# A Type System that Reconciles Classes and Extents[†]

Peter Buneman[‡]
Computer and Information Science
University of Pennsylvania
200 South 33rd Street
Philadelphia, PA 19104-6389, USA
peter@cis.upenn.edu

Atsushi Ohori[§]
Kansai Laboratory, OKI Electric Industry
Crystal Tower, 1-2-27 Shiromi
Chuo-ku, Osaka 540
Japan
ohori@okilab.oki.co.jp

**Abstract**

We present a type system that naturally couples two different, and apparently contradictory, notions of inheritance that occur in object-oriented databases. To do this we distinguish between the *type* and a *kind* of a value. A type describes the entire structure of a value, while a kind describes only the availability of certain fields or methods. This distinction allows us to manipulate heterogeneous collections (collections of values with differing types) in a statically type-checked language. Moreover, the type system is polymorphic and types may be inferred using an extension of the technique used in ML. This means that it is easy to express general-purpose operations for the manipulation of heterogeneous collections. We believe that this system not only provides a natural approach to static type-checking in object-oriented databases; it also offers a technique for dealing with external databases in a statically typed language.

## 1 Introduction

The term "inheritance" is used in (at least) two radically different ways in object-oriented databases. In object-oriented languages the term describes code sharing: by an assertion that *Employee* inherits from *Person* in an object-oriented language 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 has been noted in [WZ89] and appears both in languages with a subsumption rule [Car88a] and in statically type-checked object-oriented languages such as C++ [Str86] 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 re-cast the types of objects on the basis of information that he must make available.

[†]Appeared in **Proc. of 3rd International Workshop on Database Programming Languages, Nafplion, Greece, Morgan-Kaufmann Publishers, pp. 191–202, 1991.**

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

[§]Most of the work presented here was done while the second author was supported by a UK Royal Society Research Fellowship at the University of Glasgow.

We present a solution to this problem that allows us to couple the two notions of inheritance by using heterogeneous collections. At the same time we shall preserve all the desirable properties of a static type system. The solution is developed by formalizing two ideas that are needed in object-oriented databases:

- the use of a collection type such as a set or list, and
- the notion of a "partially specified" object.

The second of these has been suggested by Cardelli [Car88b] in the context of a somewhat different type system. We shall formulate these ideas in an extension to a polymorphic type system that the authors have already developed for database programming [OB88]. In particular we preserve the ability to have type-checked polymorphic code and to retain ML-style type inference, which relieves the programmer of the need for "unnecessary" type declarations – something highly desirable in query languages – by finding the most general type of a program.

The way we achieve this is to incorporate into our type system a distinction between the *type* of an object and its *kind*. In object-oriented terminology the former specifies the class of an object and hence its exact structure, while the latter specifies that certain methods are available. To describe a concrete notation, we shall start by using record types. An assertion of the form $e$`:[Name:string, Age:num]` means that $e$ has just a `Name` and `Age` field and no others; this describes the complete type of $e$. An assertion of the form $e$:$\mathcal{P}$`(<Name:string, Age:num>)` means that at least `Name` and `Age` fields are available on $e$. `<Name:string, Age:num>` describes a *kind*, which we can think of as a set of types – the set of record types that contain `Name:string` and `Age:num` components. The distinction between a type and a kind is that a type specifies the exact structure of a value while a kind indicates that further structure can be revealed by the appropriate operations; i.e. the actual value of $e$ has only been partially specified. In our development it will be convenient to treat kinds, syntactically, as distinguished types, and we use the operator $\mathcal{P}$, which allows us to mix (complete) types and kinds in the same syntax. For example $e$:$\mathcal{P}$`(`$\kappa$`)` means that $e$ has kind $\kappa$, and $e'$:$\{\mathcal{P}$`(`$\kappa$`)`$\}$ means that $e'$ denotes a set of objects each with kind $\kappa$.

Kinds are most useful in conjunction with heterogeneous collections, which may not have a uniform type, but may have a useful kind. Thus an assertion of the form $e$:$\{\mathcal{P}$`(<Name:string, Age:num>)`$\}$ means that $e$ is a set of records, each of which has at least a `Name` and `Age` field, and therefore relational queries involving only selection of these fields are legitimate. To show the use of this, let us assume that the following names have been given for kinds:

```
PersKind   for  <Name:string, Address:string>
EmpKind    for  <Name:string, Address:string, Sal:num>
CustKind   for  <Name:string, Address:string, Balance:num>
```

Also suppose that `DB` is a set of type $\{\mathcal{P}$`(any)`$\}$. The meaning of `any` is the set of all possible types, so that we initially have no information about the structure of members of this set. The following examples should give some idea of the use of heterogeneous sets. They are cast in an extension of the syntax of Machiavelli, described in [OBBT89], which in turn is based on that of ML[HMT88].

1. An operation **filter** $\kappa$ `(`$S$`)` can be defined, which selects all the elements of $S$ which have the fields specified by $\kappa$ and makes those fields available, i.e. **filter** $\kappa$ `(`$S$`)`:$\{\mathcal{P}$`(k)`$\}$. We may use this in a query such as

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

   The SQL-like syntax is that used in [OBBT89], and could be sugared to achieve closer similarity with SQL. The result of this query is a set of (complete) records, i.e. a relation. There is some similarity with the `*` operator of Postgres [SR87], however we may use **filter** on arbitrary kinds and heterogeneous sets; we are not confined to the extensionally defined relations in the database.

2. Since kinds denote sets of types, they can be ordered by set inclusion. In particular, `Empkind` is a "sub-kind" of `PersKind`. From this, the inclusion **filter** `EmpKind (`$S$`)` $\subseteq$ **filter** `PersKind (`$S$`)` will always hold for any heterogeneous set $S$. Moreover this inclusion can be tested in the language. Thus the "data model" (inclusion) inheritance is *derived* from an ordering on kinds 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 <- filter EmpKind (DB)
                       where x.Sal > 30,000
```

   The type inference allow the application of this function to any heterogeneous set whose members each have kind `<Balance:'a>` where `'a` is a type variable. The result is a uniformly typed set, i.e. a set of type {`[Name: string, Balance:'a]`}. Thus the application `RichCustomers(`**filter** `CustKind (DB))` is valid, but the application `RichCustomers(`**filter** `EmpKind (DB))` will not be allowed.

4. From the ordering on kinds it is possible to obtain a meet and join operation (the definitions will be given later.) The unions and intersections of heterogeneous sets have, respectively, the join and meet of their kinds. For example, the following typings are inferred.

```
union(filter CustKind (DB), filter EmpKind (DB))
   :  {P(<Name:string, Address:string>)}

intersection(filter CustKind (DB), filter EmpKind (DB))
    :  {P(<Name:string, Address:string, Sal:num, Balance:num>)}.
```

   (`intersection` is definable in the language)

In the following sections we shall describe the basic operations for dealing with sets and partial values. We shall then develop a polymorphic type system in two steps; first by giving a base calculus for expressions and types from which we may establish the soundness of the type system, and then developing a type inference algorithm for the base calculus.

# 2 Heterogeneous Sets

As far as a type system is concerned, there is little difference among the various collection types one might consider – lists, sets, bags etc; the typing rules are very similar. The only important issue is whether we build such collections over a type with equality, which is needed for operations like intersection. For database work this is important, and we shall therefore take sets over types with equality as our prime example of a collection type.

The basic operations for sets have been described in [OBBT89]. We briefly review them here:

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

**hom** is the "pump" operator in [BBKV88] and is similar to the "fold" or "reduce" operators of many functional languages. The meaning of **hom** is expressed by the equation

**hom**($f$,$op$,$z$,$\{\}$) $= z$
**hom**($f$,$op$,$z$,$\{x\}$) $= f(x)$
**hom**($f$,$op$,$z$,$s$) $=$ op(**hom**($f$,$op$,$z$,$s_1$), **hom**($f$,$op$,$z$,$s_2$)) where $s_1 \cup s_2 = s$ and $s_1 \cap s_2 = \phi$.

This operation only has a well-defined result if $op$ is an associative and commutative operation with identity $z$. For example, the sum of the set `{1,2,3,4}` can be expressed as **hom**`(id, add, 0, {1,2,3,4})`, and its cardinality as **hom**`(`**fn** `x => 1, add, 0, {1,2,3,4})`. Here `id` is the identity function, **fn** `x => 1` is the function that returns `1` on any argument, `add` is two-argument addition, and `{1,2,3,4}` is syntactic sugar for a set built using the operations **union** and {`.`}. It was noted in [OBBT89] that a large number of set constructing forms could be built using **hom**. In particular, **select** ... **from** ... **where** ... is a syntactic shorthand for more primitive functions that may be constructed using hom.

A restricted form of **hom** that we shall make use of in this paper is `homu`, defined as

```
fun homu(f,s) = hom(union, f, {}, s)
```

which is well defined since **union** and {} have the appropriate properties. Examples of its use are:

```
fun map(f,s) = homu(fn x => {f(x)}, s)
fun extract(p,s) = homu(fn x => if p(x) then {x} else {}, s)
fun flatten(s) = homu(fn x => x, s)
```

thus `map(even,{1,2,4})` yields {`true,false`}, `extract(even,{1,2,4})` yields {`2,4`}, and `flatten({{2}, {2,3}, {1,4,7}})` yields {`1,2,3,4,7`}.

While applications of `homu` are always well-defined, it is up to the programmer to check that applications of the more general **hom** satisfy the appropriate conditions. A proper understanding how this may be done, of the relationships between collection types or, indeed, the principles of programming with them requires further investigation (see [BS91, BBN91].)

We are going to combine set operations with operations on *partial values*. These are related to *dynamic* values, which were originally introduced by Cardelli [Car86] as a means for dealing with persistent values in a statically typed language. A value may be packaged together with its type into a value of type `Dynamic`. Such values may later be coerced back into values of some specific type - an operation that may fail if the type component of the dynamic value fails to match that specified in the coercion. Dynamic values are "all-or-nothing": creating a dynamic value effectively hides all of its structure, which can only be recovered by coercing it back to its full type.

Coercion — as with other operations we shall need — is an operation that may fail, and how we represent this failure in a programming language is a matter of taste. Possibilities include the introduction of an exception handling mechanism or the use of an exhaustive *typcase* construct. In order not to complicate the language we shall exploit the available bulk type and make such an operation return a singleton set if it succeeds and an empty set if it fails. For example, we shall shortly introduce a coercion operation **as** $\kappa$ `(`$e$`)` which coerces a partial value $e$ to kind $\kappa$ if it has the appropriate type. If it can be so coerced to a value $v$, this expression yields $\{v\}$, otherwise it yields {}. One of the advantages of this approach is that it provides a compact representation of many programs that return sets (see, for example, **filter** `below`.)

Our partial values can be thought of as dynamic values with some of their structure exposed. Thus when we have a partial type assertion of the form $e$`:`$\mathcal{P}$`(<Name:string, Address:string>)` we not only mean that $e$ has `Name` and `Address` fields (i.e. that $e$`.Name` and $e$`.Address` are legitimate expressions of type `string`); we also mean that there are further operations that may be applied to $e$ to expose more of the value. There are four such operations of which the last three may fail. For these we adopt our strategy of returning singleton or empty sets.:

1. **dynamic**`(`$e$`)`. This is used to construct a dynamic value and has type $\mathcal{P}$`(<`$\tau$`>)` where `<`$\tau$`>` is the singleton kind and $\tau$ is the type of $e$. A heterogeneous set could be constructed by

   {**dynamic**`([Name = "Joe", Age = 10])`, **dynamic**`([Name = "Jane", Balance = 109.54])`}

   This has type {$\mathcal{P}$`(<Name:string>)`}, which is the meet of {$\mathcal{P}$`(<[Name:string, Age:num>)`} and {$\mathcal{P}$`(<[Name:string, Balance:num]>)`}. Typically a "foreign" database might contain such sets about which we have no initial type information.

2. **as** $\kappa$ `(`$e$`)` which, for any kind $\kappa$ "exposes" the properties of $e$ specified by the kind $\kappa$. 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** `<Name:string>` `(`**dynamic**`([Name = "Joe", Balance = 43.21]))`, $e$ will have partial type {$\mathcal{P}$`(<Name: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 `homu` we are now in a position to construct the **filter** operation, mentioned in the introduction, which ties the inclusion of extents to the ordering on kinds. Because we do not have type parameters, it cannot be defined in the language. However it can be treated as a syntactic abbreviation:

   **filter** $\kappa$ `(`$S$`)` $\equiv$ `homu(fn x => as` $\kappa$ `(x),`$S$`)`

3. **coerce** $\tau$ `(`$e$`)` coerces the partial value denoted by $e$ to a (complete) value of type $\tau$. This will only succeed if the type component of $e$ is $\tau$. 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 = 109.54]))` will yield the empty set while **coerce** `[Name:string, Balance:num] (`**dynamic**`([Name = "Jane", Balance = 109.54]))` will return the set {`[Name = "Jane", Balance = 109.54]`}

4. **fuse**`(`$e_1, e_2$`)` combines the partial values denoted by $e_1$ and $e_2$. This will only succeed if the (complete) values of $e_1$ and $e_2$ are equal. If $e_1$ has kind $\kappa_1$ and $e_2$ has kind $\kappa_2$ then **fuse**`(`$e_1, e_2$`)` will have kind $\kappa_1 \sqcup \kappa_2$. If

   $e_1 =$**dynamic**`([Name = "Jane", Age = 21, Balance = 109.54])`,
   $e_2 =$ **as** `<Name:string>` $e_1$,
   $e_3 =$ **as** `<Age:num>` $e_1$, and
   $e_4 =$ **as** `<Name:string> dynamic([Name = "Jane"])`,

   then **homu**`(`**fuse**`,`**union**`(`$e_2, e_3$`))` will be a singleton set of type $\{\mathcal{P}$`(<Name:string, Age:num>)`$\}$ while **fuse**`(`$e_2, e_4$`)` will return an empty set. Note that **fuse** requires arguments of types on which equality is defined; in some sense it 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.

   **fuse** may be used to define set intersection as in

   ```
   fun fuse1(x,s) = homu(fn y => fuse(x,y), s)
   fun intersection(s1,s2) = homu(fn y => fuse1(y,s2), s1)
   ```

We now give a formal description of the language.

# 3 The Base Calculus

This section defines the base calculus as an extension of the simply typed lambda calculus and its operational semantics. We then establish the soundness of the type system. This calculus requires explicit type specifications and does not support polymorphism. This is an intermediate language in our formal development. The full language is the base calculus together with ML style type inference system we shall develop in the next section, where expressions need no type specifications as in the examples we have shown.

## 3.1 Expressions

The set of expressions of the base calculus is given by the following syntax:

$M$ `::= (`$c$`:`$\tau$`)` | $x$ | **fn** $x$`:`$\tau$ `=>` $M$ | $M$`(`$M$`)` | `[`$l$`=`$M$`,...,`$l$`=`$M$`]` | $M$`.`$l$ | **modify**`(`$M$`,`$l$`,`$M$`)` |
`({}:{`$\tau$`})` | {$M$} | **union**`(`$M$`,`$M$`)` | **hom**`(`$M$`,`$M$`,`$M$`,`$M$`)` |
**dynamic**`(`$M$`:`$\tau$`)` | **fuse**`(`$M$`,`$M$`)` | **as** $\kappa$ `(`$M$`)` | **coerce** $\tau$ `(`$M$`)`

`(`$c$`:`$\tau$`)` stands for typed constants and $M$`(`$M$`)` for function application. **modify**`(`$M_1$`,`$l$`,`$M_2$`)` is field modification (or update) which creates a new record by modifying the $l$ field of the record $M_1$ to $M_2$. In this simply typed calculus, this operation is redundant as it is definable by using field selection and record construction. With the type inference and ML polymorphism we shall develop later, however, this operation has more general polymorphic type than the equivalent expression and plays an important role in defining generic code involving records. We have already explained all the other term constructors that are not part of a standard calculus with records.

$$
\begin{array}{rcl}
\tau & :: & \texttt{<}\tau\texttt{>} \\
\tau & :: & \texttt{any} \\
\sigma & :: & \texttt{eq} \\
\texttt{[}l_1\texttt{:}\tau_1\texttt{,...,}\ l_n\texttt{:}\tau_n\texttt{,...]} & :: & \texttt{<}l_1\texttt{:}\tau_1\texttt{,...,}l_n\texttt{:}\tau_n\texttt{>} \\
\mathcal{P}\texttt{(<[}l_1\texttt{:}\tau_1\texttt{,...,}\ l_n\texttt{:}\tau_n\texttt{,...]>)} & :: & \texttt{<}l_1\texttt{:}\tau_1\texttt{,...,}l_n\texttt{:}\tau_n\texttt{>} \\
\mathcal{P}\texttt{(<}l_1\texttt{:}\tau_1\texttt{,}\ \texttt{...,}l_n\texttt{:}\tau_n\texttt{,...>)} & :: & \texttt{<}l_1\texttt{:}\tau_1\texttt{,...,}l_n\texttt{:}\tau_n\texttt{>} \\
\mathcal{P}\texttt{(<}l_1\texttt{:}\tau_1\texttt{,...,}\ l_n\texttt{:}\tau_n\texttt{,...>}^{eq}\texttt{)} & :: & \texttt{<}l_1\texttt{:}\tau_1\texttt{,...,}l_n\texttt{:}\tau_n\texttt{>} \\
\texttt{[}l_1\texttt{:}\sigma_1\texttt{,...,}\ l_n\texttt{:}\sigma_n\texttt{,...,}l_m\texttt{:}\sigma_m\texttt{]} & :: & \texttt{<}l_1\texttt{:}\sigma_1\texttt{,...,}l_n\texttt{:}\sigma_n\texttt{>}^{eq} \\
\mathcal{P}\texttt{(<[}l_1\texttt{:}\sigma_1\texttt{,...,}\ l_n\texttt{:}\sigma_n\texttt{,...,}l_m\texttt{:}\sigma_m\texttt{]>)} & :: & \texttt{<}l_1\texttt{:}\sigma_1\texttt{,...,}l_n\texttt{:}\sigma_n\texttt{>}^{eq} \\
\mathcal{P}\texttt{(<}l_1\texttt{:}\sigma_1\texttt{,...,}\ l_n\texttt{:}\sigma_n\texttt{,...>}^{eq}\texttt{)} & :: & \texttt{<}l_1\texttt{:}\sigma_1\texttt{,...,}l_n\texttt{:}\sigma_n\texttt{>}^{eq}
\end{array}
$$

Figure 1: The Kinding Relation

## 3.2 Types, Kinds and Typing Rules

The set of types (ranged over by $\tau$) and the set of eqtypes (i.e. types with computable equality, ranged over by $\sigma$) are defined by the following syntax:

$$
\begin{array}{l}
\tau ::= b \mid b^{eq} \mid \texttt{[}l\texttt{:}\tau\texttt{,...,}l\texttt{:}\tau\texttt{]} \mid \{\tau\} \mid \tau \to \tau \mid \mathcal{P}\texttt{(}\kappa\texttt{)} \\
\sigma ::= b^{eq} \mid \texttt{[}l\texttt{:}\sigma\texttt{,...,}l\texttt{:}\sigma\texttt{]} \mid \{\sigma\} \mid \mathcal{P}\texttt{(}\epsilon\texttt{)}
\end{array}
$$

$b$ stands for base types and $b^{eq}$ for those with computable equality. $\kappa$ and $\epsilon$ stands for kinds and eqkinds defined by the syntax:

$$
\begin{array}{l}
\kappa ::= \texttt{<}\tau\texttt{>} \mid \texttt{<}l\texttt{:}\tau\texttt{,...,}l\texttt{:}\tau\texttt{>} \mid \texttt{any} \\
\epsilon ::= \texttt{<}\sigma\texttt{>} \mid \texttt{<}l\texttt{:}\sigma\texttt{,...,}l\texttt{:}\sigma\texttt{>}^{eq} \mid \texttt{eq}
\end{array}
$$

In our formal development, kinds denote sets of types sharing common operations. $\texttt{<}\tau\texttt{>}$ is the kind of the singleton set of type $\tau$. `any` is the kind of all types and `eq` is the kind of all eqtypes. We write $\tau :: \kappa$ if the type $\tau$ has the kind $\kappa$. Figure 1 defines this relation. Kinds are used to define typing rules for elimination operations for dynamic values. It also plays an important role in developing a type inference algorithm in section 4.

In order to assign types to **union** and **fuse** in such a way that the type system achieves the desired coupling of classes and extents we have advocated, we define the following ordering on partial types induced by their kind structures. For simplicity in this definition, we assume that the labels in any record type or record kind occur in some standard order.

$$
\begin{array}{rcl}
\mathcal{P}\texttt{(any)} & \leq & \mathcal{P}\texttt{(}\kappa\texttt{)} \\
\mathcal{P}\texttt{(eq)} & \leq & \mathcal{P}\texttt{(}\epsilon\texttt{)} \\
\mathcal{P}\texttt{(<}l_1\texttt{:}\tau_1\texttt{,...,}l_n\texttt{:}\tau_n\texttt{>)} & \leq & \mathcal{P}\texttt{(<[}l_1\texttt{:}\tau'_1\texttt{,...,}\ l\texttt{:}\tau'_n\texttt{,...]>)} \text{ if } \tau_i \leq \tau'_i\ (1 \leq i \leq n) \\
\mathcal{P}\texttt{(<}l_1\texttt{:}\tau_1\texttt{,...,}l_n\texttt{:}\tau_n\texttt{>)} & \leq & \mathcal{P}\texttt{(<}l_1\texttt{:}\tau'_1\texttt{,...,}\ l\texttt{:}\tau'_n\texttt{,...>)} \text{ if } \tau_i \leq \tau'_i\ (1 \leq i \leq n) \\
\mathcal{P}\texttt{(<}l_1\texttt{:}\tau_1\texttt{,...,}l_n\texttt{:}\tau_n\texttt{>)} & \leq & \mathcal{P}\texttt{(<}l_1\texttt{:}\sigma'_1\texttt{,...,}\ l\texttt{:}\sigma'_n\texttt{,...>}^{eq}\texttt{)} \text{ if } \tau_i \leq \sigma'_i\ (1 \leq i \leq n) \\
\mathcal{P}\texttt{(<}l_1\texttt{:}\sigma_1\texttt{,...,}l_n\texttt{:}\sigma_n\texttt{>}^{eq}\texttt{)} & \leq & \mathcal{P}\texttt{(<[}l_1\texttt{:}\sigma'_1\texttt{,...,}\ l\texttt{:}\sigma'_n\texttt{,...,}l_m\texttt{:}\sigma'_m\texttt{]>)} \text{ if } \sigma_i \leq \sigma'_i\ (1 \leq i \leq n) \\
\mathcal{P}\texttt{(<}l_1\texttt{:}\sigma_1\texttt{,}\ \texttt{...,}l_n\texttt{:}\sigma_n\texttt{>}^{eq}\texttt{)} & \leq & \mathcal{P}\texttt{(<}l_1\texttt{:}\sigma'_1\texttt{,...,}\ l\texttt{:}\sigma'_n\texttt{,...>}^{eq}\texttt{)} \text{ if } \sigma_i \leq \sigma'_i\ (1 \leq i \leq n)
\end{array}
$$

The ordering on types is the smallest reflexive relation containing the above relation. This ordering corresponds to the set inclusion ordering on kinds. The ordering has the following property:

**Proposition 1** *$\leq$ is a partial order with pairwise bounded joins and meets.* ∎

Using these definitions, we can now define the type system as a proof system for *typings* of the form $\mathcal{A} \rhd M : \tau$ where $\mathcal{A}$ is a function from a finite set of variables to types, called a *type assignment*. Figure 2 shows some of the typing rules. The partiality of partial types is increased by the rule (union), decreased by the rule (fuse) and changed by the rule (as). Note that the ordering on partial types is used to control the change of the partiality in rules (fuse) and (union). (The complete set of typing and reduction rules will be given in a full paper.)

The calculus has a static type-checking algorithm. This follows from the following:

(record) $$\frac{\mathcal{A}\triangleright M_i : \tau_i \ \ (1 \le i \le n)}{\mathcal{A}\triangleright \texttt{[}l_1\texttt{=}M_1\texttt{,...,}l_n\texttt{=}M_n\texttt{]} : \texttt{[}l_1\texttt{:}\tau_1\texttt{,...,}l_n\texttt{:}\tau_n\texttt{]}}$$

(dot) $$\frac{\mathcal{A}\triangleright M : \tau_1}{\mathcal{A}\triangleright M.l : \tau_2} \qquad \texttt{if } \tau_1 \texttt{::<}l\texttt{:}\tau_2\texttt{>}$$

(union) $$\frac{\mathcal{A}\triangleright M_1 : \{\tau_1\} \qquad \mathcal{A}\triangleright M_2 : \{\tau_2\}}{\mathcal{A}\triangleright \textbf{union}\texttt{(}M_1\texttt{,}M_2\texttt{)} : \{\tau_3\}} \qquad \text{if } \tau_3 = \tau_1 \sqcap \tau_2$$

(fuse) $$\frac{\mathcal{A}\triangleright M_1 : \sigma_1 \qquad \mathcal{A}\triangleright M_2 : \sigma_2}{\mathcal{A}\triangleright \textbf{fuse}\texttt{(}M_1\texttt{,}M_2\texttt{)} : \{\sigma_3\}} \qquad \text{if } \sigma_3 = \sigma_1 \sqcup \sigma_2$$

(dynamic) $$\frac{\mathcal{A}\triangleright M : \tau}{\mathcal{A}\triangleright \textbf{dynamic}\texttt{(}M\texttt{)} : \mathcal{P}\texttt{(<}\tau\texttt{>)}}$$

(as) $$\frac{\mathcal{A}\triangleright M : \mathcal{P}\texttt{(}\kappa\texttt{)}}{\mathcal{A}\triangleright \texttt{as(}M\texttt{:}\mathcal{P}\texttt{(}\kappa'\texttt{))} : \{\mathcal{P}\texttt{(}\kappa'\texttt{)}\}}$$

Figure 2: Some of the Typing Rules for the Base Calculus

**Proposition 2** *For any expression $M$ and $\mathcal{A}$, there is at most one $\tau$ such that $\mathcal{A}\triangleright M : \tau$. Moreover, there is an algorithm which, given $\mathcal{A}$ and $M$, computes the unique $\tau$ such that $\mathcal{A}\triangleright M : \tau$ if one exists; otherwise it reports failure.*

This is proved by proving the stronger statement that any term has at most one typing derivation. The proof is by induction on the number of applications of typing rules. ▌

## 3.3 Operational Semantics

Following [Tof88], we give an operational semantics by defining a set of rules, which reduce closed terms to *canonical values*. Let $E$ stands for *variable environments* which map finite set of variables to canonical values. We write $E\{x \mapsto v\}$ for the function $E'$ such that $dom(E') = dom(E) \cup \{x\}$, $E'(x) = v$ and $E'(y) = E(x)$ for any $y \neq x$. The set of canonical values for our language (ranged over by $v$) is defined as follows:

$$v \ \texttt{::=} \ \texttt{(}c\texttt{:}b\texttt{)} \mid \textbf{closure}\texttt{(}E\texttt{,}x\texttt{,}M\texttt{)} \mid \texttt{[}l\texttt{=}v\texttt{,...,}l\texttt{=}v\texttt{]} \mid \{v\texttt{,...,}v\} \mid \texttt{(}\tau\texttt{,}v\texttt{)} \mid \texttt{wrong}$$

where **closure**$(E,x,M)$ stands for function closures, and `wrong` is the canonical value of a run time type error. Among the set of canonical values, we need to single out the set of of canonical eqvalues (ranged over by $d$) as:

$$d \ \texttt{::=} \ \texttt{(}c\texttt{:}b^{eq}\texttt{)} \mid \texttt{[}l\texttt{=}d\texttt{,...,}l\texttt{=}d\texttt{]} \mid \{d\texttt{,...,}d\} \mid \texttt{(}\sigma\texttt{,}d\texttt{)}$$

For canonical values of sets, we assume an appropriate equivalence relation and consider them as equivalence classes.

We write $E \vdash M \Longrightarrow v$ to denote that $M$ is reduced to a canonical value $v$ under the variable environment $E$. We define the *extent* $[\![\kappa]\!]$ of a kind $\kappa$ as $\tau \in [\![\kappa]\!]$ iff there is some $\tau'$ such that $\tau \le \tau'$ and $\tau' :: \kappa$. Figure 3 shows some of the reduction rules.

For canonical values, we can define their actual types. We write $\models v : \tau$ if $v$ has type $\tau$. The rules for typings of canonical values are given in figure 4. Note that canonical values have in general multiple typings but `wrong` has none.

We can then show the following theorem.

**Theorem 1 (Soundness of the Type System)** *Let $\emptyset_{\mathcal{A}}, \emptyset_E$ respectively be the empty type assignment and the empty variable environment. For any closed expression $M$, if $\emptyset_{\mathcal{A}}\triangleright M : \tau$ and $\emptyset_E \vdash M \Longrightarrow v$ then $\models v : \tau$.*

We write $E \models \mathcal{A}$ if $dom(E) = dom(\mathcal{A})$ and for all $x \in dom(\mathcal{A})$, $\models E(x) : \mathcal{A}(x)$. The theorem is proved by showing the following stronger property by induction on the structure of expressions.

> For any variable environment $E$ and any typing $\mathcal{A}\triangleright M : \tau$, if $E \models \mathcal{A}$ and $E \vdash M \Longrightarrow v$ then $\models v : \tau$.

$$E \vdash x \Longrightarrow \mathit{if}\ \ x \in \mathit{dom}(E)\ \ \mathit{then}\ \ E(x)\ \ \mathit{else}\ \ \texttt{wrong}$$

$$\frac{E \vdash M_1 \Longrightarrow \textbf{closure}(E', x, M_2),\ \ E \vdash M_3 \Longrightarrow v_1,\ \ E'\{x \mapsto v_1\} \vdash M_2 \Longrightarrow v_2}{E \vdash M_1(M_3) \Longrightarrow v_2}$$

$$\frac{E \vdash M \Longrightarrow [l_1{=}v_1, \ldots, l_n{=}v_n] \ \text{ or } \ (\tau, [l_1{=}v_1, \ldots, l_n{=}v_n])}{E \vdash M.l_i \Longrightarrow v_i (1 \le i \le n)}$$

$$\frac{E \vdash M_1 \Longrightarrow d \qquad E \vdash M_2 \Longrightarrow d}{E \vdash \textbf{fuse}(M_1, M_2) \Longrightarrow \{d\}} \qquad \frac{E \vdash M_1 \Longrightarrow d_1 \qquad E \vdash M_2 \Longrightarrow d_2}{E \vdash \textbf{fuse}(M_1, M_2) \Longrightarrow \{\}}\ (\text{if } d_1 \neq d_2)$$

$$\frac{E \vdash M \Longrightarrow (\tau, v)}{E \vdash \textbf{as}\ \kappa\ (M) \Longrightarrow \{(\tau, v)\}}\ (\text{if } \tau \in [\![\kappa]\!]) \qquad \frac{E \vdash M \Longrightarrow (\tau_1, v)}{E \vdash \textbf{as}\ \kappa\ (M) \Longrightarrow \{\}}\ (\text{if } \tau_1 \notin [\![\kappa]\!])$$

Figure 3: Some of the Reduction Rules

$$\models (c{:}\tau) : \tau$$

$$\models \texttt{fun}(E, x, M) : \tau_1 \to \tau_2 \qquad (\text{if } \forall v.\ \text{if} \models v : \tau_1 \text{ and } E\{x \mapsto v\} \vdash M \Longrightarrow v' \text{ then} \models v' : \tau_2.)$$

$$\frac{\models v_i : \tau_i}{\models [l_1{=}v_1, \ldots, l_n{=}v_n] : [l_1{:}\tau_1, \ldots, l_n{:}\tau_n]}$$

$$\frac{\models v_i : \tau}{\models \{v_1, \ldots, v_n\} : \{\tau\}}$$

$$\frac{\models v : \tau}{\models (\tau, v) : \mathcal{P}(\kappa)} \qquad (\text{if } \ \tau \in [\![\kappa]\!])$$

Figure 4: The Typing Rules for Canonical Values

Since `wrong` has no typing, this theorem implies the following.

**Corollary 1** *For any closed expression $M$, if $\emptyset_{\mathcal{A}} \triangleright M : \tau$ and $\emptyset_E \vdash M \Longrightarrow v$ then $v \neq$* `wrong`.

## 3.4 Programming Examples I: Representing Extent Inclusion

We now show how this calculus achieves the extent inclusion by examples. We use pairs $(M, N)$, product type $\tau_1 * \tau_2$ and $n$-argument function definition of the form **fun** $f\ \ (x, \ldots, x)$ = $\cdots$. They are easily defined using records. Again note that the examples below carry type specifications but in our full language they can be omitted.

All of the functions we have introduced in previous sections are indeed definable functions in the base calculus (with appropriate type specifications). For example, `fuse1` and `intersect` become

```
fun fuse1 (x:σ1,s:{σ2}) = homu(fn (y:σ2)=>fuse(x,y), s)
fun intersect (s1:{σ1},s2:{σ2}) = homu(fn (x:σ1)=>fuse1(x,s2), s1)
```

which have the following typings:

$$\texttt{fuse1} : \sigma_1 * \{\sigma_2\} \to \{\sigma_1 \sqcup \sigma_2\}$$
$$\texttt{intersect} : \{\sigma_1\} * \{\sigma_2\} \to \{\sigma_1 \sqcup \sigma_2\}$$

Recall that `PersKind` is defined as `<Name:string, Address:string>` and `EmpKind` as `<Name:string, Address:string, Sal:num>`. The function

```
fun Person_of(S:P(any)) = filter PersKind (S)
```

is then of type $\{\mathcal{P}(\texttt{any})\} \to \{\mathcal{P}(\texttt{PersKind})\}$ and

```
fun Employee_of(S:P(any)) = filter EmpKind (S)
```

$$
\begin{array}{rcl}
T & :: & \mathtt{U} \\
\mathcal{P}(\kappa) & :: & \mathtt{P} \\
t^{<l_1:T_1,\ldots,l_n:T_n,\ldots>} & :: & \mathtt{<}l_1\mathtt{:}T_1\mathtt{,}\ldots\mathtt{,}l_n\mathtt{:}T_n\mathtt{>} \\
u^{<l_1:T_1,\ldots,l_n:T_n,\ldots>} & :: & \mathtt{<}l_1\mathtt{:}T_1\mathtt{,}\ldots\mathtt{,}l_n\mathtt{:}T_n\mathtt{>} \\
\mathtt{[}l_1\mathtt{:}T_1\mathtt{,}\ldots\mathtt{,}l_n\mathtt{:}T_n\mathtt{,}\ldots\mathtt{]} & :: & \mathtt{<}l_1\mathtt{:}T_1\mathtt{,}\ldots\mathtt{,}l_n\mathtt{:}T_n\mathtt{>} \\
\mathcal{P}\mathtt{(<[}l_1\mathtt{:}\tau_1\mathtt{,}\ \ldots\mathtt{,}l_n\mathtt{:}\tau_n\mathtt{,}\ldots\mathtt{]>)} & :: & \mathtt{<}l_1\mathtt{:}\tau_1\mathtt{,}\ldots\mathtt{,}l_n\mathtt{:}\tau_n\mathtt{>} \\
\mathcal{P}\mathtt{(<}l_1\mathtt{:}\tau_1\mathtt{,}\ \ldots\mathtt{,}l_n\mathtt{:}\tau_n\mathtt{,}\ldots\mathtt{>)} & :: & \mathtt{<}l_1\mathtt{:}\tau_1\mathtt{,}\ldots\mathtt{,}l_n\mathtt{:}\tau_n\mathtt{>} \\
\mathcal{P}\mathtt{(<}l_1\mathtt{:}\sigma_1\mathtt{,}\ \ldots\mathtt{,}l_n\mathtt{:}\sigma_n\mathtt{,}\ldots\mathtt{>}^{eq}\mathtt{)} & :: & \mathtt{<}l_1\mathtt{:}\sigma_1\mathtt{,}\ldots\mathtt{,}l_n\mathtt{:}\sigma_n\mathtt{>}
\end{array}
$$

Figure 5: The Kinding Relation on Type Schemes

is of type $\{\mathcal{P}\mathtt{(any)}\} \rightarrow \{\mathcal{P}\mathtt{(EmpKind)}\}$. By the definition of the operational semantics, it is always the case that if $E \vdash \mathbf{as}\ \kappa\ M \Longrightarrow \{(\tau,v)\}$ then $E \vdash \mathbf{as}\ \kappa'\ M \Longrightarrow \{(\tau,v)\}$ for any $\kappa,\kappa'$ such that $[\![\kappa]\!] \subseteq [\![\kappa']\!]$. Since $[\![\mathtt{EmpKind}]\!] \subseteq [\![\mathtt{PersKind}]\!]$, for any heterogeneous set expression $M$, it follows that $\mathtt{Employee\_of}(M)$ and $\mathtt{Person\_of}(M)$ respectively yield sets $S_1$ and $S_2$ of canonical values such that $S_1 \subseteq S_2$. Thus the inclusion relationship is derived from the relationship on kinds.

So far, we have only captured the inclusion relation of extents in a static type system. In the next section, we will develop ML style type inference system and show that the desired features of method inheritance between those sets are also automatically achieved by ML style polymorphism.

# 4 Type Inference Algorithm

Type inference is a method used to infer the set of all derivable typings for a given untyped term. Let $e$ stand for the set of untyped terms obtained form the terms of the base calculus by erasing all type specifications of the form $\mathbf{fn}\ x\mathtt{:}\tau\ \mathtt{=>}\ M$, $(\{\}\mathtt{:}\{\tau\})$, $\mathbf{dynamic}(M\mathtt{:}\tau)$. We write $\mathcal{A}\triangleright e : \tau$ if there is some $M$ such that $e$ is the type erasure of $M$ and $\mathcal{A}\triangleright M : \tau$. The type inference problem is then stated as the problem to construct a representation of the set $\{(\mathcal{A},\tau)|\mathcal{A}\triangleright e : \tau\}$

To solve this problem, we define *conditional typing schemes* [OB88] to represent sets of typings. This is the refinement of *type schemes* [Hin69, Mil78] with syntactic conditions on substitutions of type variables, which are needed to represent constraints associated with some of our typing rules. Some of the conditions can be integrated directly in type schemes by refining them as *kinded (eq)type schemes* by introducing kind constraints on (eq)type variables [Oho90]. The appropriate set of kinded type schemes (ranged over by $T$) and kinded eqtype schemes (ranged over by $E$) are give as:

$$
\begin{array}{l}
T ::= t^K \mid u^K \mid b \mid b^{eq} \mid \mathtt{[}l\mathtt{:}T\mathtt{,}\ldots\mathtt{,}l\mathtt{:}T\mathtt{]} \mid T \rightarrow T \mid \mathcal{P}\mathtt{(}\kappa\mathtt{)} \\
E ::= u^K \mid b^{eq} \mid \mathtt{[}l\mathtt{:}E\mathtt{,}\ldots\mathtt{,}l\mathtt{:}E\mathtt{]} \mid \mathcal{P}\mathtt{(}\epsilon\mathtt{)}
\end{array}
$$

$t^K, u^K$ are *kinded type variables* and *kinded eqtype variables* respectively ranging over types and eqtypes. $K$ denote *kind schemes* which constrain their instantiation. $\kappa,\epsilon$ are kinds and eqkinds we have defined for the base calculus. The set of kind schemes appropriate for our calculus is:

$$K ::= \mathtt{U} \mid \mathtt{P} \mid \mathtt{<}l\mathtt{:}T\mathtt{,}\ldots\mathtt{,}l\mathtt{:}T\mathtt{>}$$

where $\mathtt{U}$ is the universal kind scheme, $\mathtt{P}$ is the partial kind scheme and $\mathtt{<}l\mathtt{:}\sigma\mathtt{,}\ldots\mathtt{,}l\mathtt{:}\sigma\mathtt{>}$ stands for record kind schemes. The kinding relation is given in figure 5. The notion of substitutions is refined to represent kind constraints. A *kind preserving substitution* $\theta$ is a function from the set of kinded type variables to kinded type schemes such that $\theta(t^K) \neq t^K$ for only finitely many $t^K$, it maps eqtype variables to eqtype schemes, and $\theta(t^K) :: \theta(K)$ for all $t^K$. The notion of unifier and most general unifiers are defined as usual. Robinson's unification algorithm is refined to kinded type schemes:

**Proposition 3** *There is an algorithm $\mathcal{U}$ which computes a most general kind preserving unifier of a given set of pairs (equations) of kinded type schemes if one exists; otherwise it reports failure.* ∎

This is a simple extension of a result shown in [Oho90]. ∎

In order to represent the constraints associated with the rules (union) and (fuse), we have to introduce explicit conditions on substitutions of type variables. The following definitions are straightforward adaptation of the method developed in [OB88]. Define *conditions* as formula of the forms: $T = lub(T,T)$ or $T = glb(T,T)$. We say that a kind preserving substitution $\theta$ *is ground for* $X$ if, for any type variable $t^K$ in $X$, $\theta(t^K)$ is a type. A substitution $\theta$ ground for $c$ *satisfies* a condition $c$, denoted by $\theta \models c$, if

1. $c \equiv T_1 = lub(T_2, T_3)$ and $\theta(T_1) = \theta(T_2) \sqcup \theta(T_3)$, or
2. $c \equiv T_1 = glb(T_2, T_3)$ and $\theta(T_1) = \theta(T_2) \sqcap \theta(T_3)$.

Let $C$ be a set of conditions, $\Sigma$ be an assignment of kinded type schemes to variables. Define $\theta \models C$ iff $\theta \models c$ for all $c \in C$. A *conditional principal typing scheme* is a formula of the form $C, \Sigma \triangleright e : T$ such that $\mathcal{A} \triangleright e : \tau$ iff there is a kind preserving substitution $\theta$ such that $\theta \models C$, $\mathcal{A}(x) = \theta(\Sigma(x))$ for all $x \in dom(\Sigma)$ and $\tau = \theta(T)$. Since the definition of conditional typing schemes and the conditions have similar properties of those in [OB88], the following theorem can be proved similarly.

**Theorem 2** *There is an algorithm $\mathcal{P}$ which, given an untyped term $e$, returns either $(C, \Sigma, T)$ of failure such that if $\mathcal{P}(e) = (C, \Sigma, T)$ then $C, \Sigma \triangleright e : T$ is a principal conditional typing scheme otherwise $e$ has no typing.* ∎

## 4.1 Programming Examples II: Coupling Classes and Extents

We now show examples to demonstrate how we achieve inheritance through ML style polymorphism. Moreover, the introduction of the ordering on partial types, inheritance is naturally coupled with the inclusion of extents we have examined in the previous section.

Suppose we have two set expressions `Employees` and `Students` for which the following typings are inferred:

```
Employees :  {P(<Name:string, Address:string, Sal:num>)}
Students :  {P(<Name:string, Address:string, Advisor:string>)}
```

Let `SupportedStudents = intersect(Employees, Student)` where `intersect` is the function we have defined in section 2. If we follow the same reasoning of section 3.4, we can see that `SupportedStudents` $\subseteq$ `Students` and `SupportedStudents` $\subseteq$ `Employees`, as desired.

Next, we show how method inheritance is achieved. Suppose we define the functions

```
fun advisors S = homu(fn x => x.Advisor, S);
fun add_salary S = homu(fn x=>modify(x, Sal, x.Sal + 500), S);
```

For them, the type system infers the following schemes (we omit the kind tag `U`.)

$\emptyset, \emptyset \triangleright$ `advisor` : $\{t_1^{<Advisor:t_2>}\} \rightarrow \{t_2\}$
$\emptyset, \emptyset \triangleright$ `add_salary` : $\{t_1^{<Sal:num>}\} \rightarrow \{t_1^{<Sal:num>}\}$

By kind preserving instantiation, the type system allows them to be applied respectively to `Students` and `Employees`. We can think of these polymorphic functions as "methods" applicable to `Students` and `Employee` respectively. As we have advocated, we expect both of the two functions to be applicable to the intersection `SupportedStudents`. This is automatically achieved in our type system. The type system infers the following conditional typing for `intersect`:

$\{t_1$ `= lub(`$t_2$`,`$t_3$`)`$\}, \emptyset \triangleright$ `intersect` : $\{t_2\}$`*`$\{t_3\} \rightarrow \{t_1\}$

and from this, it infers the following typing:

```
SupportedStudents :  {P(<Name:string, Address:string, Sal:num, Advisor:string>)}
```

Again, by kind preserving instantiation, the type system allows *both* of the above two functions to be applied to `SupportedStudents`. Moreover the results of these application do not show any loss of type information. For example, the type of `add_salary(SupportedStudents)` is the same as that of `SupportedStudents` as desired.

Thus we have successfully constructed a type system that achieves the desired coupling of inheritance of methods and inclusion of extents. To our knowledge no other statically typed programming language allows this coupling.

# 5 Conclusions and Further Investigation

The combination of partial types with the appropriate "bulk" data type provides a method of dealing with heterogeneous collections in a statically typed language. However these two additions to the type system are almost independent of one another. The only basic operations in which they interact are **as**, **coerce** and **fuse** which return singleton or empty collections. It would be possible to describe partial types without making use of a collection type. This is a convenience that avoids the need to deal with run-time exceptions or a specific sum (variant) type to handle the cases in which these operations fail. The choice is a matter of taste. Another such matter is whether **as** or **coerce** should be restricted to carrying partial values into "more specific" kinds or values.

We have discussed only one example, the set, of a collection type in detail. An almost identical account could be given for heterogeneous lists, and it appears possible to use the same techniques in connection with other bulk types such as bags or ordered sets. What we do not yet understand is what is the general characterization of collection types? Why do partial types appear to fit naturally with certain types and not with others? This seems to call for a rather general insight into programming structures.

The ordering on types we have exploited is trivial except on partial record types. Does it need to be extended to other types? For database and object-oriented programming, the notion of "object identity", which appears to be similar to reference, is often invoked. It seems to be straightforward to extend the ordering to work on reference types. However, it is not clear that we need to extend it, say, to function types. At issue here is whether there is some useful "information" ordering on functions. There are certainly useful extensionally defined functions in databases – hash tables and B-trees for example, but is there a natural *partial* description of such structures as there is for records?

# References


[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.

[BBN91] V. Breazu Tannen, P. Buneman, and S. Naqvi Structural Recursion as a Query Language. This collection.

[BS91] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with lists/bags/sets. In *Proceedings of ICALP*, 1991.

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

[Car88a] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. (Special issue devoted to Symp. on Semantics of Data Types, France, 1984).

[Car88b] L. Cardelli. Typeful Programming. Technical Report 45, Systems Research Center, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, California 94301.

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

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

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

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

[OBBT89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli – a polymorphic language with static type inference. In *Proceedings of the ACM SIGMOD conference*, pages 46–57, Portland, Oregon, May – June 1989.

[Oho90] A. Ohori. Extending ML polymorphism to record structure. Technical Report, University of Glasgow, 1990.

[SR87] M. Stonebraker and L.A. Rowe (eds). "The POSTGRES Papers." Technical Memorandum UCB/ERL M86/85, ERL, College of Engineering, UC Berkeley, June, 1987.

[Str86] B. Stroustrup. *The C++ Programming Language* Addison-Wesley, Reading, Mass, 1986.

[Tof88] M. Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, Department of Computer Science, University of Edinburgh, 1988.

[WZ89] P. Wegner and S.B. Zdonik. Models of Inheritance. In *2nd International Conference on Database Programming Languages*, 1989, pp 248-258 Portland, Oregon. Morgan-Kaufmann