees.

We have also ignored the details of dealing with references. The above type inference method cannot be directly extended to references, since the operational semantics for references does not agree with polymorphic type discipline for **let** binding. As pointed out in [Mac88, Tof88], the straightforward application of the type inference method of [Mil78] to references yields unsound type system. The following example is given in [Mac88]:

```
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 **unit * 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. Solutions have been proposed in [Tof88, Mac88]. They differ in details treatment but they are both based on the idea that the type system restricts substitution on type variables in reference types in such a way that references created by a polymorphic functions are monomorphic. Since both of these mechanisms can be regarded as a new form of kind constraint on type variables, we believe that either of them can safely be incorporated with our type system. However, for want of a better mechanism, we restrict reference constructor to take only a monomorphic type.

With these refinements, ML's complete static type inference is extended to records, variants and set data types, as stated in the following result:

Theorem 2 *Let e be any raw term of Machiavelli. If $Typing(e, \emptyset) = (\mathcal{K}, \mathcal{A}, \tau)$ then $\mathcal{K}, \mathcal{A} \rhd e : \tau$ is a principal typing of e. If $Typing(e, \emptyset)$ reports failure then e has no typing.*

Just as legal ML programs correspond to principal typing schemes with empty type assignment, legal Machiavelli programs correspond to principal kinded typing schemes with empty type assignment, ie. typings of the form $\mathcal{K}, \emptyset \rhd e : \tau$. Machiavelli prints a typing $\mathcal{K}, \emptyset \rhd e : \tau$ as

$e : \tau'$

where τ' is a type whose type variables are printed together with their kind constraints in $\mathcal{K}$ in the following formats:

type variables t with $\mathcal{K}(t) = \mathsf{U}, \ldots$	'a,'b,...
description type variables d with $\mathcal{K}(t) = \mathsf{D}, \ldots$	"a,"b,...
type variables t with $\mathcal{K}(t) = [\![l_1 : \tau_1, \ldots, l_n : \tau_n]\!], \ldots$	'a::[l_1:τ_1,...,l_n:τ_n],...
description type variables d with $\mathcal{K}(t) = [\![l_1 : \tau_1, \ldots, l_n : \tau_n]\!]_d, \ldots$	"a::[l_1:τ_1,...,l_n:τ_n],...
type variables t with $\mathcal{K}(t) = \langle\!\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle\!\rangle, \ldots$	'a::<l_1:τ_1,...,l_n:τ_n>,...
description type variables d with $\mathcal{K}(t) = \langle\!\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle\!\rangle_d, \ldots$	"a::<l_1:τ_1,...,l_n:τ_n>,...

as already seen in examples. Thus the type output in the following example

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

is a representation of the following kinded typing scheme:

$$\{t_2 :: U, t_1 :: [\![\mathsf{Name} \ : \ t_2]\!]\}, \emptyset \rhd \mathsf{fn\ x{=>}\ x.Name} \ : \ t_1 \rightarrow t_2$$

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 nature 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, specifically, *projection* and *natural join*, which 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 system and increases the complexity of the type inference problem. The typing rule for *join* is associated with a complex condition which can no longer be represented by a 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. (See also [CM89, HP91] for polymorphic calculi with record concatenation.) 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. Next, 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 represents the exact set of provable typings. Finally, we describe the method to check the satisfiability of conditions before the evaluation of the term associated with those conditions. In other words, we are still able to guarantee that a typechecked program will not cause a runtime 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 this end Machiavelli generalizes the following four operations to arbitrary description terms and introduces them as polymorphic functions in its type system:

$\mathsf{eq}(e_1,e_2)$	*equality test,*
$\mathsf{join}(e_1,e_2)$	*database join,*
$\mathsf{con}(e_1,e_2)$	*consistency check,*
$\mathsf{project}(e,\delta)$	*projection of d onto the type δ.*

The intuition underlying their generalization is the idea exploited in [BJO91] 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 [Oho90]. In what follows, we describe how equality, join and projection are generalized to acyclic 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 [Oho90].

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

$$t_1 = [\mathsf{Name} = [\mathsf{First} = \text{"Joe"}]];$$

and

$$t_2 = [\mathsf{Name} = [\mathsf{Last} = \text{"Doe"}]]$$

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

$$t = [\mathsf{Name} = [\mathsf{First} = \text{"Joe"}, \mathsf{Last} = \text{"Doe"}]].$$

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:

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

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.

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

This approach also provides a uniform treatment of *null values* [Zan84, Bis81], which are used in databases that represent incomplete information. Join and projection extend smoothly to data containing null values. However care must be taken [Lip79, IL84] to ensure that use of the algebra with these extended operations provides the semantics intended by the programmer. To represent null values, we also extend the syntax of Machiavelli terms with:

$\mathsf{null}(b)$	*the null value of a base type b*
<>	*the (polymorphic) null value of variant types*

Other incomplete values can be built from these using the constructors for description terms.

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 obtain such an ordering, we first define the pre-order $\preceq$ on description terms. For acyclic descriptions, $\preceq$ is given as:

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

The last rule for sets is intended 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 its equivalence class. In what follows, we denote by $\sqsubseteq$ the ordering induced by the preorder $\preceq$. Among representatives, there is a canonical one having the property that it does not contain a set term whose members are comparable, i.e. an anti-chain. 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, which computes a canonical representation of the denoted equivalence class. The equality (eq) is a generalization of *structural equality* to sets and null values. Figure 11 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 [BJO91] it is shown that:

Theorem 3 *If r_1, r_2 are first-normal form relations then* $\mathsf{join}(r_1,r_2)$ *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 description type, such as {int}.

```
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 11: Natural join of higher-order relations

We now turn to projection. In the relational model, it is defined as a projection on a set of labels. We generalize it to an operation which projects a complex description onto some "substructure". In a programming language, the 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 ground description types (i.e. those description types that do not contain type variables) is

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

Projection is therefore 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] }
```

By using the ordering we have just defined, projection can be specified as:

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

which can be shown to be computable for any description type δ.

4.2 Extended Expressions and Their Evaluation

The syntax of expressions is extended with the constants null(b) and <> and the term constructors join, con, and project we have just described:

$$e ::= \cdots \mid \mathsf{null}(b) \mid \langle\rangle \mid \mathsf{join}(e,e) \mid \mathsf{con}(e,e) \mid \mathsf{project}(e,\delta)$$

We extend the evaluation rules for expressions described in section 3 with the rules for these 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:

$\mathsf{join}(d_1,d_2) \rightarrowtail d_3$	*if* d_3 *is a canonical representative of* $D(d_1) \sqcup D(d_2)$
$\mathsf{con}(d_1,d_2) \rightarrowtail \mathsf{true}$	*if* $D(d_1) \sqcup D(d_2)$ *exists*
$\mathsf{con}(d_1,d_2) \rightarrowtail \mathsf{false}$	*if* $D(d_1) \sqcup D(d_2)$ *does not exist*
$\mathsf{project}(d_1,\delta) \rightarrowtail d_2$	*if* d_2 *is a canonical representative of the least upper bound of the set* $\{D(d) \mid D(d) \sqsubseteq D(d_1), d:\delta\}$
$\mathsf{eq}(d_1,d_2) \rightarrowtail \mathsf{true}$	*if* $D(d_1) \sqsubseteq D(d_2)$ *and* $D(d_2) \sqsubseteq D(d_1)$
$\mathsf{eq}(d_1,d_2) \rightarrowtail \mathsf{false}$	*if* $D(d_1) \not\sqsubseteq D(d_2)$ *or* $D(d_2) \not\sqsubseteq D(d_1)$

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

4.3 Type Inference for Relational Algebra

join, project and con are polymorphic operations in the sense that they compute join and projection of various types. To represent their exact polymorphic nature, we define an ordering on ground description types that represents the ordering on the structure of descriptions. For the set of acyclic description types, the necessary ordering is given by the following inductive definition:

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

Using this ordering, types of join, project, and con are given as:

$$
\begin{aligned}
\mathsf{join} &: \delta_1 * \delta_2 \rightarrow \delta_3 \text{ such that } \delta_3 = \delta_1 \sqcup_{\ll} \delta_2 \\
\mathsf{project}(\_,\delta_2) &: \delta_1 \rightarrow \delta_2 \text{ such that } \delta_2 \ll \delta_1 \\
\mathsf{con} &: \delta_1 * \delta_2 \rightarrow \mathit{bool} \text{ such that } \delta_1 \sqcup_{\ll} \delta_2 \text{ exsists}
\end{aligned}
$$

To integrate these operations with the polymorphic core of Machiavelli defined in section 3, we need to represent the types of these operations into the type system. For this purpose, we explicitly introduce syntactic conditions on substitution of type variables that represent the three forms of constraint: $\delta_1 \sqcup_{\ll} \delta_2$ exists, $\delta = \delta_1 \sqcup_{\ll} \delta_2$, and $\delta_2 \ll \delta_1$. In fact we only need to consider the last two forms of constraint since $\delta_1 \sqcup_{\ll} \delta_2$ will exist whenever we can find a type $\delta_3 = \delta_1 \sqcup_{\ll} \delta_2$. To represent them we introduce the following syntactic conditions:

1. $\tau = \mathit{jointype}(\tau,\tau)$, and
2. $\mathit{lessthan}(\tau,\tau)$.

(NULL1) $C, \mathcal{K}, \mathcal{A} \rhd \mathsf{null}(b) : b$

(NULL2) $C, \mathcal{K}, \mathcal{A} \rhd \langle\rangle : \delta \quad$ if $\mathcal{K} \vdash \delta :: \langle\langle\rangle\rangle$

(CON) $$\frac{C, \mathcal{K}, \mathcal{A} \rhd e_1 : \delta_1 \qquad C, \mathcal{K}, \mathcal{A} \rhd e_2 : \delta_2}{C \cup \{d = \mathit{jointype}(\delta_1, \delta_2)\}, \mathcal{K}, \mathcal{A} \rhd \mathsf{con}(e_1, e_2) : \mathit{bool}\ (\ d \text{ fresh}\ )}$$

(JOIN) $$\frac{C, \mathcal{K}, \mathcal{A} \rhd e_1 : \delta_1 \qquad C, \mathcal{K}, \mathcal{A} \rhd e_2 : \delta_2}{C \cup \{d = \mathit{jointype}(\delta_1, \delta_2)\}, \mathcal{K}, \mathcal{A} \rhd \mathsf{join}(e_1, e_2) : d}$$

(PROJECT) $$\frac{C, \mathcal{K}, \mathcal{A} \rhd e : \delta_1}{C \cup \{\mathit{lessthan}(\delta_2, \delta_1)\}, \mathcal{K}, \mathcal{A} \rhd \mathsf{project}(e_1, \delta_2) : \delta_2}$$

Figure 12: The Typing Rules for Relational Operations

Note the difference between $\delta_3 = \delta_1 \sqcup_{\ll} \delta_2$ and $\tau_3 = \mathit{jointype}(\tau_1, \tau_2)$. The former is a property on the relationship between three ground description types. On the other hand, the latter is a syntactic formula denoting the constraint on substitutions of type variables in τ_1, τ_2, τ_3 to ensure that any ground instance of the former these satisfies such a property. A similar remark holds for $\delta_1 \ll \delta_2$ and $\mathit{lessthan}(\tau_1, \tau_2)$. Using these syntactic conditions on type variables, we can extend the type system to incorporate these new operations. A typing judgement in the extended system has the form $C, \mathcal{K}, \mathcal{A} \rhd e : \tau$ where the extra ingredient C is a set of syntactic conditions we have just introduced. Figure 12 shows the typing rules for the new operations. Other rules remain the same as those defined in Figure 2 and 6 except that they are now relative to a given set of conditions. For example, the rule ABS becomes

(ABS) $$\frac{C, \mathcal{K}, \mathcal{A}(x, \tau_1) \rhd e : \tau_2}{C, \mathcal{K}, \mathcal{A} \rhd \mathsf{fn}\ x \Rightarrow e : \tau_1 \rightarrow \tau_2}$$

In particular, these other rules only propagate the given set of conditions and do not change its contents.

Since the conditions we introduced involve the ordering that is defined only on *ground* types, we need to interpret a typing judgement in this extended system as a *scheme* representing the set of all *ground typings* obtained by substituting its type variables with appropriate ground types. This interpretation is consistent with our treatment of let construct (LET rule in Figure 2) and its semantics described in [Oho89a]. A ground substitution θ *satisfies* a condition c if

1. if $c \equiv \tau_1 = \mathit{jointype}(\tau_2, \tau_2)$ then $\theta(\tau_1), \theta(\tau_2), \theta(\tau_3)$ are all description types satisfying $\theta(\tau_1) = \theta(\tau_2) \sqcup_{\ll} \theta(\tau_3)$,
2. if $c \equiv \mathit{lessthan}(\tau_1, \tau_2)$ then $\theta(\tau_1), \theta(\tau_2)$ are description types satisfying $\theta(\tau_1) \ll \theta(\tau_2)$.

θ satisfies a set C of conditions if it satisfies each member of C. We say that a ground typing $\emptyset, \emptyset, \mathcal{A} \rhd e : \tau$ is an instance of $C, \mathcal{K}, \mathcal{A}' \rhd e : \tau'$ if there is a ground substitution θ that respects $\mathcal{K}$ and satisfies C such that $\mathcal{A}\uparrow^{dom(\mathcal{A}')} = \theta(\mathcal{A}')$ and $\tau = \theta(\tau')$. As seen in this definition, a typing in the extended system is subject to a set of conditions associated with it. To emphasize this fact, we call typing judgement in the extended type system a *conditional typing*. A conditional typing scheme $C, \mathcal{A} \rhd e : \tau$ is *principal* if any derivable ground typing for e is an instance of it. The following result establishes the complete inference of principal conditional typing schemes.

```
-> fun join3(x,y,z) = join(x,join(y,z));
>> val join3 = fn : ("a * "b * "c) -> "d
   where { "d = jointype("a,"e), "e = jointype("b,"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 13: Some Simple Relational Examples

Theorem 4 *There is an algorithm which, given any raw term e, returns either* failure *or a tuple* $(C, \mathcal{K}, \mathcal{A}, \tau)$ *such that if it returns* $(C, \mathcal{K}, \mathcal{A}, \tau)$ *then* $C, \mathcal{K}, \mathcal{A} \rhd e : \tau$ *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 [Oho89b].

Figure 13 gives two simple examples of the typing schemes that are inferred by Machiavelli. The type ("a * "b * "c) -> "d where { "d = jointype("a,"e), "e = jointype("b,"c) } of the three-way join join3 is the representation of the principal conditional typing scheme:

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

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 beginning 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 a set of conditions. Unfortunately, however, the satisfiability checking cannot be made efficient since it is shown that [OB88] that checking these conditions is itself NP-complete. A practical solution is to *delay* the satisfiability check of a set of conditions until its type variables are fully instantiated. Once the types of all type variables in a condition are known, its satisfiability can be efficiently checked and it can then 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 built out of records, variants and sets. It allows us to define directly in Machiavelli databases supporting complex structures including non-first-normal form relations, nested relations and complex objects. Figure 14 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 15 shows some simple query processing

```
-> 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 14: A Part-Supplier Database in Generalized Relational Model

for the database example in figure 14. 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 the operations of the relational data model and the types available in current programming languages. 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 16 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 17 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. In the case of a composite part, it first generates a set of record consisting of a subpart number and its cost and then accumulates the costs of subparts by using hom. In order to prevent the set constructor from collapsing subpart costs which are equal, the computed subpart cost is paired with the subpart number. 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 with instantiations of **"a** in the type of **expensive-parts**. Also, the apparent 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. This is particularly useful when we

```
(*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 15: 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 16: A Simple Implementation of Polymorphic Transitive Closure

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 Heterogeneous sets

The previous section provided an extension to a polymorphic type system for records that enabled us to infer the type-correctness of programs that involve operations of the relational algebra – notably *projection* and *join*. This extension involved an ordering on types and joins on types.It could be argued that there is little point in doing this, because in practical query languages *projection* and *join* are not used. As we have seen in section 2, we may implement an SQL-like sublanguage using cartesian product together with the operations on records (formation and field selection) described in section 3. Apparently the use of an ordering on types and joins on types is only of academic interest!

The authors believe otherwise. Extensions to the mechanisms used in section can be used to address a problem that arises in object-oriented databases, where there is an apparent need for the use of *heterogeneous collections*. The problem arises from two apparently contradictory uses of inheritance that arise in programming languages and in databases. In object-oriented languages the term describes code sharing: by an assertion that *Employee* inherits from *Person* we mean that the methods defined for the class *Person* are also applicable to instances of the class *Employee*. In databases – notably in data modeling techniques – we associate sets $Ext(Person)$ and $Ext(Employee)$ with the entities *Person* and *Employee* and the inheritance of *Employee* from *Person* specifies set inclusion: $Ext(Employee) \subseteq Ext(Person)$.

It seems that these two notions should somehow be coupled, but on the face of it there is a contradiction. If members of $Ext(Employee)$ are instances of *Employee*, how can they be members of $Ext(Person)$ whose members must all be instances of *Person*? One way out of this is to relax what we mean by "instance of" and to allow an instance of *Employee* also to be an instance of *Person*. We can now take $Ext(Person)$ as a heterogeneous set, some of whose members are also instances of *Employee*. Type systems, however, can make the manipulation of heterogeneous collections difficult or impossible by "losing" information. For example if l has type $list(Person)$ and e has type *Employee*, the result of $insert(e, l)$ will still have type $list(Person)$, and the first element of this list will only have type *Person*. By inserting e into l the type system has somehow "lost" part of the structure of e such as the availability of a *Salary* field or method. This problem appears both in languages with a subsumption rule [Car88] and in statically type-checked object-oriented languages such as C++ [Str87] which claim the ability to represent heterogeneous collections as an important feature. In some cases the information is not recoverable; in others it can only be recovered in a rather dangerous fashion by asking the programmer to maintain information about the type of an object and to re-cast those objects on the basis of this information. A solution to this problem was described by the authors in [BO91]. The approach decribed here fits uniformly with the techniques developed in the preceding sections.

5.1 Dynamic and partial values

Before proceeding further, it is important to make a distinction concerning type systems which is, roughly, the distinction between statically and dynamically typed languages. Our approach to type systems has so far been *syntactic*; we have used types (more specifically type inference) to describe the well-formed expressions of our language. For our language there is an extension of a result due to Milner, that well-formed expressions

```
(*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 17: Query Processing Using Polymorphic Functions

do not go "wrong" in that they do not allow an operation to be applied to a value of an inappropriate type. But this syntactic approach does not immediately tell us whether, or in what form, types should be present in the evaluation of an expression. Very little type information is carried in the executable code of an ML or Pascal program, while in the implementation of dynamically typed languages such as Lisp or Smalltalk, each value carries enough information to determine its type. Moreover, in dynamically typed languages this information is available to the programmer in the form of expressions such as `(INTEGERP X)`,which allow us to interrogate the type of a variable. Allowing such expressions negates, in general, any possibility of static type-checking. However, by suitably containing the way in which type information is used in the execution of a program, one may obtain the many of the benefits of dynamic type checking in a statically-typed framework. The idea, due to Cardelli and Mycroft [Car86] is to use *dynamic* values. These are values that carry their type with them, and can be regarded as a pair consisting of a type and a value of that type. A formal system for type systems with *dynamic* was developed in [ACPP91].

In these proposals there are two operations on dynamic values; at any type τ we have:

dynamic : τ -> dynamic
coerce(τ) : dynamic -> τ

The function dynamic creates a value of type dynamic out of a value of any type – operationally it pairs the value with its type. Conversely coerce(τ) takes such a pair and returns the value component provided the type component is τ. It raises an exception otherwise. A standard use for dynamic values is for representing persistent data, since the type of external data cannot be guaranteed. For example 2 + coerce(int)(read(input_stream)) will either add 2 to the input or raise an exception. The coerce operation can be thought of as a localized dynamic type-check, and an exception-handling mechanism is apparently needed to deal with the possibility of failure.

Our approach to heterogeneous collections is to generalize the notion of a dynamic type to one in which *some* of the structure is visible. A type $\mathcal{P}$`([Name : string, Age : int])` denotes dynamic values whose actual type δ is "bigger" than [Name : string, Age : int], i.e. [Name : string, Age : int] $\ll \delta$ where $\ll$ is the ordering we used to represent types of relational operators. Thus an assertion of the form e:$\mathcal{P}$([Name : string, Age : int]) means that e is a dynamic value, but it is known to be a record and that least Name and Age fields are available on e. We shall refer to such partially specified dynamic values as *partial values*. Note that a partial value is like a dynamic value in that it always carries its (complete) type. The new type constructor $\mathcal{P}$ allows us to mix those partial values with other term constructors in the language. For example, $e' : \{\mathcal{P}(\delta)\}$ means that e' is a set of objects each of which is a partial value whose complete type is bigger than δ (under the ordering $\ll$.) It is this use of the ordering on types in conjunction with a set type that allows us to express heterogeneous collections. An assertion of the form $e : \{\mathcal{P}$([Name : string, Age : int])$\}$ means that e is a set of records, each of which has at least a Name : string and Age : int field, and therefore relational queries involving only selection of these fields are legitimate. As a special case of partial types, we introduce a constant type any denoting dynamic values on which no information is known – it is a (completely) dynamic value.

To show the use of partial types, let us assume that the following names have been given for partial types:

Person* for $\mathcal{P}$([Name : string, Address : string])
Employee* for $\mathcal{P}$([Name : string, Address : string, Salary : int])
Customer* for $\mathcal{P}$([Name : string, Address : string, Balance : int])

Also suppose that DB is a set of type {any} so that we initially have no information about the structure of members of this set. Here are some examples of how such a database may be manipulated in a type-safe language

1. An operation filter $\mathcal{P}(\delta)$ (S) can be defined, which selects all the elements of S which have partial type $\mathcal{P}(\delta)$, i.e. filter $\mathcal{P}(\delta)$ (S) : $\{\mathcal{P}(\delta)\}$. We may use this in a query such as

   ```
   select [Name=x.Name, Address=x.Address]
   from x <- filter Employee* (DB)
   where x.Salary > 10,000
   ```

   The result of this query is a set of (complete) records, i.e. a relation. There is some similarity with the * form of Postgres [SR86], however we may use filter on arbitrary kinds and heterogeneous sets; we are not confined to the extensionally defined relations in the database.

2. Under our interpretation of partial types, if $\delta_1 \ll \delta_2$ then $\mathcal{P}(\delta_1)$ is more partial than $\mathcal{P}(\delta_2)$ and any partial value of type $\mathcal{P}(\delta_2)$ also has type $\mathcal{P}(\delta_1)$. This property can be used to represent the desired set inclusion in the type system. In particular, Person* is more partial than Employee*. From this, the inclusion filter Employee* $(S) \subseteq$ filter Person* (S) will always hold for any heterogeneous set S, in particular for the database DB. Thus the "data model" (inclusion) inheritance is *derived* from a property of type system rather than being something that must be achieved by the explicit association of extents with classes.

3. We have the ability to write functions such as

   ```
   fun RichCustomers(S) = select [Name=x.Name, Balance=x.Balance]
                          from x <- intersect(S,filter Customer* (DB))
                          where x.Salary > 30,000
   ```

   Type inference allows the application of this function to any heterogeneous set each members of which has at least the type $\mathcal{P}$([Salary : int]). The result is a uniformly typed set, i.e. a set of type {[Name : string, Balance : int]}. Thus the application RichCustomers(filter Employee* (DB)) is valid, but the application RichCustomers(filter Customer* (DB)) does not have a type, and this will be statically determined by the failure of type inference.

4. By modifying the technique we used to give a polymorphic type of join, we can define the typing rules for unions and intersections of heterogeneous sets. By adding a partial type any, the partialness ordering has meet and join operations. The union and intersection of heterogeneous sets have, respectively, the join and meet of their partial types. Thus, the type system can infer an appropriate partial type of heterogeneous set obtained by various set operations. For example, the following typings are inferred.

   ```
   union(filter Custmer* (DB), filter Employee* (DB))
     : {P([Name : string, Address : string])}

   intersection(filter Customer* (DB), filter Employee* (DB))
     : {P([Name : string, Address : string, Salary : int, Balance : int])}.
   ```

(intersection is definable in the language) These inferred types automatically allow appropriate polymorphic functions to be applied to the result of these set operations. For example, since the type of an intersection of two heterogeneous sets is the join of the types, polymorphic functions applicable to either of the two sets are applicable to the intersection. Thus, we successfully achieve the desired coupling of set inclusion and method inheritance.

In the following subsections we shall describe the basic operations for dealing with sets and partial values. We shall then give typing rules to extend Machiavelli to include those partial values.

5.2 The Basic Operations

To deal with partial values we introduce four new primitive operations: dynamic, as, coerce and fuse. We also extend the meaning of some of the existing primitives, such as union

dynamic(e). This is used to construct a partial value and has type $\mathcal{P}(\delta)$ where δ is the type of e. A heterogeneous set may be constructed with

{dynamic([Name = "Joe", Age = 10]), dynamic([Name = "Jane", Balance = 10954])}

This expression implicitly makes use of union, and as a result of the extended typing rules for union, the expression has type $\{\mathcal{P}([\text{Name} : \text{string}])\}$, which is the meet of $\{\mathcal{P}([\text{Name} : \text{string}, \text{Age} : \text{int}])\}$ and $\{\mathcal{P}([\text{Name} : \text{string}, \text{Balance} : \text{int}])\}$.

The remaining three primitives may all fail. Rather than introduce an exception handling mechanism, we adopt the strategy that if the operation succeeds, we return the result in a singleton set, and if it fails, we return the empty set[3].

as $\mathcal{P}(\delta)$ (e). This, for any description type δ, "exposes" the properties of e specified by the type δ. This returns a singleton set containing the partial value if the coercion is possible and the empty set if it is not. For example, if e = as $\mathcal{P}$([Name : string]) (dynamic([Name = "Joe", Balance = 43.21])), e will have partial type $\{\mathcal{P}([\text{Name} : \text{string}])\}$ and an expression such as select x.Name from x <- e will type check, while select x.Balance from x <- e will not.

Using as and hom we are now in a position to construct the filter operation, mentioned earlier, which ties the inclusion of extents to the ordering on types. Because we do not have type parameters, it cannot be defined in the language. However it can be treated as a syntactic abbreviation:

$$\text{filter } \mathcal{P}(\delta)\ (S) \equiv \text{hom}(\text{fn x => as } \mathcal{P}(\delta)\ (\text{x}),\text{union}, S, \{\})$$

coerce δ (e). This coerces the partial value denoted by e to a (complete) value of type δ. It will only succeed if the type component of e is δ. Again, if the operation succeeds we return the singleton set, otherwise we return the empty set. For example coerce [Name : string] (dynamic([Name = "Jane", Balance = 10954])) will yield the empty set while coerce [Name : string, Balance : int] (dynamic([Name = "Jane", Balance = 10954])) will return the set {[Name = "Jane", Balance = 10954]} fuse(e_1, e_2). This combines the partial

[3]This mechanism, while it fits naturally with our operations on sets and provides concise implementations of a number of useful functions, may, if improperly used, produce results that are open to misinterpretation – "extensional query failures" discussed by linguists [Kap81].

values denoted by e_1 and e_2. It will only succeed if the (complete) values of e_1 and e_2 are equal. If e_1 has partial type $\mathcal{P}(\delta_1)$ and e_2 has partial type $\mathcal{P}(\delta_2)$ then fuse(e_1, e_2) will have the partial type $\mathcal{P}(\delta_1 \sqcup_{\ll} \delta_2)$. If

e_1 =(dynamic([Name = "Jane", Age = 21, Balance = 10954])),
e_2 = as $\mathcal{P}$([Name : string]) e_1,
e_3 = as $\mathcal{P}$([Age : int]) e_1, and
e_4 = as $\mathcal{P}$([Name : string]) dynamic([Name = "Jane"]),

then fuse(e_2, e_3) will be a singleton set of type $\{\mathcal{P}$([Name : string, Age : int])$\}$ while fuse(e_2, e_4) will return an empty set. fuse may be used to define set intersection as in

```
fun fuse1(x,s) = hom(fn y => fuse(x,y), union, s, {})
fun intersection(s1,s2) = hom(fn y => fuse1(y,s2), union, s1, {})
```

Note that in some sense fuse can be regarded as an operation that is more basic than equality for we can compute whether the partial values v_1 and v_2 are equal (as complete values) by empty(fuse(v_1, v_2)). Complete values have nothing to do with "object identity". The combination of partial types with some form of reference does not appear to represent any great difficulties, but is not dealt with here.

5.3 Extension of the Language

To incorporate these partial values, we extend the definition of the language. The set of types is extended to include any and the partial type constructor $\mathcal{P}(\delta)$:

$\tau ::= \cdots \mid$ any $\mid \mathcal{P}(\delta)$

We identify the following subset (ranged over by π) which may contain partial types.

$\pi ::=$ d $\mid b_d \mid [l{:}\pi,\ldots,l{:}\pi] \mid \langle l{:}\pi,\ldots,l{:}\pi\rangle \mid \{\pi\} \mid$ ref$(\pi) \mid$ any $\mid \mathcal{P}(\delta)$

The set of terms is extended to include operations for partial values.

$e ::= \cdots \mid$ dynamic$(e) \mid$ fuse$(e,e) \mid$ as $\mathcal{P}(\delta)$ $e \mid$ coerce δ e

To extend the type system to those new term constructors for partial values, we define an ordering on the above subset of types, which represents the partialness of types. We write $\pi \lesssim \pi'$ to denote that π is more partial than π'. The rules to define this ordering are:

$$
\begin{aligned}
\text{any} &\lesssim \mathcal{P}(\delta) \text{ for any } \delta \\
\mathcal{P}(\delta_1) &\lesssim \mathcal{P}(\delta_2) \quad \text{if } \delta_1 \ll \delta_2 \\
b_d &\lesssim b_d \\
[l_1{:}\pi_1,\ldots,l_n{:}\pi_n] &\lesssim [l_1{:}\pi'_1,\ldots,l_n{:}\pi'_n] \quad \text{if } \pi_i \lesssim \pi'_i \ (1 \le i \le n) \\
\langle l_1{:}\pi_1,\ldots,l_n{:}\pi_n\rangle &\lesssim \langle l_1{:}\pi'_1,\ldots,l_n{:}\pi'_n\rangle \quad \text{if } \pi_i \lesssim \pi'_i \ (1 \le i \le n) \\
\{\pi\} &\lesssim \{\pi'\} \quad \text{if } \pi \lesssim \pi' \\
\text{ref}(\pi) &\lesssim \text{ref}(\pi') \quad \text{if } \pi \lesssim \pi'
\end{aligned}
$$

$$\text{(DYNAMIC)}\quad \frac{C,\mathcal{K},\mathcal{A} \rhd e : \delta}{C,\mathcal{K},\mathcal{A} \rhd \mathsf{dynamic}(e) : \mathcal{P}(\delta)}$$

$$\text{(AS)}\quad \frac{C,\mathcal{K},\mathcal{A} \rhd e : \mathcal{P}(\delta)}{C,\mathcal{K},\mathcal{A} \rhd \mathsf{as}\ \mathcal{P}(\delta')\ e : \{\mathcal{P}(\delta')\}}$$

$$\text{(COERCE)}\quad \frac{C,\mathcal{K},\mathcal{A} \rhd e : \mathcal{P}(\delta)}{C,\mathcal{K},\mathcal{A} \rhd \mathsf{coerce}\ \delta'\ e : \{\delta'\}}$$

$$\text{(FUSE)}\quad \frac{C,\mathcal{K},\mathcal{A} \rhd e_1 : \pi_1 \qquad C,\mathcal{K},\mathcal{A} \rhd e_2 : \pi_2}{C \cup \{d = \mathit{jointype}_{\lesssim}(\pi_1,\pi_2)\},\mathcal{K},\mathcal{A} \rhd \mathsf{fuse}(e_1,e_2) : \{d\}}$$

$$\text{(UNION)}\quad \frac{C,\mathcal{K},\mathcal{A} \rhd e_1 : \{\pi_1\} \qquad C,\mathcal{K},\mathcal{A} \rhd e_2 : \{\pi_2\}}{C \cup \{d = \mathit{meettype}_{\lesssim}(\pi_1,\pi_2)\},\mathcal{K},\mathcal{A} \rhd \mathsf{union}(e_1,e_2) : \{d\}}$$

Figure 18: Typing Rules for Partial Values

The first two of these rules derive the order on partial types directly from the ordering $\ll$ that we introduced in section 4. The remaining rules lift this ordering component-wise to all description types. The following are examples of this ordering.

$$\mathcal{P}([\mathsf{Name} : \mathsf{string}, \mathsf{Address} : \mathsf{string}]) \lesssim \mathcal{P}([\mathsf{Name} : \mathsf{string}, \mathsf{Address} : \mathsf{string}, \mathsf{Balance} : \mathsf{int}])$$

$$[\mathsf{Acc\_No} : \mathsf{int}, \mathsf{Customer} : \mathcal{P}([\mathsf{Name} : \mathsf{string}, \mathsf{Address} : \mathsf{string}, \mathsf{Balance} : \mathsf{int}])] \lesssim [\mathsf{Acc\_No} : \mathsf{int}, \mathsf{Customer} : \mathcal{P}([\mathsf{Name} : \mathsf{string}, \mathsf{Address} : \mathsf{string}, \mathsf{Balance} : \mathsf{int}, \mathsf{Salary} : \mathsf{int}])]$$

Figure 18 gives the typing rules for the new term constructors. The new condition $d = \mathit{jointype}_{\lesssim}(\mathcal{P}(\pi_1),\mathcal{P}(d))$ used in rules (FUSE) denotes the condition on the ground substitutions θ such that $\theta(d) = \theta(\pi_1) \sqcup_{\lesssim} \theta(\pi_2)$, and the condition $d = \mathit{meettype}_{\lesssim}(\pi_1,\pi_2)$ used in the rule (UNION) denotes the ground substitutions θ such that $\theta(d) = \theta(\pi_1) \sqcap_{\lesssim} \theta(\pi_2)$.

Standard elimination operations introduced in Section 2 and database operations we defined in Section 4 are not available on types containing the partial type constructor $\mathcal{P}$. The only exception is the field selection, which requires only partial information on types specified by kinds. From an expression e of type of the form $\mathcal{P}([\ldots,l{:}\delta,\ldots])$, the l field can be safely extracted. The result of the field selection $e.l$ is δ itself if δ is a base type. However, if δ is a compound type then the actual type of the l field of the expression e is some δ' such that $\delta \lesssim \delta'$. In this case, the type of the result of field selection $e.l$ is the partial type $\mathcal{P}(\delta)$. Recall the typing rule for field selection:

$$\text{(DOT)}\quad \frac{\mathcal{K},\mathcal{A} \rhd e : \tau_1 \qquad \mathcal{K} \vdash \tau_1 :: [\![l : \tau_2]\!]}{\mathcal{K},\mathcal{A} \rhd e.l : \tau_2}$$

To make this rule to be applicable to the above two cases for partial values, we only need to define the following kinding rule for partial types.

$$\mathcal{K} \vdash \mathcal{P}([l_1{:}\delta_1,\ldots,l_n{:}\delta_n,\ldots]) :: [\![l_1{:}\pi_1,\ldots,l_n{:}\pi_n]\!] \quad \text{where } \pi_i = \delta_i \text{ if } \delta_i \text{ is a base type otherwise } \pi_i = \mathcal{P}(\delta_i)$$

Other rules defined in Figure 5 remain unchanged except that types may contain partial types. A record kind now ranges also over partial types and the field selection becomes polymorphic over partial types as well as complete types.

```
-> DB;
>> val it = {···} : {any}
-> val employees = filter Employee* DB;
>> val employees = {···} : {P([Name : string, Address : string, Salary : int])}
-> val customers = filter Customer* DB;
>> val customers = {···} : {P([Name : string, Address : string Balance : int])}
-> union(employees,customers);
>> val it = {···} : {P([Name : string,Address : string])}
-> intersection(employees,customers);
>> val it = {···} : {P([Name : string,Address : string, Balance : int, Salary : int])}

-> fun RichEmployees S = select x.Name
                         from x <- S
                         where x.Salary > 30,000
>> val RichEmployees = fn : {'a::[Salary : int, Name : 'b]} -> {'b}
-> RichEmployees(employees) : {···} : {string}

-> fun GoodCustomers S = select x.Name
                          from x <- S
                          where x.Balance > 3,000
>> val GoodSustomers = fn : {'a::[Balance : int, Name : 'b]} -> {'b}

-> fun GoodEmployees S = intersection(GoodCustomers(S),RichEmployees(S));
>> val GoodEmployees = fn : {'a::[Balance : int, Salary : int, Name : 'b]} -> {'b}
-> GoodEmployees(intersection(employees,customers));
>> val it = {···} : {string}
```

Figure 19: Programming with Heterogeneous Sets

For this extended language, we can still have a complete type inference algorithm. The necessary technique is essentially the same as that for typechecking join operation we have described in the previous section. We then have a language that uniformly integrate heterogeneous sets in its type system. For example, the function

```
wealthy : {'a::[Name : 'b, Salary : int]} -> {'b}
```

we defined in the introduction may also be applied to heterogeneous sets of type such as {$\mathcal{P}$([Name : string, Salary : int])} . Figure 19 gives examples involving partial values.

6 Conclusions

We have demonstrated an extension to the type system of ML which, using *kinded* type inference, allows record formation and field selection to be implemented as polymorphic operations. This together with a set type allows us to represent sets of records – relations – and a number of operations (union, difference, selection and projection onto a single attribute) of a generalized (non first-normal-form) relational algebra. This has been implemented; in particular a recent technique [Oho92] for compiling field selection into an efficient indexing operation is being combined with the record operations mentioned above in an extension to Standard ML of New Jersey [AM91].

A further extension to this type system using *conditional* type schemes allows us to provide polymorphic projection and natural join operations, giving a complete implementation of a generalized relational algebra. It could be argued that these operations are not important since they are not present in practical relation query languages. Instead a product and single-column projection are usually employed. However a similar type inference scheme can be used in a technique for statically checking the safety of operations on heterogeneous collections, in which each member of a collection of dynamically typed values have some common structure. The approach we have described provides, we believe, a satisfactory account of how relational database programming, and some aspects of object-oriented programming may be brought into the framework of a polymorphically typed programming language, and it may be used as the basis for a number of further investigations into the principles of database programming. We briefly review a few here.

Generalizing relational algebra. The ideas used to provide the generalized relational algebra described in sections 2 and 4 originated in a domain-theoretic description of relations in which each tuple is regarded as a partial description of – or approximation to – a real-world object. Operations of this generalized algebra are provided by considering how a set of tuples may approximate a set of real-world objects. It is debatable whether the whole apparatus of domain theory, used to represent the infinite structures found in the semantics of programs, is needed for the finite structures in databases. A constructive characterization of relational operations is given in [Oho90] using regular trees, using similar notions of approximation but in a domain with simpler underlying properties. It is this characterization that we have used here; in particular it has allowed us to describe recursive values and types.

We believe that this approach to database semantics may bear further fruit, especially in the currently topical study of heterogeneous databases. In providing techniques to combine two or more databases, each database may be thought of as a partial description to the resulting database, and the understanding of how an individual database may approximate the combined database may provide some general-purpose merging techniques.

Abstract Types and Classes. While we have covered some aspects of object-oriented databases, we have not dealt with the most important aspect of classes in object-oriented programming: that of abstraction and code sharing. In [OB89] statically typed polymorphic class declarations are described. The implementation type of a class is normally a record type, whose fields correspond to the instance variables in object-oriented terminology. That methods correctly use the implementation type is done through checking the correctness of field selection, as described in this paper, and the same techniques may be carried into subclasses to check that code is properly inherited from the superclass. For example, one can define a class Person as:

```
class Person = [Name:string, Age:int]
```

```
with
    fun make_person (n,a) = [Name=n, Age=a] : string * int -> Person
    fun name p = p.Name : sub -> string
    fun age p = p.Age : sub -> int
    fun increment_age p = modify(p,Age,p.Age + 1) : sub -> sub
end
```

where sub is a special type variable ranging over the set of all subtypes of Person, which are to be defined later. Inclusion of the sub variable in the type of methods name, age, and increment_age reflects the user's intention being that these methods should be inherited by the subtypes of Person. From this, the extended type system infers the following typing for each method defined in this class.

```
class Person with
    make_person : string * int -> Person
    name : ('a < Person) -> string
    age : ('a < Person) -> int
    increment_age : ('a < Person) -> ('a < Person)
```

The notation ('a < Person) is another form of a kinded type variable whose instances are restricted to the set of subtypes of Person. This can be regarded as an integration of the idea of bounded type abstraction introduced in [CW85] and data abstraction. As in an object-oriented programming language, one can define a subclasses of Person as:

```
class Employee = [Name:string, Age:int, Salary:int] isa Person
with
    fun make_employee (n,a) = [Name=n, Age=a, Salary=0] : string * int -> Employee
    fun salary e = e.Salary : sub -> int
    fun add_salary (e,s) = modify(e,Salary,e.Salary + s) : sub * int -> sub
end
```

By the declaration of isa Person, this class inherits methods name, age, increment_age from Person. The prototype implementation of Machiavelli prints the following type information for this subclass definition.

```
class Employee isa Person with
    make_employee : string * int -> Employee
    add_salary : ('a < Employee) * int -> ('a < Employee)
    salary : ('a < Employee) -> int
inherited methods:
    name : ('a < Person) -> string
    age : ('a < Person) -> int
    increment_age : ('a < Person) -> ('a < Person)
```

The type system can statically check the type consistency of methods that are inherited. It is also possible to define classes that are subclasses of more than one classes, such as ResearchFellow below.

```
class Student = [Name:string, Age:int, Grade:real] isa Person
with
    fun make_student (n,a) = [Name=n, Age=a, Grade=0.0] : string * int -> Employee
    fun grade s = s.Grade : sub -> real
```

```
    fun set_grade (s,g) = modify(s,Salary,g) : sub * real -> sub
end

class ResearchFellow = [Name:string, Age:int, Salary:int, Grade:real]
isa {Employee, Student} with
    fun make_RF (n,a) = [Name=n, Age=a, Grade=0.0, Salary = 0] : string * int -> ResearchFellow
end
```

Classes can be parameterized by types and the type inference system we have described can be extended to programs involving classes and subclass definitions.

One possible addition to this idea is the treatment of object identity. Throughout this paper we have held to the view that object identity, as a programming construct, is nothing more than reference, and that object creation and update are satisfactorily described by the operations on references given in ML and a number of other programming languages. However Abiteboul and Bonner [AB91] have given a catalog of operations on objects and classes, not all of which can be described by means of this simple approach to object identity. Some of the operations appear to call for the passing of reference through an abstraction. For example one may think of Person object identities as references to instances of a Person class and Employee object identitites as references to instances of a Employee class. But this approach precludes the possibility that some of the Person and Student identities may be the same, in fact the latter may be a subset of the former. The ability to ask whether two abstractions are both "views" of the same underlying object appears to call for the ability to pass a reference through an abstraction. If this can be done, we believe it is possible to implement most, if not all, the operations suggested by Abiteboul and Bonner.

Sets and other collection types. Our original description of Machiavelli [OBBT89] attracted some attention [IPS91] because of the use of hom as the basic operation for compuation on sets. The reason for using hom was simply to have a small, but adequate collection of operations on sets on which to base our type system. For the purpose of type inference or type checking, the fewer primitive functions the better. In our development, record types and set types are almost independent; there are only a few primitive operations that involve both, and these occur in sections 4 and 5. For other purposes we could equally well have used record types in conjunction with lists, bags or some other collection type. In fact the use of lists, bags and sets is common in object-oriented programming, and some object-oriented databases [Obj91] supply all three as primitive types.

The study of the commonality between these various collection types is a fruitful extension to the ideas provided here. It may provide us with better ways of structuring syntax [Wad90], with an understanding of the commonality between collection types [WT91], and a more general approach to query languages and optimization for these types [BTBW91].

7 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 thank the referees for their careful reading; we are also grateful for 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.

[AB91] S. Abiteboul and A. Bonner. Objects and views. In *Proceeding of ACM SIGMOD Conference*, pages 238–247, 1991.

[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.

[ACPP91] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.

[AM91] A. W. Appel and D. B. MacQueen. Standard ml of new jersey. In *Proceedings of Third International Symposium on Programming Languages and Logic Programming*, pages 1–13, 1991.

[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.

[BJO91] P. Buneman, A. Jung, and A. Ohori. Using powerdomains to generalize relational databases. *Theoretical Computer Science*, 91(1):23–56, 1991.

[BO91] P. Buneman and A. Ohori. A Type System that Reconciles Classes and Extents. In *Proc. 3rd International Workshop on Database Programming Languages*, pages 191–202, Nafplion, Greece, August 1991. Morgan Kaufmann Publishers.

[BTBN91] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language In *Proc. 3rd International Workshop on Database Programming Languages*, pages 9–19, Nafplion, Greece, August 1991. Morgan Kaufmann Publishers.

[BTBW91] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally Embedded Query Languages In *Proc. International Conference on Database Theory*, Berlin, October 1992. Springer LNCS.

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

[BTS91] V. Breazu-Tannen and R. Subrahmanyam. Logical and Computational Aspects of Programming with Sets/Bags/Lists, Proceedings of the 18th International Colloquium on Automata, Languages, and Programming, Madrid (Spain), July 1991, Springer LNCS 510, pp. 60–75.

[Car86] L. Cardelli. Amber. In *Combinators and Functional Programming, Lecture Notes in Computer Science 242*, pages 21–47. Springer-Verlag, 1986.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. (Special issue devoted to Symposium 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 *International conference on Very Large Data Bases*, August 1986.

[CM84] G. Copeland and D. Maier. Making smalltalk a database system. In *Proceedings of the ACM SIGMOD conference*, pages 316–325. ACM, June 1984.

[CM89] L. Cardelli and J. Mitchell. Operations on records. In *Proceedings of Mathematical Foundation of Programming Semantics, Lecture Notes in Computer Science 442*, pages 22–52, 1989.

[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 Surveys*, 17(4):471–522, December 1985.

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

[Gir71] J.-Y. Girard. Une extension de l'interpretation de gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et théorie des types. In *Second Scandinavian Logic Symposium*. North-Holland, 1971.

[GS89] J. Gallier and W. Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, 67(2):203–260, 1989.

[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.

[HP91] R. Harper and B. Pierce. A record calculus based on symmetric concatenation. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1991.

[HPJW+92] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Perterson. Report on programming language Haskel a non-strict, purely functional language version 1.2. *SIGPLAN Notices, Haskel special issue*, 27(5), 1992.

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

[IPS91] Neil Immerman, Sushant Patnaik, and David Stemple. The Expressiveness of a Family of Finite Set Languages. In *Proceedings of 10th ACM Symposium on Principles of Database Systems*, pages 37–52, 1991.

[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.

[Kap81] S.J. Kaplan Appropriate responses to inappropriate questions. *Elements of Discourse Understanding* (A.K. Joshi, B.L. Webber and I. Sag, eds.) Cambridge 1981.

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

[Mac88] D. MacQueen. References and weak polymorphism. 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.

[Mit90] J.C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter chapter 8, pages 365–458. MIT Press/Elsevier, 1990.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[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.

[OBBT89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database Programming in Machiavelli: a Polymorphic Language with Static Type Inference. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 46–57, Portland, Oregon, June 1989.

[Obj91] ObjectStore Reference Manual. Object Design Inc., Burlington, MA. !991.

[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.

[Oho89a] A. Ohori. A simple semantics for ML polymorphism. In *Proceedings of ACM/IFIP Conference on Functional Programming Languages and Computer Architecture*, pages 281–292, London, England, September 1989.

[Oho89b] A. Ohori. *A Study of Types, Semantics and Languages for Databases and Object-oriented Programming.* PhD thesis, University of Pennsylvania, 1989.

[Oho90] A. Ohori. Semantics of types for database objects. *Theoretical Computer Science*, 76:53–91, 1990.

[Oho92] A Ohori. A compilation method for ML-style polymorphic record calculi. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 154–165, 1992.

[Rem89] D. Remy. Typechecking records and variants in a natural extension of ML. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 242–249, 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*, 5(2), 1977.

[Str87] B. Stroustrup. The C++ programming language. *Addison-Wesley*, 1987.

[SR86] M. Stonebraker and L. Rowe, The Design of Postgres In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, Washington, DC, May 1986.

[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.

[Wad90] P. Wadler. Comprehending Monads ACM Conference on Lisp and Functional Programming, Nice, June 1991.

[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 Symposium on Logic in Computer Science*, pages 92–97, 1989.

[WT91] David A. Watt and Phil Trinder. Towards a Theory of Bulk Types. Fide Technical Report 91/26, Glasgow University, Glasgow G12 8QQ, Scotland, July 1991.

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