Typing Schemes for Objects with Locality

Florian Matthes*

Dept. of Computer Science
University of Hamburg
Schlüterstraße 70
D-2000 Hamburg 13, FRG

Atsushi Ohori‡

Kansai Laboratory
OKI Electric Industry
Crystal Tower
1-2-27 Chuo-ku
Osaka 540, Japan

Joachim W. Schmidt*

Dept. of Computer Science
University of Hamburg
Schlüterstraße 70
D-2000 Hamburg 13, FRG

Abstract

The crucial characteristic of object-oriented databases is the concept of object identity which allows the direct representation of various kinds of dependencies between objects, for example, sharing and cyclicity. For object stores to become a viable technology for large shared databases, a certain degree of spatial control over object dependencies (or object *locality*) seems to be essential. This paper exploits the power of a static type system to capture and evaluate locality information on objects. First, we represent objects by *references* to complex expressions in a functional language. To control the locality of objects, the space of references is partitioned into a set of subspaces with an explicit *reachability constraint*. Next, we define a type system where the locality of an object is part of its static type specification and the predefined reachability constraint is enforced via a static typing discipline. We conclude by highlighting the impact of locality information on the operational support to be expected by next generation database systems.

1 Introduction and Motivation

Object-oriented data models are based on the notion of *object identity*, i.e. "the ability to distinguish objects from one another regardless of their content, location, or addressability" [?]. They thereby relieve database modelers and programmers from the tedious and error-prone "manual" management of key values and their associated uniqueness and referential integrity constraints as required, for example, in relational databases.

This shift from copy semantics to reference semantics in database systems, together with recent advances in computationally complete database languages reinforces the interest in persistent *object stores* [?], providing long term storage of and shared access to data objects and code fragments with mutual cross references via object identifiers.

*This work was supported in part by the European Commission unter ESPRIT BRA contract # 4092 (FIDE).

‡Most of the work presented here was carried out during the second author's visit to the University of Glasgow supported by a British Royal Society Research Fellowship.

There is, however, an increasing awareness of the *problems of scale* that have to be solved in order to make object stores a viable technology for large scale, multi-user or distributed databases.

A promising approach to the solution of these problems is to partition the logically homogeneous object store into disjoint *repositories* (files [?, ?], pools [?], databases [?, ?]) and to localize objects in these repositories. Despite a great deal of research and implementation efforts in object-oriented databases and persistent programming, very little is known about the properties of such repositories, the localization of objects in repositories and, most importantly, the dependencies between objects induced by the creation and propagation of object identifiers.

The interest of this work is to integrate the advantages of a partitioned store into a data model for complex objects. Furthermore, we would like to achieve this integration within a typed programming language so that the database programmer can also enjoy the benefits of a static type system.

In this paper, we propose an abstract partitioning mechanism of an object store in a type system of a programming language. We model objects through *references* to complex expressions, as they are implemented in Standard ML [?]. In contrast to Standard ML, however, our reference space is partitioned into disjoint repositories. Each repository is associated with a set of repositories representing the *reachability set* of the repository to other repositories. We then develop a static type system to reason about the dependency of a value to repositories (or the *locality* of a value). In the type system, the locality of a value is represented as part of its type and the reachability constraint among repositories is enforced by a static type discipline.

The main goal of this paper is to analyze the problem of object locality control and to explain our solution with emphasis on the rationale behind the type system. The reader is referred to [?] for a complete description of the type system and proofs of various technical properties. The paper is organized as follows. In section ??, we set the stage by defining a core database language with records, collections, references and repositories. Section ?? develops a type system for that language. We introduce first the notion of types and localities and then define type checking rules. The proof of the soundness of these rules with respect to the run-time evaluation of the language is the subject of section ??. Section ?? outlines a method to develop a type inference system for objects with locality. The paper concludes with a discussion of the use of static locality information in database systems and describes possible extensions to our type system.

2 Modeling Objects in a Partitioned Store

As argued in [?], major properties of objects (without locality control) can be represented by references when they are combined with a rich set of value constructors. In a functional language, references can be introduced by the following set of primitives:

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 previously assigned to the reference r. r`:=`v assigns a new value v to a given reference r (discarding the value previously assigned to r). In a database context, these operations correspond respectively to the creation of an object with identity, retrieving the value of an object, and changing the associated value (*attribute* or *state*) of an object without affecting its identity. See [?] for a more detailed discussion of their relevance for database programming and a formal account of these primitives in a purely functional framework.

The following examples (taken from [?]) illustrate the (standard) semantics of references in our model that are formally captured by the reduction rules in section ??. The invocation of **new** such as

```
David = new([Name="David", Age= 25]);
```

will create a reference whose associated value is the record `[Name="David", Age=25]` and can naturally be regarded as an object with identity which has a `Name` and an `Age` attribute. Sharing and mutability are also represented by references. If we define the following two person objects from `David`:

```
Jessica = new([Name="Jessica", Age = 2, Father = David]);
Frederick = new([Name="Frederick", Age = 3, Father = David]);
```

then `Jessica` and `Frederick` *share* the same person `David` as the value of the `Father` field and the expression **eq**`((!Jessica).Father,(!Frederic).Father)` is `true`. An update to the object `David` as

```
David := modify(!David, Age, (!David).Age + 1);
```

is reflected in the `Father` as seen from both `Jessica` and `Frederick`, where **modify**(r,f,v) is the operation that creates a new record form the record r by modifying the field f with the value v. After the above statement, both `(!((!Jessica).Father)).Age` and `(!((!Frederick).Father)).Age` are evaluated to `26`.

By virtue of these properties references capture the central features of objects but they also introduce strong explicit data dependencies among objects. In the above examples, the objects `Jessica` and `Frederick` *depend* on the object `David` and are meaningless unless `David` exists. Furthermore, it is possible to introduce new dependencies by creating copies of the `Father` attribute of `Frederick` or `Jessica`. Additional complications arise if we want to include functions (methods) within objects, since functions induce implicit dependencies to objects through their variable binding mechanism. It should be clear that in the context of large, shared and long lived data structures the control of such dynamic dependencies becomes crucial, e.g. to support optimized data placement and efficient garbage collection.

The challenge is now to introduce a mechanism to partition the space of objects into suitably selected subspaces and to control the generation and propagation of cross-references between these subspaces. For this purpose, we introduce the notion of *repositories* with a predefined dependency constraint. Our primary intuition behind repositories is that they correspond to partitions of a persistent store. Here, we simply assume that they are abstract named entities. It is not hard to add persistence to our system by using the techniques developed in persistent programming languages [?, ?] (reachability-based) or in modular database programming languages [?, ?, ?] (statically declared).

We assume that there is a fixed set $\mathcal{R}$ of repository names (ranged over by R). We require that references are bound at creation-time to a repository as in:

$$\mathbf{new}(R, M)$$

where R is the repository name to which the reference is bound. This binding between a reference and its repository is immutable, i.e. objects possess no mobility. To model the intended constraint on the dependencies among repositories, we associate to each repository R a set $\mu(R)$ of repositories, called the *reachability set* of R. This is to impose the constraint that if a reference in a repository R refers to a value containing other references then they must be in some repositories in the reachability set $\mu(R)$. This restriction on cross-references is deliberately chosen to be "shallow", i.e. there is no explicit restriction on the set of repositories that can be reached transitively via several references. This decision is in the spirit of modular systems where every component should only introduce local constraints on the overall system structure. With this mechanism at hand, we are able to abstract from the multiplicity of dynamic reference creation and propagation operations in a persistent system and to view the database at the granularity of repositories and their reachability sets. Section ?? will elaborate on the impact of this kind of compile-time information on databases with a structured object store.

As a very simple example, suppose we have repositories `Permanent` and `Temporary` with the reachability sets $\mu(\texttt{Permanent}) = \emptyset$ and $\mu(\texttt{Temporary}) = \{\texttt{Permanent}\}$. This imposes the constraint that references in the repository `Temporary` can refer to references in `Permanent` as well as `Temporary` while references in `Permanent` can only refer to references in `Permanent`. Then

```
helen = new(Permanent, [Name = "Helen", Age = 35]);
john = new(Temporary, [Name = "John", Age = 41]);
joe = new(Permanent, [Name = "Joe", Age = 29, Boss = helen]);
susan = new(Temporary, [Name = "Susan", Age = 18, Boss = helen]);
```

are all legal but

```
mary = new(Permanent, [Name = "Joe", Age = 29, Boss = john]);
```

violates the constraint and we expect the type system to detect the violation and to reject this statement.

We integrate these operations for references in a typed functional language with labeled records and lists as defined by the following syntax:

$$\begin{array}{ll} M ::= & (c\!:\!\tau) \mid x \mid \lambda x : \tau . M \mid M(M) \\ & \mid \texttt{[}l\texttt{=}M, \ldots, l\texttt{=}M\texttt{]} \mid M.l \mid \mathbf{modify}(M, l, M) \\ & \mid (\texttt{nil}\!:\!\tau) \mid \mathbf{insert}(M, M) \mid \mathbf{head}(M) \mid \mathbf{tail}(M) \\ & \mid \mathbf{new}(R, M\!:\!\tau) \mid M := M \mid\ !M \end{array}$$

where τ stands for types which will be discussed in the next section. $(c\!:\!\tau)$ stands for typed constants, $\texttt{[}l\texttt{=}M, \ldots, l\texttt{=}M\texttt{]}$ is the syntax for labeled records as we have already seen in the above examples. $M.l$ is field selection from a record, **modify**`(`M_1`,`l`,`M_2`)` is field modification (or field update) as already explained. `(nil:`τ`)` is the empty list of type τ. **head** and **tail** are standard primitives for lists (coresponding, respectively, to *car* and *cdr* in Lisp). The type specification in **new**`(`R`,`$M:\tau$`)` is needed to show the soundness of the type system. Since τ is always the same as the type of M, this can be automatically inserted by the type system and can therefore be safely omitted.

With appropriate syntactic shorthand, this language can serve as a language to define and manipulate objects. We write x_1=M_1;M_2 as a shorthand for $(\lambda x_1:\tau_1.M_2)(M_1)$ where τ is the type of M_1 and write x_1=M_1;$\cdots$;x_n=M_n as a shorthand for x_1=M_1; (x_2=M_2; ($\cdots$; x_n=M_n)$\cdots$). Under a call-by-value evaluation strategy, this provides a mechanism for value bindings and sequential evaluation.

3 Typing Objects with Locality

In a conventional type system, a type represents the structure of a value and typechecking is the process of checking the consistency of operations with respect to the structures of the values involved. Our goal is to extend such a conventional type system to include repository information so that a type also captures the dependency of a value on repositories. The type system can thereby also check the consistency of reference creation and manipulation with respect to reachability constraints among repositories. In this section we will develop such a type system.

3.1 Types and Localities

An expression, in general, contains references through which it depends on a set of repositories. We call this dependency the *locality* of an expression. Formally, a locality π is a set of repositories. The first step in defining the type system is to enrich the language of types to include locality information. Since types in a static type system must be static entities, i.e. they must be denotable at compile-time, we assume that the set of repositories and their reachability sets are known at compile-time. This is certainly a restriction as it prohibits a dynamic reconfiguration of repositories. The relaxation of this restriction is a topic of future interest. We will take this point up later in section ??.

It turns out that the only type constructors that require an explicit locality specification are:

`ref(`π`,`τ`)`	*the reference type constructor, and*
$\tau_1 \xrightarrow{\pi} \tau_2$	*the function type constructor.*

A value of a reference type `ref(`π`,`τ`)` is a reference in one of the repositories of π. τ specifies the type of the referenced value. Note that the repository information is ambiguous unless π is a singleton set. As we shall see later, this ambiguity is viatal in achieving a smooth mixture of lists (or more general collection types) and references.

The locality tag π in $\tau_1 \xrightarrow{\pi} \tau_2$ means that a function may possibly manipulate references in π. To see the need of this explicit locality tag, consider the following functions.

```
f = λx:ref({R},int).λy:int.(!x) + y;
g = f(new(R,2))
```

F is a function that takes a reference `x` to an integer in a repository `R` and returns a function that increments its argument `y` by the dereferenced value `!x`. The expression `g`, therefore, is a function which takes an integer and returns an integer. In a conventional type system it is given the type $\texttt{int} \rightarrow \texttt{int}$. Although neither its domain type nor its range type have a connection to any repository, `g` nevertheless depends on the repository `R` through the static environment where the variable `x` is bound to a reference in R. To completely capture the localities of values involving functions, we must record such dependencies within the type of an expression. Therefore, `g` has type $\texttt{int} \xrightarrow{\{R\}} \texttt{int}$ in our type system.

The meaning of the extra locality tag π in function types can also be illustrated by looking at standard implementations of functional languages, where a run-time value of a function type is a function closure containing an environment and it is this environment that induces the dependency of the value to repositories.

With these refinements, the set of types is given by the following abstract syntax:

$$\tau ::= b \mid \texttt{unit} \mid \tau \xrightarrow{\pi} \tau \mid \texttt{[}l\texttt{:}\tau\texttt{,}\ldots\texttt{,}l\texttt{:}\tau\texttt{]} \mid \{\tau\} \mid \texttt{ref(}\pi\texttt{,}\tau\texttt{)}$$

`unit` is the trivial type whose only value is `Void`. b ranges over base types. $\texttt{[}l\texttt{:}\sigma\texttt{,}\ldots\texttt{,}l\texttt{:}\sigma\texttt{]}$ stands for record types and $\{\tau\}$ for set types.

Types in our system not only represent the structure of values but they also represent their locality. The locality $\mathcal{L}(\tau)$ represented by the type τ is defined inductively as follows:

$$\begin{array}{rcl}
\mathcal{L}(b) & = & \emptyset \\
\mathcal{L}(\texttt{unit}) & = & \emptyset \\
\mathcal{L}(\tau_1 \xrightarrow{\pi} \tau_2) & = & \pi \\
\mathcal{L}(\texttt{[}l_1\texttt{:}\tau_1\texttt{,}\ldots\texttt{,}l_n\texttt{:}\tau_n\texttt{]}) & = & \bigcup\{\mathcal{L}(\tau_1) \mid 1 \leq i \leq n\} \\
\mathcal{L}(\{\tau\}) & = & \mathcal{L}(\tau) \\
\mathcal{L}(\texttt{ref(}\pi\texttt{,}\tau\texttt{)}) & = & \pi
\end{array}$$

Note that the domain type and the range type of a function type do not induce any dependency. The rule for reference types reflects our decision that the reachability constraint specified by the map μ is *shallow*. The locality of the type $\texttt{ref(}\pi\texttt{,}\tau\texttt{)}$ is always π irrespective of the locality of τ. If a *deep* restriction were preferred, then the required rule would be:

$$\mathcal{L}(\texttt{ref(}\pi\texttt{,}\tau\texttt{)}) \quad = \quad \pi \cup \mathcal{L}(\tau)$$

3.2 Typing Rules

We are now in a position to define a proof system for *typing judgements* in such a way that for any well-typed program (closed expression) P, a static typing judgement $P : \tau$ implies the following three desirable properties of the (dynamic) evaluation of P.

1. The structure of the value produced by P conforms to the structural part of τ;
2. The locality of the value produced by P conforms to $\mathcal{L}(\tau)$;
3. The reference environment (the object store) does not violate the predefined reachability constraint imposed by μ.

Since expressions in general contain free variables, a typing judgement is defined with respect to a *type assignment* $\mathcal{A}$ which is a function from a finite subset of variables to types. We write $\mathcal{A}\{x \mapsto \tau\}$ for the function $\mathcal{A}'$ such that $dom(\mathcal{A}') = dom(\mathcal{A}) \cup \{x\}$, $\mathcal{A}'(x) = \tau$, and $\mathcal{A}'(y) = \mathcal{A}(y)$ if $y \neq x$. We write $\mathcal{A} \rhd M : \tau$ if expression M has type τ under type assignment $\mathcal{A}$. In the rest of this section, we analyze each of the expression constructors and define their typing rules.

(const) $\quad \mathcal{A} \rhd (c : \tau) : \tau$

(var) $\quad \mathcal{A} \rhd x : \tau \quad$ if $x \in dom(\mathcal{A}), \mathcal{A}(x) = \tau$

(app) $$\frac{\mathcal{A} \rhd M_1 : \tau_1 \xrightarrow{\pi} \tau_2 \qquad \mathcal{A} \rhd M_2 : \tau_1}{\mathcal{A} \rhd M_1(M_2) : \tau_2}$$

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

(dot) $$\frac{\mathcal{A} \rhd M : \tau_1}{\mathcal{A} \rhd M.l : \tau_2} \quad \text{if } \tau_1 \text{ is a record type with } l : \tau_2$$

(modify) $$\frac{\mathcal{A} \rhd M_1 : \tau_1 \qquad \mathcal{A} \rhd M_2 : \tau_2}{\mathcal{A} \rhd \mathbf{modify}(M_1, l, M_2) : \tau_1} \quad \text{if } \tau_1 \text{ is a record type with } l : \tau_2$$

(empty) $\quad \mathcal{A} \rhd (\texttt{nil}:\{\tau\}) : \{\tau\}$

(head) $$\frac{\mathcal{A} \rhd M : \{\tau\}}{\mathcal{A} \rhd \mathbf{head}(M) : \tau}$$

(tail) $$\frac{\mathcal{A} \rhd M : \{\tau\}}{\mathcal{A} \rhd \mathbf{tail}(M) : \{\tau\}}$$

(deref) $$\frac{\mathcal{A} \rhd M_1 : \texttt{ref}(\pi, \tau)}{\mathcal{A} \rhd \texttt{!}M_1 : \tau}$$

Figure 1: The remaining typing rules

We start with references. For reference creation $\mathbf{new}(R, M)$, we must ensure that the locality of M satisfies the reachability constraint $\mu(R)$. This is expressed in our type system as $\mathcal{L}(\tau) \subseteq \mu(R)$ where τ is the type of M. The desired typing rule for reference creation is then given as:

$$\text{(new)} \qquad \frac{\mathcal{A} \rhd M : \tau}{\mathcal{A} \rhd \mathbf{new}(R, M : \tau) : \mathtt{ref}(\{R\}, \tau)} \quad \mathtt{if}\ \mathcal{L}(\tau) \subseteq \mu(R)$$

For example, in the previous section, we considered the two repositories `Permanent` and `Temporary` with the associated reachability sets $\mu(\mathtt{Permanent}) = \emptyset$ and $\mu(\mathtt{Temporary}) = \{\mathtt{Permanent}\}$ and created references named `helen`, `john`, `joe` and `susan`. All of them are legal and the following typings are derivable:

```
helen:  ref({Permanent}, [Name:string, Age:int])
john:  ref({Temporary}, [Name:string, Age:int])
joe:  ref({Permanent}, [Name:string, Age:int,
          Boss:ref({Permanent}, [Name:string, Age:int])])
susan:  ref({Temporary}, [Name:string, Age:int,
            Boss:ref({Permanent}, [Name:string, Age:int])])
```

The reachability constraint among repositories is enforced by the side condition $\mathcal{L}(\tau) \subseteq \mu(R)$ of the typing rule (new). In the case of `susan`, for example, the condition is

$$\begin{aligned}&\mathcal{L}(\mathtt{[Name{:}string,\ Age{:}int,}\\ &\quad \mathtt{Boss{:}ref(\{Permanent\},\ [Name{:}string,\ Age{:}int])])}\\ &\subseteq\ \mu(\mathtt{Temporary})\ =\ \{\mathtt{Permanent}\}\end{aligned}$$

which is indeed true. The rule for dereferencing is standard (see Fig. ??). We will consider the rule for assignment after considering the interaction between references and lists.

The locality of each of the above reference types is a singleton set representing the exact repository of the reference. More flexible or *ambiguous* reference types become necessary when we want to achieve a smooth interaction between references and lists. In order to make locality information as transparent as possible to those users who do not make any use of it, we would like to allow lists which contain references of different localities as long as their structure is the same. This conflicts with the (strictly homogeneous) list type constructor. In order to typecheck statically the elimination operation **head**, lists must be homogeneous while objects with different localities have different types. We reconcile this conflict by *weakening* the locality information of the element type of a list type. Remember that a value of the type $\mathtt{ref}(\pi,\tau)$ is a reference in *one of* the repositories in π. This ambiguity allows us to assign a type to a list containing elements with different localities. For example, the list {`helen, john`} is given the type {`ref({Permanent, Temporary},[Name:string, Age:int])`}. A value of this type is a list whose elements are references either in `Permanent` or `Temporary`. This idea is generalized to arbitrary complex types by exploiting the following partial ordering induced by the ambiguity of locality:

$$\begin{array}{rcl} b & \ll & b \\ \tau_1 \xrightarrow{\pi} \tau_2 & \ll & \tau_1' \xrightarrow{\pi'} \tau_2' \quad \text{if } \tau_1' \ll \tau_1, \tau_2 \ll \tau_2', \pi \subseteq \pi' \\ [l_1 : \tau_1, \ldots, l_n : \tau_n] & \ll & [l_1 : \tau_1', \ldots, l_n : \tau_n'] \quad \text{if } \tau_i \ll \tau_i' \ (1 \le i \le n) \\ \{\tau\} & \ll & \{\tau'\} \quad \text{if } \tau \ll \tau' \\ \mathtt{ref}(\pi,\tau) & \ll & \mathtt{ref}(\pi',\tau) \quad \text{if } \pi \subseteq \pi' \end{array}$$

Informally, $\tau_1 \ll \tau_2$ means that the locality information of any part of type τ_1 is at least as precise as that of the corresponding part in τ_2. This ordering is used to define the typing rule for list construction:

$$\texttt{(insert)} \qquad \frac{\mathcal{A} \triangleright M_1 : \tau_1 \qquad \mathcal{A} \triangleright M_2 : \{\tau_2\}}{\mathcal{A} \triangleright \mathbf{insert}(M_1, M_2) : \{\tau_3\}} \qquad \text{if } \tau_3 = \tau_1 \sqcup \tau_2$$

In the example of $\{$`helen, john`$\}$, which is shorthand for **insert**`(helen,` **insert**`(john, nil))`, τ_1 =`ref(`$\{$`Permanent`$\}$`, [Name:string, Age:int])` and τ_2 =`ref(`$\{$`Temporary`$\}$`, [Name:string, Age:int])` and therefore we have $\tau_1 \sqcup \tau_1$ =`ref(`$\{$`Permanent, Temporary`$\}$`, [Name:string, Age:int])` as desired. The other rules for list types are again standard.

The rule for assignment is also specified using the ordering on types induced by ambiguous locality information.

$$\texttt{(assign)} \qquad \frac{\mathcal{A} \triangleright M_1 : \texttt{ref}(\pi, \tau_1) \qquad \mathcal{A} \triangleright M_2 : \tau_2}{\mathcal{A} \triangleright M_1 := M_2 : \texttt{unit}} \qquad \text{if } \tau_2 \ll \tau_1$$

The rationale behind this rule is that we can lose locality information in assigning a value to a reference.

Another expression constructor that interacts with locality information is lambda abstraction. To give the rule for lambda abstraction, we need to know the exact condition for a lambda term $\lambda \mathtt{x}{:}\tau.M$ to have a function type $\tau_1 \xrightarrow{\pi} \tau_2$ under a type assignment $\mathcal{A}$. As in any standard functional calculus, the domain type τ_1 should be the type of the lambda variable and the range type τ_2 should be the type of the body M under the type assignment obtained from $\mathcal{A}$ by extending $\mathcal{A}$ with the entry $x : \tau_1$. What should be asserted on the locality π? It should reflect the locality of the run-time value that corresponds to the lambda term. As noted earlier in section ??, π has to describe the dependencies on repositories induced by the function environment that binds a set of variables assumed in $\mathcal{A}$ to the actual parameters supplied through function applications.

We therefore define the locality π of the function type as the union $\{\mathcal{L}(\mathcal{A}(x)) | x \in dom(\mathcal{A})\}$. In the following, we simply write $\mathcal{L}(\mathcal{A})$ for the above union. The typing rule for function abstraction is now defined as:

$$\texttt{(abs)} \qquad \frac{\mathcal{A}\{x \mapsto \tau_1\} \triangleright M : \tau_2}{\mathcal{A} \triangleright \lambda \mathtt{x}{:}\tau_1.M : \tau_1 \xrightarrow{\mathcal{L}(\mathcal{A})} \tau_2}$$

Typing rules for the other expression constructors are standard and are summarized in Fig. ??.

4 Soundness of the Type System

In order for a static type system to be useful, it must be *sound* with respect to the run-time computation of the language. Intuitively, the soundness states that the static type information of an expression represents the actual properties of the value computed at run-time from the expression. In our system, this implies that, as in a conventional type system, a well-typed program will not produce a run time type error but it also guarantees that a well-typed program will not produce a reference environment that violates the predefined reachability constraint. In this section we will explain the method used to establish the soundness of our system and sketch the proof.

One approach to show the soundness of a type system is to model run-time computation by a reduction relation and to show that typing judgements are preserved by the reduction relation. We first define the set of *canonical values* that represent run-time values of our language. For each repository name R, we assume that there is a countably infinite set of *reference atoms.* We further assume that a reference atom in a given repository is identified by an integer. The use of integers is not essential. Any countable set of atomic elements with a linear ordering can be used. We write $r^{\tau}_{R}(i)$ for the reference atom in the repository R identified by the integer i. The set of *canonical values* (ranged over by v) of the language is given by the following syntax:

$$\begin{aligned} v \ ::= & \ (c\mathtt{:}\tau) \mid \mathtt{fun(}E\mathtt{,}x\mathtt{,}M\mathtt{)} \mid \mathtt{[}l\mathtt{=}v\mathtt{,}\ldots\mathtt{,}l\mathtt{=}v\mathtt{]} \mid \{v\mathtt{,}\ldots\mathtt{,}v\} \\ & \mid r^{\tau}_{R}(i) \mid \mathtt{except} \mid \mathtt{wrong} \end{aligned}$$

where `except` is the value to denote a run-time exception (which is needed for partially defined functions like **head** and **tail**), `wrong` represents a run-time type error and $\mathtt{fun(}E\mathtt{,}x\mathtt{,}e\mathtt{)}$ denotes function closures where E is a *variable environment* which maps a finite set of variables to canonical values.

In order to represent mutable references, we define a repository environment (R-environment) as a record $[I = i, S = f]$ where i is an integer and f is a function which maps the set of integers $\{1, \ldots, i-1\}$ to canonical values. This is to model the *contents* of the repository R. The integer i is used to maintain the "next available reference index". An $\mathcal{R}$-environment $\mathcal{E}$ is a set of repository environments indexed by $\mathcal{R}$, i.e. it is a function on $\mathcal{R}$ such that $\mathcal{E}(R)$ is an R-environment. A run time value is then modeled by a pair $(\mathcal{E}, v)$ of an $\mathcal{R}$-environment (modeling the state of the partitioned object store) and a canonical value (maintaining the progress of program execution). Following [?], we present the operational semantics as a reduction system for such pairs under a value environment E. We write $E \vdash (\mathcal{E}, M) \Longrightarrow (\mathcal{E}', v)$ if $(\mathcal{E}, M)$ is reduced to $(\mathcal{E}', v)$ under E. Some of the reduction rules are shown in Fig. ??. The rules for other expressions are similar to those found in [?].

A canonical value v is a value independent of any external environment. We can define the locality $\mathcal{L}^c(v)$ of a canonical value v as:

$$\begin{aligned}
\mathcal{L}^c((c\mathtt{:}\tau)) &= \emptyset \\
\mathcal{L}^c(\mathtt{fun(}E\mathtt{,}x : \tau\mathtt{,}M\mathtt{)}) &= \bigcup\{\mathcal{L}^c(E(x)) | x \in dom(E)\} \\
\mathcal{L}^c(\mathtt{[}l_1\mathtt{=}v_1\mathtt{,}\ldots\mathtt{,}l_n\mathtt{=}v_n\mathtt{]}) &= \bigcup\{\mathcal{L}^c(v_i) | 1 \leq i \leq n\} \\
\mathcal{L}^c(\{v_1\mathtt{,}\ldots\mathtt{,}v_n\}) &= \bigcup\{\mathcal{L}^c(v_i) | 1 \leq i \leq n\} \\
\mathcal{L}^c(r^{\tau}_{R}(i)) &= \{R\} \\
\mathcal{L}^c(\mathtt{except}) &= \emptyset
\end{aligned}$$

In order to reason about the properties of the run-time evaluation we use the following

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

$$E \vdash (\mathcal{E}, \lambda \texttt{x}\!:\!\tau.M) \Longrightarrow (\mathcal{E}, \texttt{fun(}E\texttt{,}x\texttt{,}M\texttt{)})$$

$$\frac{\begin{array}{c} E \vdash (\mathcal{E}_1, M_1) \Longrightarrow (\mathcal{E}_2, \texttt{fun(}E'\texttt{,}x : \tau\texttt{,}M_2\texttt{)}) \\ E \vdash (\mathcal{E}_2, M_3) \Longrightarrow (\mathcal{E}_3, v_1) \\ E'\{x \mapsto v_1\} \vdash (\mathcal{E}_3, M_2) \Longrightarrow (\mathcal{E}_4, v_2) \end{array}}{E \vdash (\mathcal{E}_1, M_1\texttt{(}M_3\texttt{)}) \Longrightarrow (\mathcal{E}_4, v_2)}$$

$$\frac{E \vdash (\mathcal{E}_1, M) \Longrightarrow (\mathcal{E}_2, v)}{\begin{array}{c} E \vdash (\mathcal{E}_1, \mathbf{new}\texttt{(}R\texttt{,}M : \tau\texttt{)}) \Longrightarrow (\\ \mathcal{E}_2\{R \mapsto (\mathcal{E}_2(R).I + 1, (\mathcal{E}_2(R).S)\{\mathcal{E}_2(R).I \mapsto v\})\}, r^\tau_R(\mathcal{E}_2(R).I)) \end{array}}$$

$$\frac{E \vdash (\mathcal{E}_1, M_1) \Longrightarrow (\mathcal{E}_2, r^\tau_R(i)) \qquad\qquad E \vdash (\mathcal{E}_2, M_2) \Longrightarrow (\mathcal{E}_3, v)}{E \vdash (\mathcal{E}_1, M_1 \texttt{:=} M_2) \Longrightarrow (\mathcal{E}_3\{R \mapsto (\mathcal{E}_3(R).I, \mathcal{E}_3(R).S\{i \mapsto v\})\}, \texttt{Void})}$$

Figure 2: Some of the reduction rules

notation: We write $\models v : \tau$ for a typing of a canonical value v. A variable environment E *respects* a type assignment $\mathcal{A}$, denoted by $E \models \mathcal{A}$, if $\mathit{dom}(E) = \mathit{dom}(\mathcal{A})$ and for all $x \in \mathit{dom}(\mathcal{A})$, $\models E(x) : \mathcal{A}(x)$. A reference environment $\mathcal{E}$ is a *model* of μ, written $\mathcal{E} \models \mu$, if for all $R \in \mathcal{R}$ and for for all $r^\tau_R(i) \in \mathit{dom}(\mathcal{E}(R).S)$, $\models \mathcal{E}(R).S(i) : \tau$. The rules for typings of canonical values are then given as:

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

$$\models r^\tau_R(i) : \texttt{ref(}\pi\texttt{,}\tau\texttt{)} \ \text{ for all integer } \ i \ \text{ and } R \in \pi$$

$$\models \texttt{except} : \tau \quad \text{for all } \tau$$

$$\begin{array}{l} \models \texttt{fun(}E\texttt{,}x\texttt{,}M\texttt{)} : \tau_1 \xrightarrow{\pi} \tau_2 \qquad \text{if } \mathcal{L}^c(E) \subseteq \pi \text{ and} \\ \quad \forall v \forall \mathcal{E}.\ \text{if} \models v : \tau_1,\ \mathcal{E} \models \mu \text{ and } E\{x \mapsto v\} \vdash (\mathcal{E}, M) \Longrightarrow (\mathcal{E}', v') \\ \qquad\quad \text{then } \mathcal{E}' \models \mu \text{ and} \models v' : \tau_2. \end{array}$$

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

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

$$\frac{\models v : \tau_1}{\models v : \tau_2} \ \text{ if } \ \tau_1 \ll \tau_2$$

It should be noted that the definitions for $\mathcal{L}^c(v)$ and $\models v : \tau$ are not syntactic but reflect the actual computational properties of the value. Canonical values have in general multiple typing but `wrong` has none.

We can then show the following theorem.

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

This is proved by showing the following stronger property by induction on the structure of expressions.

> For any variable environment E, any $\mathcal{R}$-environment $\mathcal{E}$ and any typing $\mathcal{A} \rhd M : \tau$, if $E \models \mathcal{A}$, $\mathcal{E} \models \mu$ and $E \vdash (\mathcal{E}, M) \Longrightarrow (\mathcal{E}', v)$ then $\mathcal{E}' \models \mu$ and $\models v : \tau$.

For details of this proof, the reader is referred to [?]. Since `wrong` has no typing, this theorem implies the following

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

Since it can be shown that if $\models v : \tau$ then $\mathcal{L}^c(v) \subseteq \mathcal{L}(\tau)$, the above theorem also implies the following desired property on the locality of expressions.

Corollary 2 *For any closed expression M, if $\emptyset_{\mathcal{A}} \rhd M : \tau$ and $\emptyset_E \vdash (\emptyset_{\mathcal{E}}, M) \Longrightarrow (\mathcal{E}, v)$ then $\mathcal{L}^c(v) \subseteq \mathcal{L}(\tau)$ and $\mathcal{E}$ respects the repository constraint μ.*

The difference between $\mathcal{L}$ and $\mathcal{L}^c$ should be noted. $\mathcal{L}(\tau)$ is a static property available at compile-time. On the other hand, we have defined $\mathcal{L}^c(v)$ based on the actual reference structure contained in v so that it represents the *actual* locality of the value v. As such, this information is not available at compile-time. The above corollary guarantees that the locality of an expression represented by its type is always a correct estimate of the actual locality of the result of the run-time computation and that a type correct program will never produce a reference environment that violates the predefined reachability constraint.

5 Towards a Type Inference System

The type system we have just described is based on the simply typed lambda calculus and is too restricted to be a type system for a practical programming language. First, it cannot represent generic code; and second it requires tedious and often obvious type declarations. This restriction is particularly problematic in our system due to the need for additional locality specifications which make programs even less generic and the required type specifications more complicated. It is therefore essential to introduce *polymorphism* and *type inference* into our type system. One way to achieve this is to extend the type inference method developed for the polymorphic programming language ML [?, ?]. Here we only give an informal description on how this extension can be introduced. A full description of the system requires a certain amount of mathematical development and is beyond the scope of this paper.

In ML, program code does not carry type specifications. The distinguishing feature of ML's type system is that for any type consistent *raw expression* (i.e an expression without type specifications) it infers a *principal type scheme* containing *type variables.* This is a type scheme such that all ground instances obtained from it by substituting type variables with some type terms are types of the expression, and conversely, any type of the expression is such a ground instance. For example, for the raw expression λ`x.x` (the identity function), ML infers the principal type scheme `'a`$\rightarrow$`'a` where `'a` is a type variable. The (infinite) set of types obtainable form this scheme by substituting `'a` with some type is exactly the set of derivable types for the above raw expression, and therefore, the raw expression can be used as a program of any type of the form $\tau \rightarrow \tau$. By this mechanism, ML achieves polymorphism and relieves the programmer of making complicated type assertions.

In order to apply this idea to our system, we need to extend ML's type inference system in three ways. The first one is to introduce variables for localities. The necessity of this extension is seen by considering the raw expression λ**`x.!x`**. According to our typing rules, it is easily seen that this raw expression has types of the form $\mathtt{ref(}\pi_1\mathtt{,}\tau\mathtt{)} \xrightarrow{\pi_2} \tau$ for any locality π_1 and any type τ. (The locality π_2 is determined by the type assignment.) A type scheme that represents those types would therefore look like **`ref('r,'a)`** $\rightarrow$ **`'a`** where **`'r`** is a *locality variable* representing arbitrary localities. This extension can be done by introducing *sorts* in the language of type schemes, as it was done in [?].

Another and more difficult extension is needed to represent various *constraints* associated with some of the typing rules in our type system. For example, the rule (new) for reference creation is specified with a constraint of the form $\mathcal{L}(\tau) \subseteq \mu(R)$ saying that the locality of the argument type must be smaller than the predefined reachability set of the repository. The rules for $M.l$ (field extraction), **modify**, $M_1 := M_2$ and **insert** are also specified with constraints which cannot be represented by conventional type schemes. To give the exact typing schemes to raw expressions that involve those expression constructors, we must invent a syntactic mechanism to represent those constraints. One approach is to refine the notion of typing schemes to *conditional* typing schemes [?]. This is a typing scheme associated with a set of syntactic conditions that control the instantiation of type variables. As an example, a most general typing scheme for the raw expression λ**`x.new(R,x)`** is the following syntactic formula

`'a` $\rightarrow$ **`ref(R,'a)`** `where{` **`subset(locality('a),`**μ**`(R))`** `}`

`subset(locality('a),`μ**`(R))`** in the `where` clause is a syntactic condition on the substitution of the type variable **`'a`**. A ground substitution θ *satisfies* this condition if $\mathcal{L}(\theta(\texttt{'a})) \subseteq \mu(R)$. A substitution θ can be applied to the above typing scheme only if it satisfies the condition. Under this definition the above formula represents the precise set of types of the above raw expression. Other constraints can also be handled by introducing appropriate syntactic conditions. We can then apply the method developed in [?] to combine these conditions and the basic type inference methods to achieve a complete type inference system for the type system described in this paper.

In order to obtain the full power of ML-style polymorphism, we must combine the type inference system for our base language with the polymorphic *let* constructor of the form **let** x = M **in** N **end**. This construct allows different occurrences of the variable x in N to be instantiated with different type variable substitutions and it is the only source of ML polymorphism. The necessary extension to include this construct is well understood for ML [?]. (See also [?] for various formal properties of the type system of [?].) However, it is known [?, ?] that the typing rule for **let** as defined in [?] does not agree with the operational semantics for references and a naive mixture of references and the polymorphic **let** results in an unsound type system. The following example is given in [?]:

```
let
   f = new(λx.x)
in (f:=(λx.x + x), (!f)(true))
end
```

If the type system treats the primitive **new** as an ordinary expression constructor then it would infer the type *bool* for the above expression but the expression causes a run-time type error if the evaluation of a pair is left-to-right. In [?, ?], solutions were proposed. They differ in details of the technical treatment but are both based on the idea that the type system prohibits reference values from having a polymorphic type. Either of these solutions can be adopted by our system.

The resulting type system is particularly useful for controlling locality. It not only enforces a predefined reachability constraint but it also infers the exact locality information for arbitrary complex data structures.

6 Exploiting Object Locality

At first glance, the introduction of locality information in database systems seems to merely complicate the tasks of programmers and users: They lose the illusion of a uniform (persistent) store and have to be aware of the repository structure and the restrictions imposed by reachability constraints.* However, since we regard *operational support* for large scale data structures with persistence, access optimization, access control, garbage collection, concurrency control and recovery as an essential property of data-intensive applications, language mechanisms for locality control seem to be required.

Specifically, we envisage the exploitation of repositories and object locality

- to express organizational and logical clusterings of objects via repositories with suitably selected dependency sets;
- to associate non-functional attributes with repositories (e.g., physical placement on

*Note that a partitioned object store with the trivial reachability constraint $\mu(R) = \mathcal{R}$ for all $R \in \mathcal{R}$ would re-establish this illusion.

devices and network nodes, persistence mechanisms, access rights and ownerships, concurrency control and recovery policies, object faulting strategies);

- to equip module interfaces with locality specifications constraining the utilization of exported references;
- to support the removal and the modification of entire repositories by explicit information about dependent repositories;
- to improve the performance of "global" object store operations (e.g., garbage collection, address translation, backup, recovery) by exploiting the locality within individual repositories.

Another interesting property of our type system is the fact that a non-empty locality tag π in a function type specification $\tau_1 \xrightarrow{\pi} \tau_2$ indicates that the evaluation of a function of this type depends on side effects in repositories π, an information that is particularly valuable for query optimizers in database systems [?].

The generalization of type inference techniques to incorporate object locality, as outlined in section ??, will allow a smooth coexistence of "querying" users unaware of reachability constraints and programmers making heavy use of locality information to develop modular application code amenable to system evolution and performance optimization.

7 Summary and Further Investigations

We have discussed the problems arising in partitioned object stores supporting object identity and developed a type system to formalize and control dependencies between complex objects. The framework of our study was a typed functional programming language enriched with essential features of object-oriented systems like records, (higher-order) functions and references. We developed a static type system where the locality of an expression (i.e. its dependency on repositories) is represented as part of its type and reachability constraints among repositories are enforced by a static typing discipline.

This work on object locality is a first step towards capturing non-structural properties of values in a type system. Therefore, many interesting theoretical and practical issues remain to be investigated. We sketch some of them in the remainder of this paper.

In a static type system it is only possible to capture an approximation of the actual locality of a run-time value. In some cases, the estimate might be too coarse to be practical. One cause of this is due to function closures. To obtain a sound type system, we defined in section ?? the locality of a function closure as the possible locality of the *entire* environment in the closure, which might contain many references that will never be used by the function. One possible refinement is to consider only the set of free variables in the body of a function definition.

In our system a reference is always mutable in any context. In practice, however, object identifiers would be treated in a somewhat more controlled manner. For example, one may not be allowed to change objects in a certain context. In such cases, we may want to distinguish the *strength* of the dependencies induced by different kinds of references. One possible approach would be to separate object identification from mutability and to introduce mutable and immutable reference types.

By exploiting an ordering based on the ambiguity of locality information, we achieved a degree of flexibility in constructing lists of values of different localities. However, the current system is still very restricted in that it does not allow the mixing of references whose referred values have different localities. For example, suppose we have the following objects

```
helen = new(Permanent, [Name = "Helen", Age = 35]);
john = new(Temporary, [Name = "John", Age = 41]]);
susan = new(Temporary, [Name = "Susan", Age = 18, Boss = helen]);
mary = new(Temporary, [Name = "Mary", Age = 18, Boss = john]);
```

then the list {`helen, john`} is legal but {`susan, mary`} is illegal. This restriction is needed to maintain the soundness of the type system. However, if these references are immutable then this restriction becomes unnecessary. This example provides another incentive to study immutable reference types.

Another natural extension is to include recursive types which are essential to represent cyclic structures. One approach is to follow [?, ?] and to use *regular trees* [?] to represent recursive types.

Throughout this paper we assumed that the set $\mathcal{R}$ of repositories and their reachability constraint μ are fixed and known to the compiler. One extension that seems to be useful for maintaining long-lived data is to allow elements of $\mathcal{R}$ to be treated as values in some context. To achieve this extension without sacrificing a static type system constitutes a challenge. Useful techniques for this extension might be derived from the typing discipline for modules [?] and the level distinction in a type system [?].

A final extension related to the type inference method we have described in section ?? is not to assume a *predefined* reachability constraint μ but to infer a minimal (or principal) μ a given expression satisfies.

Acknowledgement

The authors would like to thank Malcolm Atkinson for stimulating discussions on controlling object locality. The second author also thanks John Launchbury for his comments about our earlier results on type inference.

References

[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.

[ABM88] M.P. Atkinson, P. Buneman, and R. Morrison, editors. *Data Types and Persistence.* Topics in Information Systems. Springer-Verlag, 1988.

[ACC81] M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott. PS-Algol: An algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7), July 1981.

[Car88] L. Cardelli. Types for data-oriented languages. In *Advances in Database Technology, EDBT '88*, volume 303 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1988.

[CD87] M.J. Carey and D.J . DeWitt. An overview of the EXODUS project. *Database Engineering, Special Issue on Extensible Database Systems*, 10(2), June 1987.

[CDRS86] M. Carey, D. DeWitt, J. Richardson, and E. Sheikta. Object and file management in the EXODUS extensible database system. In *Proc. of the 12th VLDB Conference*, Kyoto, Japan, August 1986.

[Cop85] M. Coppo. A ccompleteness theorem for recursively defined types. In G. Goos and J. Hartmanis, editors, *Automata, Languages and Programming, 12th Colloquium*, volume 194 of *Lecture Notes in Computer Science*, pages 120–129. Springer, July 1985.

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

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

[HFLP89] L.M. Haas, J.C. Freytag, G.M. Lohmann, and H. Pirahesh. Extensible query processing in Starburst. In *ACM-SIGMOD International Conference on Management of Data*, pages 377–388, Portland, Oregon, 1989.

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

[KC86] S. Khoshafian and G. Copeland. Object identity. In *Proc. of 1st Int. Conf. on OOPSLA*, Portland, Oregon, October 1986.

[Mac86] D.B. MacQueen. Using dependent types to express modular structure. In *Conf. Record 13th Ann. Symp. Principles of Programming Languages*, pages 277–26. ACM, January 1986.

[Mac88] D. MacQueen. References and weak polymorphism. Note in Standard ML of New Jersey Distribution Package, 1988.

[MBCD89] R. Morrison, A.L. Brown, R. Connor, and A. Dearle. The Napier88 reference manual. PPRR 77-89, Universities of Glasgow and St Andrews, 1989.

[MH88] J.C. Mitchell and R. Harper. The essence of ML. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 28–46, San Diego, Ca., January 1988.

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

[Mos89] J.E.B. Moss. Addressing large distributed collections of persistent objects: The Mneme project's approach. In *Proc. of the 2nd Workshop on Database Programming Languages, Portland, Oregon*, pages 358–374, June 1989.

[MS89] F. Matthes and J.W. Schmidt. The type system of DBPL. In *Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, pages 255–260, June 1989.

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

[OBBT89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli – a polymorphic language with static type inference. In *ACM-SIGMOD International Conference on Management of Data*, pages 46–57, Portland, Oregon, 1989.

[Oho90] A. Ohori. Representing object identity in a pure functional language. In *Proc. 3rd Int. Conf. on Database Theory*, Paris, France, 1990.

[OMS90] A. Ohori, F. Matthes, and J.W. Schmidt. A static type system for object locality control. (In preparation), 1990.

[R8́9] D. Rémy. Typechecking records and variants in a natural extension of ML. In D. MacQueen, editor, *ACM Conf. on Principles of Programming Languages*, 1989.

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

[SEM88] J.W. Schmidt, H. Eckhardt, and F. Matthes. DBPL Report. DBPL-Memo 111-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, 1988.

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

[Wan84] M. Wand. A types-as-sets semantics for Milner-style polymorphism. In *Proc. 11th ACM Symp. on Principles of Programming Languages*, pages 158–164, January 1984.