Orderings and Types in Databases*

Atsushi Ohori†

Department of Computer and Information Science/D2
University of Pennsylvania
Philadelphia, PA 19104-6389

Abstract

This paper investigates a method to represent database objects as typed expressions in programming languages. A simple typed language supporting non-flat records, higher-order relations, and natural join expressions is defined. A denotational semantics of this language is then presented. Expressions are interpreted into a domain containing Smyth's powerdomain. In order to give semantics to types, a new model of types, a filter model is proposed. Types are then interpreted as filters in a domain. The type inference system of the language is shown to be sound in this model.

1 Introduction

There are a number of attempts to generalize the relational data model beyond first-nomal-form relations [FT83, ÖY85, RKS85]; there are also other data models that can be seen as generalizations of the relational data model [AB84, BK86]. The motivation of this study is to draw out the connection between these "higher-order" relations and data types in programming languages so that we can develop a strongly typed programming language in which these data structures are directly available as typed expressions.

We regard database objects as *descriptions* of real-world objects. Such descriptions are *ordered* by how well they describe real-world objects. Relations are then regarded as sets of descriptions describing sets of real-world objects. In [BJO91], it is shown that *natural join* can be characterized as the least upper bound operation in Smyth's powerdomain of descriptions. Based on this result, we present a simple typed language that supports non-flat records, higher-order relations, and natural join expressions. We then present a denotational semantics of this language.

Expressions of the language are interpreted in a domain containing Smyth's powerdomain. In order to give semantics to types, we propose a filter model of types. We regard types as sets of values having common structures. In a domain of descriptions, such sets have properties that they are upward closed and they are closed under finite greatest lower bounds. We therefore interpret types as filters in a semantic domain and show the semantic soundness of the type system. The filter model is particularly suitable for types of partial objects. This model can also give precise semantics to *multiple inheritance* studied by Cardelli [Car84].

The rest of this paper is organized as follows. In section 2 we introduce non-flat records to represent database objects and define their ordering. We then introduce types of records and define their ordering. In section 3 we extend expressions, types, and their orderings to sets to represent higher-order relations. We then show that natural join expressions can be generalized in typed higher-order relations. In section 4, we give a formal definition of our language. In section 5, we construct a semantic domain of expressions and give semantics to expressions. We then introduce the filter model of types and give semantics to types and show the soundness of the type inference system.

*Published in a chapter of the book **Database Programming Langauges**, P. Buneman and F. Bancilhon editor, Addison-Wesley, 1989.

†This work was supported by grants from AT&T, the Army Research Office, and by the National Science Foundation(IRI 86-10617). On leave form Oki Electric Industry Co., JAPAN

2 Database Objects as Records

We represent database objects as labeled record structures. Records are associations of labels and values. We assume that we are given a countable set L of labels and sets $B_1, \ldots, B_n$ of primitive values such as the set of integers. Expressions for records are then inductively defined as:

1. b is an expression if $b \in B_i$.
2. $(l_1 \to e_1, \ldots, l_n \to e_n)$ is an expression if $e_1, \ldots, e_n$ are expressions and $l_1, \ldots, l_n \in L$, where $l_1, \ldots, l_n$ are all distinct.

The following is an example of an expression:

$$(\mathit{Name} \to \mathit{'J.\ Doe'}, \mathit{Emp\#} \to 1234, \mathit{Age} \to 21)$$

In database programming it is convenient to have *null* values to represent incomplete information. For example, we want to allow the following expression:

$$(\mathit{Name} \to \mathit{'J.\ Doe'}, \mathit{Emp\#} \to 1234, \mathit{Age} \to \mathit{null}_{int})$$

when the value of *Age* is unknown. In order to allow these null values we add the following rule:

3. null_{B_i} is an expression.

One distinguishing property about these database objects is that they are *ordered*. The ordering comes from an assumption – usually unstated because it is so obvious – that they describe some real-world objects. As such descriptions, they are essentially incomplete. These incomplete descriptions are partially ordered by how well they describe real-world objects. In the relational data model this ordering was first observed by Zaniolo [Zan84] in connection with null values. The following is an example of this ordering:

$$\begin{array}{rl} & (\mathit{Name} \to \mathit{'J.\ Doe'}, \mathit{Emp\#} \to 1234) \\ \sqsubseteq & (\mathit{Name} \to \mathit{'J.\ Doe'}, \mathit{Emp\#} \to 1234, \mathit{Age} \to \mathit{null}_{int}) \\ \sqsubseteq & (\mathit{Name} \to \mathit{'J.\ Doe'}, \mathit{Emp\#} \to 1234, \mathit{Age} \to 21) \end{array}$$

Formally we define:

1. $e \sqsubseteq e$.
2. $\mathit{null}_{B_i} \sqsubseteq b$ for all $b \in B_i$.
3. $(l_1 \to e_1, \ldots, l_n \to e_n) \sqsubseteq (l_1 \to e'_1, \ldots, l_m \to e'_m)$ whenever $n \leq m$ and $e_i \sqsubseteq e'_i$ for all $1 \leq i \leq n$.

From this definition, it is easily seen that $\sqsubseteq$ is a partial ordering on expressions.

The least upper bound (lub) of this ordering corresponds to the conjunction of descriptions if they are compatible. For example, if

$$e_1 = (\mathit{Name} \to \mathit{'J.\ Doe'}, \mathit{Emp\#} \to 1234)$$

and

$$e_2 = (\mathit{Emp\#} \to 1234, \mathit{Age} \to 21)$$

then

$$e_1 \sqcup e_2 = (\mathit{Name} \to \mathit{'J.\ Doe'}, \mathit{Emp\#} \to 1234, \mathit{Age} \to 21)$$

However, $(\mathit{Name} \to \mathit{'J.\ Doe'}, \mathit{Emp\#} \to 1234) \sqcup (\mathit{Name} \to \mathit{'K.\ Smith'})$ does not exist. As we shall see in the next section, natural join operation can be regarded as the lub operation extended to a powerdomain. This lub operation is also known as the *unification* in *unification-based* grammatical formalisms, where data are descriptions of linguistic entities (see [Shi85] for a survey).

Next we define types for these expressions. Since each primitive set of values corresponds to a basic type and each label denotes certain set of values, types for expressions are defined as:

1. For each primitive set of values B_i there is a constant type τ_i.

2. $(l_1 : \sigma_1, \ldots, l_n : \sigma_n)$ is a type if $\sigma_1, \ldots, \sigma_n$ are types and $l_1, \ldots, l_n \in L$, where $l_1, \ldots, l_n$ are all distinct.

These types can be regarded as specifications of structures of database objects. Since database objects are partial descriptions, these types should specify partial structures. A value is regarded as having a type if the value has the partial structure specified by the type. This observation leads us to define the following typing rules syntactically similar to the type system proposed by Cardelli [Car84]:

1. $b : \tau_i$ if $b \in B_i$.

2. $null_{B_i} : \tau_i$.

3. $(l_1 \to e_1, \ldots, l_n \to e_n) : (l_1 : \sigma_1, \ldots, l_m : \sigma_m)$ if $m \leq n$ and for all $1 \leq i \leq m, e_i : \sigma_i$.

The following is an example of typing:

$$(Name \to \text{'}J.\ Doe', Emp\# \to 1234) : (Name : string, Emp\# : int)$$

From the definitions of typing and $\sqsubseteq$ we can show by simple structural induction that:

Theorem 1 *If $e : \sigma$ and $e \sqsubseteq e'$ then $e' : \sigma$.*

Indeed the following typing is also valid:

$$(Name \to \text{'}J.\ Doe', Emp\# \to 1234, Age \to : 21) : (Name : string, Emp\# : int)$$

In our type system, types therefore correspond to upward closed sets of values. Intuitively, this corresponds to the fact that if a database object has certain structure then any better defined objects also have the structure. For example, if a database object has an attribute *Name* with the type *string*, then we expect that all better defined objects also have this structure.

Now if we regard types as sets of values then the above typing rules induce an inclusion ordering on types. We define a syntactic relation $\preceq$ on types to represent this ordering:

1. $\sigma \preceq \sigma$.

2. $(l_1 : \sigma_1, \ldots, l_n : \sigma_n) \preceq (l_1 : \sigma'_1, \ldots, l_m : \sigma'_m)$ if $m \leq n$ and for all $1 \leq i \leq m, \sigma_i \preceq \sigma'_i$.

It is easy to check that $\preceq$ is a partial ordering. This ordering is the ordering of the generality of specifications of types. For example,

$$(Name : string, Emp\# : int, Age : int) \preceq (Name : string, Emp\# : int)$$

Since more general means less informative, we can see why the definition of $\preceq$ is the inverse of the definition of $\sqsubseteq$.

From the definitions of typing and $\preceq$ we can show by simple structural induction that:

Theorem 2 *If $e : \sigma$ and $\sigma \preceq \sigma'$ then $e : \sigma'$.*

The next theorem connects $\sqsubseteq$ and $\preceq$:

Theorem 3 *If $e : \sigma, e' : \sigma'$ and $e \sqcup e'$ exists then $\sigma \sqcap \sigma'$ exists and $e \sqcup e' : \sigma \sqcap \sigma'$.*

Proof. By induction on the structures of e and e'.□
For example, we have:

$$\begin{aligned} &(Name \to \text{'}J.\ Doe', Emp\# \to 1234) \sqcup (Emp\# \to 1234, Age \to 21) \\ = \quad &(Name \to \text{'}J.\ Doe', Emp\# \to 1234, Age \to 21) \end{aligned}$$

Name	*Age*	*Emp#*
$'J.\ Doe'$	21	1234
$'K.\ Smith'$	31	5678

$$\{ \quad (Name \to 'J.\ Doe', Age \to 21, Emp\# \to 1234), \\ (Name \to 'K.\ Smith', Age \to 31, Emp\# \to 5678) \quad \}$$

Figure 1: A relation and its representation as a set of expressions

and

$$(Name : string, Emp\# : int) \sqcap (Emp\# : int, Age : int) \\ = \quad (Name : string, Emp\# : int, Age : int)$$

thus

$$(Name \to 'J.\ Doe', Emp\# \to 1234) \sqcup (Emp\# \to 1234, Age \to 21) \\ : \quad (Name : string, Emp\# : int) \sqcap (Emp\# : int, Age : int)$$

As we shall see in the next section, this property, when extended to sets, provides types for generalized natural join expressions.

3 Relations as Sets of Records

Relations are sets of database objects and databases are sets of relations. We therefore want to allow sets of expressions themselves as expressions. Figure 1 shows a simple example of a relation and its representation as a set of expressions. In this section we extend expressions, types, and their orderings to sets.

Since individual expressions correspond to partial descriptions, sets of expressions correspond to sets of partial descriptions and presumably describe sets of real-world objects. We therefore want to treat these sets of descriptions as descriptions of sets of objects and to order them by their goodness of descriptions. If our primary interest in database programming is query processing or information retrieval from given set of data, then an appropriate ordering is:

$$A \sqsubseteq_0 B \text{ iff } \forall b \in B \exists a \in A. a \sqsubseteq b$$

known as Smyth's powerdomain ordering. Intuitively, this is an ordering on sets of descriptions which "over-describe" real-world sets; a set contains enough descriptions to describe all objects in a real-world set but may contain irrelevant descriptions. $A \sqsubseteq_0 B$ means that B is a less ambiguous and better defined description to a real-world set. A query processing can then be regarded as a process which takes a set of descriptions D and return another set of descriptions A such that $D \sqsubseteq_0 A$. Indeed natural join and selection, the two major operations for query processing, have the property that they carry relations higher in this ordering. It should be noted, however, that this ordering is not appropriate for the ordering on databases themselves. If our interests are operations on databases such as database merging then we need other orderings. In [BJO91] various properties of orderings on database sets, including this ordering were studied.

For arbitrary sets, however, $\sqsubseteq_0$ is not a partial ordering; it is a pre-ordering and a partial ordering is derived by taking equivalence classes. Define $A \simeq B$ as $A \sqsubseteq_0 B$ and $B \sqsubseteq_0 A$. If $A \simeq B$ then we regard A and B as having same amount of information. We use this equivalence relation as equality between sets of descriptions and regard a set of descriptions as a representative of the corresponding equivalence class. Then $\sqsubseteq_0$ becomes a partial ordering. Thus we now regard equivalence classes of sets of expressions as descriptions of sets of objects and extend expressions to these equivalence classes. We also extend the ordering $\sqsubseteq$ on expressions to these equivalence classes, i.e. if $[A]$ and $[B]$ are equivalence classes of sets of expressions A and B then $[A] \sqsubseteq [B]$ if $A \sqsubseteq_0 B$.

For $\simeq$ we have [Smy78]:

$$\begin{array}{l}\{(Pname \to 'Nut', Supplier \to \{\ (Sname \to 'Smith', City \to 'London'),\\ \qquad\qquad\qquad\qquad\qquad\quad (Sname \to 'Jones', City \to 'Paris'),\\ \qquad\qquad\qquad\qquad\qquad\quad (Sname \to 'Blake', City \to 'Paris')\}),\\ \quad (Pname \to 'Bolt', Supplier \to \{\ (Sname \to 'Blake', City \to 'Paris'),\\ \qquad\qquad\qquad\qquad\qquad\quad (Sname \to 'Adams', City \to 'Athens')\})\}\end{array}$$

Figure 2: Higher-order relation

Theorem 4 *$A \simeq \overline{A}$ and $A \simeq B$ iff $\overline{A} = \overline{B}$, where $\overline{A} = \{e | \exists a \in A. a \sqsubseteq e\}$.*

If we restrict attentions to finite sets, then this theorem says that a set A is equivalent to the co-chain of the set of minimal elements in A, where a co-chain is a set such that no member in the set is greater than any other member in the set. Thus we can use co-chains as canonical representatives of equivalence classes. Intuitive justification for this equivalence is that if an object x is in an answer to a query then we know that any better defined object y such that $x \sqsubseteq y$ also satisfies the query. Thus all better defined objects are redundant and can be eliminated from the answer.

We have seen that sets of expressions can be also regarded as descriptions and the approximation ordering $\sqsubseteq$ on expressions can be extended to sets of expressions. We can then include sets of expressions in our language and allow records to contain these sets as values. Since now sets are regarded as expressions ordered by $\sqsubseteq$, by applying the same argument, we can further extend our language to allow sets of sets of expressions as expressions. Indeed we can carry this extension process to any depth.

In the syntax of the language this extension can be done by simply adding the rule:

4. $\{e_1, \ldots, e_k\}$ is an expression if $e_1, \ldots, e_k$ are expressions.

where we allow the empty set $\{\}$ as an expression, since the empty set can be regarded as a valid response to a query. We call these expressions as *set expressions*. Set expressions are regarded as representatives of corresponding equivalence classes. The extended language not only allows simple relations such as the example in Figure 1 but also allows sets of relations and "higher-order" relations such as the example in Figure 2.

In the previous section we have seen that the lub of expressions under the ordering $\sqsubseteq$ corresponds to the conjunction of descriptions. About the lub of the extended ordering on set expressions:

Lemma 5 *$A \sqcup B = \{a \sqcup b | a \in A, b \in B, a \sqcup b$ exists$\}$ where all sets are regarded as representatives of their equivalence classes.*

Proof. It is clear that $\{a \sqcup b | a \in A, b \in B, a \sqcup b$ exists$\}$ is an upper bound of A and B. Let C be any upper bound of A and B, i.e. $A \sqsubseteq_0 C, B \sqsubseteq_0 C$. Then by the definition of $\sqsubseteq_0$, for any $c \in C$ there are $a \in A, b \in B$ such that $a \sqsubseteq c, b \sqsubseteq c$. Then we have $a \sqcup b \sqsubseteq c$. Thus $\{a \sqcup b | a \in A, b \in B, a \sqcup b$ exists$\} \sqsubseteq_0 C$.□

The importance of this lub in connection with relational algebra is stated in the following theorem [BJO91]:

Theorem 6 *If A, B are co-chains of flat records then $A \sqcup B$ is the natural join of A and B.*

From this connection we can regard the lub operation as a generalized natural join on extended expressions. We write $A \bowtie B$ for $A \sqcup B$ if A, B are set expressions. Note that if A, B are set expressions then $A \bowtie B$ always exists.

This operation can be also used as selection operation. For example, if

$$\begin{array}{l}e = \{(Name \to 'J.\ Doe', Age \to 21),\\ \qquad (Name \to 'K.\ Smith', Age \to 31)\}\end{array}$$

then

$$e \bowtie \{(Age \to 21)\} = \{(Name \to 'J.\ Doe', Age \to 21)\}$$

We now turn our attentions to types for database sets. As we extended expressions to include set expressions, we extend our type system to include set types by adding the following rule to the syntax of types:

4. $\{\sigma_1, \ldots, \sigma_n\}$ is a type if $\sigma_1, \ldots, \sigma_n$ are types.

where we also allow $\{\}$ for convenience.

We noted in the previous section that types specify partial structures of objects. For set types, this corresponds to the following typing rule:

4. $\{e_1, \ldots, e_n\} : \{\sigma_1, \ldots, \sigma_m\}$ if $\forall e \in \{e_1, \ldots, e_n\} \exists \sigma \in \{\sigma_1, \ldots, \sigma_m\}. e : \sigma$.

It is easy to check that this typing rule yields an upward closed set in set expressions under our ordering on sets and the theorem 1 also holds.

This typing rule also induces an inclusion ordering on set types regarded as sets of values (i.e. sets of set expressions). In order to represent this ordering, we first define the following pre-ordering on set types:

$$\sigma \preceq_0 \sigma' \text{ iff } \forall \iota \in \sigma \exists \iota' \in \sigma'. \iota \preceq \iota'$$

As before a partial ordering is obtained by defining equivalence relation $\simeq$ as $\sigma \simeq \sigma'$ iff $\sigma \preceq_0 \sigma'$ and $\sigma' \preceq_0 \sigma$. Then by the definition of typing, $\sigma \simeq \sigma'$ iff for any e, $e : \sigma \Leftrightarrow e : \sigma'$. Therefore this equivalence relation exactly corresponds to the equality between types regarded as sets of values. We therefore regard set types as representatives of equivalence classes.

Parallel to theorem 4, we can show:

Theorem 7 *$\sigma \simeq \underline{\sigma}$ and $\sigma \simeq \sigma'$ iff $\underline{\sigma} = \underline{\sigma'}$, where $\underline{\sigma} = \{\iota | \exists \iota' \in \sigma. \iota \preceq \iota'\}$.*

Therefore set types can be also represented by co-chains.

Note that the definition of $\preceq_0$ is the inverse of the definition of $\sqsubseteq_0$ and the extended ordering $\preceq$ still corresponds to the generality of specifications. If we replace $\sigma \preceq \sigma'$ with $\sigma' \sqsubseteq \sigma$ then we get the same definitions and properties for orderings on expressions and types.

We now extend the ordering relation $\preceq$ on types to set types using the partial ordering $\preceq_0$ on equivalence classes of sets of types. It can then shown that theorem 2 still holds for the extended types. We write $\sigma \wedge \sigma'$ for $\sigma \sqcap \sigma'$ if σ, σ' are set types. From the duality of $\sqsubseteq$ and $\preceq$, we can see that $\sigma \wedge \sigma'$ always exists if σ, σ' are set types.

The following theorem connects $\bowtie$ and $\wedge$:

Theorem 8 *If A, B are set expressions with $A : \sigma_1, B : \sigma_2$ then $A \bowtie B : \sigma_1 \wedge \sigma_2$.*

Proof. Let $a \sqcup b$ be any element in $A \bowtie B$. Since $A : \sigma_1$ and $B : \sigma_2$, there are $\iota_1 \in \sigma_1$ and $\iota_2 \in \sigma_2$ such that $a : \iota_1$ and $b : \iota_2$. Then by theorem 3, $a \sqcup b : \iota_1 \sqcap \iota_2$. But by definition $\iota_1 \sqcap \iota_2 \in \sigma_1 \wedge \sigma_2$. This shows $A \bowtie B : \sigma_1 \wedge \sigma_2$.□

This theorem shows that we have successfully generalized natural join in typed higher-order relations. Figure 3 is an example of a natural join of typed higher-order relations.

4 Definition of the Language

In this section we give formal definition of our language supporting records, higher-order relations, and natural joins.

4.1 Expressions

We use $l, l_1, \ldots$ for elements of L. The syntax of expressions is given by the following abstract syntax grammar:

$$\begin{aligned} e \quad ::= \quad & b \ \ (b \in B_i) \mid null_{B_i} \mid \\ & (l_1 \to e_1, \ldots, l_n \to e_n) \mid (\ldots, l \to e, \ldots).l \mid \\ & \{e_1, \ldots, e_m\} \mid \{e_1, \ldots, e_n\} \bowtie \{e'_1, \ldots, e'_m\}. \end{aligned}$$

$$
\begin{aligned}
r_1 =&\{(Pname \to 'Nut', Supplier \to \{\ (Sname \to 'Smith', City \to 'London'),\\
&\qquad\qquad\qquad\qquad\quad (Sname \to 'Jones', City \to 'Paris'),\\
&\qquad\qquad\qquad\qquad\quad (Sname \to 'Blake', City \to 'Paris')\}),\\
&\quad (Pname \to 'Bolt', Supplier \to \{\ (Sname \to 'Blake', City \to 'Paris'),\\
&\qquad\qquad\qquad\qquad\quad (Sname \to 'Adams', City \to 'Athens')\})\}\\
&: \{(Pname : string, Supplier : \{(Sname : string, City : string)\})\}
\end{aligned}
$$

$$
\begin{aligned}
r_2 =&\{(Pname \to 'Nut', Supplier \to \{(City \to 'Paris')\}, Qty \to 100),\\
&\quad (Pname \to 'Bolt', Supplier \to \{(City \to 'Paris')\}, Qty \to 200)\}\\
&:\{(Pname : string, Supplier : \{(City : string)\}, Qty : int)\}
\end{aligned}
$$

$$
\begin{aligned}
r_1 \bowtie r_2 =&\{(Pname \to 'Nut', Supplier \to \{\ (Sname \to 'Jones', City \to 'Paris'),\\
&\qquad\qquad\qquad\qquad\quad (Sname \to 'Blake', City \to 'Paris')\}, Qty \to 100),\\
&\quad (Pname \to 'Bolt', Supplier \to \{(Sname \to 'Blake', City \to 'Paris')\}, Qty \to 200)\}\\
&:\{(Pname : string, Supplier : \{(Sname : string, City : string)\}, Qty : int)\}
\end{aligned}
$$

Figure 3: Natural join of typed higher-order relations

Among expressions, various equations should hold. The first axiom of equality is the axiom for dot expressions, (i.e. expressions of the form $(\ldots, l \to e, \ldots).l$):

$$(l_1 \to e_1, \ldots, l_i \to e_i, \ldots, l_n \to e_n).l_i = e_i \tag{1}$$

In order to define axioms for set expressions and join expressions, (i.e. the expressions of the form $\{e_1, \ldots, e_n\} \bowtie \{e'_1, \ldots, e'_m\}$), we first define the syntactic relation $\sqsubseteq$ on the sublanguage of expressions that do not contain dot expressions and join expressions as subexpressions.

$$
\begin{aligned}
e &\sqsubseteq e\\
null_{B_i} &\sqsubseteq b \text{ for all } b \in B_i\\
(l_1 \to e_1, \ldots, l_n \to e_n) &\sqsubseteq (l_1 \to e'_1, \ldots, l_m \to e'_m) \text{ if } n \le m \text{ and } e_i \sqsubseteq e'_i \text{ for all } 1 \le i \le n\\
\{e_1, \ldots, e_n\} &\sqsubseteq \{e'_1, \ldots, e'_m\} \text{ if } \forall e' \in \{e'_1, \ldots, e'_m\}.\exists e \in \{e_1, \ldots, e_n\}.e \sqsubseteq e'
\end{aligned}
$$

The axiom for set expressions is then defined as:

$$\{e_1, e_2, e_3, \ldots, e_n\} = \{e_1, e_3, \ldots, e_n\} \text{ if } e_1 \sqsubseteq e_2 \tag{2}$$

Note that this rule induces an equivalence relation that makes $\sqsubseteq$ a partial ordering. Let $\sqcup$ be the least upper bound of this partial ordering. The axiom for join expressions is defined as:

$$\{e_{i_1}, \ldots, e_{i_n}\} \bowtie \{e_{j_1}, \ldots, e_{j_m}\} = \{e'_{i_k} \sqcup e'_{j_l} | 1 \le k \le n, 1 \le l \le m, e'_{i_k} \sqcup e'_{j_l} \text{ exists}\} \tag{3}$$

where e'_{i_k}, e'_{j_l} are expressions that are equal to e_{i_k}, e_{j_l} respectively and do not contain dot expressions or join expressions as subexpressions.

These rules also define a reduction process which eliminates dot expressions and join expressions and reduces set expressions to corresponding co-chain representatives.

4.2 Types

We assume that there are constant types $\tau_1, \ldots, \tau_n$ associated with $B_1, \ldots, B_n$. Then the syntax of types for expressions is defined by the following abstract syntax grammar:

$$
\begin{aligned}
\sigma \quad ::= \quad & \tau_i |\\
& (l_1 : \sigma_1, \ldots, l_n : \sigma_n) |\\
& \{\sigma_1, \ldots, \sigma_m\} |\\
& \sigma \wedge \sigma' \quad (\text{if } \sigma, \sigma' \text{ are of the form } \{\sigma_1, \ldots, \sigma_n\}).
\end{aligned}
$$

In order to define axioms of equality of types, we first define the syntactic relation $\preceq$ on the sublanguage of types that do not contain meet types (i.e. types of the form $\sigma \wedge \sigma'$):

$$\begin{array}{rcl} \sigma & \preceq & \sigma \\ (l_1 : \sigma_1, \ldots, l_n : \sigma_n) & \preceq & (l_1 : \sigma'_1, \ldots, l_m : \sigma'_m) \text{ if } m \leq n \text{ and } \sigma_i \preceq \sigma'_i \text{ for } 1 \leq i \leq m \\ \{\sigma_1, \ldots, \sigma_n\} & \preceq & \{\sigma'_1, \ldots, \sigma'_m\} \text{ if } \forall \sigma \in \{\sigma_1, \ldots, \sigma_n\}. \exists \sigma' \in \{\sigma'_1, \ldots, \sigma'_m\}. \sigma \preceq \sigma' \end{array}$$

Axiom for set types is then defined as:

$$\{\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_n\} = \{\sigma_1, \sigma_3, \ldots, \sigma_n\} \text{ if } \sigma_2 \preceq \sigma_1 \tag{4}$$

This equation makes $\preceq$ a partial ordering. Let $\sqcap$ be the greatest lower bound of this partial ordering. The axiom for meet types is then defined as:

$$\{\sigma_1, \ldots, \sigma_n\} \wedge \{\sigma'_1, \ldots, \sigma'_m\} = \{\sigma_i \sqcap \sigma'_j | 1 \leq i \leq n, 1 \leq i \leq m, \sigma_i \sqcap \sigma'_j \text{ exists}\} \tag{5}$$

4.3 Rules For Type Inference

Not all expressions are meaningful. One goal of a type system is to identify the set of all syntactically meaningful expressions as the set of *well typed* expressions. We write $\vdash e : \sigma$ for e is *well typed* with type σ. Such well typed expressions are systematically inferred by a *type inference system.*

A type inference system consists of axioms for constant types and inference rules for compound types. Axioms for our type system are:

const $\quad \vdash b : \tau_i \quad$ for all $b \in B_i$

null $\quad \vdash null_{B_i} : \tau_i \quad$ for all B_i

Inference rules for our type system are:

subtype $\quad \dfrac{\vdash e : \sigma \qquad \sigma \preceq \sigma'}{\vdash e : \sigma'}$

records $\quad \dfrac{\vdash e_1 : \sigma_1, \ldots, \vdash e_n : \sigma_n}{\vdash (l_1 \rightarrow e_1, \ldots, l_n \rightarrow e_n) : (l_1 : \sigma_1, \ldots, l_n : \sigma_n)}$

dot $\quad \dfrac{\vdash e : (\ldots, l : \sigma, \ldots)}{\vdash e.l : \sigma}$

set $\quad \dfrac{\vdash e_1 : \sigma_1, \ldots, \vdash e_n : \sigma_n}{\vdash \{e_1, \ldots, e_n\} : \{\sigma_1, \ldots, \sigma_n\}}$

join $\quad \dfrac{\vdash e_1 : \sigma_1 \qquad \vdash e_2 : \sigma_2}{\vdash e_1 \bowtie e_2 : \sigma_1 \wedge \sigma_2}$ $\quad$ if $\sigma_1 = \{\sigma_1^1, \ldots, \sigma_1^n\}$ and $\sigma_2 = \{\sigma_2^1, \ldots, \sigma_2^m\}$

We say that $\vdash e : \sigma$ holds iff there is a proof of it using the above axioms and inference rules.

Based on this type inference system, we can define a typechecking function *type* that takes an expression e and returns a type of e if it is well typed otherwise returns *error*. *type* is defined inductively as follows:

$$\begin{array}{rcl} type(b) & = & \tau_i \ (b \in B_i) \\ type(null_{B_i}) & = & \tau_i \\ type((l_1 \rightarrow e_1, \ldots, l_n \rightarrow e_n)) & = & (l_1 : type(e_1), \ldots, l_n : type(e_n)) \end{array}$$

$$
\begin{aligned}
type(e.l) &= \text{if } type(e) = (\ldots, l:\sigma, \ldots) \text{ then } \sigma \text{ else } error \\
type(\{e_1, \ldots, e_n\}) &= \{type(e_1), \ldots, type(e_n)\} \\
type(e \bowtie e') &= \text{if } type(e) = \{\sigma_1, \ldots, \sigma_n\} \text{ and } type(e') = \{\sigma'_1, \ldots, \sigma'_m\} \text{ then} \\
&\quad type(e) \wedge type(e') \text{ else } error
\end{aligned}
$$

Then this typechecking function is correct with respect to our type inference system, i.e. we can prove the following theorem by induction on the structure of e:

Theorem 9 *If* $type(e) = \sigma$ *then* $\vdash e : \sigma$ *holds.*

This type inference system is not complete under the equality between expressions, i.e. this system does not have the property that if $\vdash e : \sigma$ and $e = e'$ then $\vdash e' : \sigma$ because of $\bowtie$. Suppose $\vdash e_1 : \sigma_1$ and $\vdash e_2 : \sigma_2$ hold and σ_1, σ_2 are set types. Then by the rule *join*, $\vdash e_1 \bowtie e_2 : \sigma_1 \wedge \sigma_2$ hold. However $e_1 \bowtie e_2$ may be equal to the empty set, which has any set types. Because of this incompleteness property, the function *type* does not necessarily return the most specific type (smallest type under $\preceq$) for a given expression e. We nevertheless think that *type* is an appropriate definition for *static* typechecking. To see this consider the following join expression

$$e : \{(Name : string, Age : int)\} \bowtie e' : \{(Name : string, Emp\# : int)\}$$

Although the result of the join may be empty set, we usually think that the type of the result relation is $\{(Name : string, Age : int, Emp\# : int)\}$.

5 Semantics of the Language

In this section we define a denotational semantics of the language. We first define a semantic domain for expressions and define a semantics of expressions. We then show that types are modeled by special subsets called filters in a domain and define a semantics of types. Finally we show the correctness of the type inference system with respect to the semantics.

5.1 Semantic Domain

A semantic domain for expressions is given by a recursive domain equation containing flat domains $\mathcal{B}_i$ for primitive values, total functions $(L \to \mathcal{D})$ for records, and Smyth's powerdomain $\mathcal{P}(\mathcal{D})$ for sets of descriptions.

$$\mathcal{D} = \mathcal{B}_1 + \cdots + \mathcal{B}_n + (L \to \mathcal{D}) + \mathcal{P}(\mathcal{D}) + \{w\} \tag{6}$$

where $+$ is the *separated* sum domain constructor, $\mathcal{B}_i = B_i \cup \{\bot_{\mathcal{B}_i}\}$ with ordering $\bot_{\mathcal{B}_i} \sqsubseteq x$ for all $x \in B_i$, and w is used to interpret the *wrong* value. For $\mathcal{P}(\mathcal{D})$ we include $\emptyset$, the empty set.

A solution of the equation (6) can be found in a particular class of complete partial orders (c.p.o.) called a *bounded complete* ω*-algebraic* c.p.o., or simply *domain.*

A c.p.o. is a partial order $(D, \sqsubseteq)$ satisfying:

1. D has the minimal element $\bot_D$.
2. each directed subset $X \subseteq D$ has a least upper bound $\sqcup X$ where a subset X is directed iff $\forall x, y \in X \exists z \in X. x \sqsubseteq z, y \sqsubseteq z$.

An *isolated* (*finite*) element of a c.p.o. $(D, \sqsubseteq)$ is an element $e \in D$ such that for any directed subset $X \subseteq D$ if $e \sqsubseteq \sqcup X$ then there is $x \in X$ such that $e \sqsubseteq x$. We write D^o for the set of isolated elements of D. A c.p.o. is said to be ω*-algebraic* iff D^o is countable and for all $x \in D$ we have $x = \sqcup\{e | e \in D^o, e \sqsubseteq x\}$. A c.p.o. is said to be *bounded complete* (*consistently complete*) if any bounded subset of D has a least upper bound, where a subset X is bounded if it has an upper bound in D.

Construction of a recursive domain without containing powerdomain can be found in many places such as [MPS86, Bar84, Sch86]. In [Smy78] Smyth showed that domains are closed under the powerdomain

construction based on the pre-ordering $\sqsubseteq_0$ and that a domain equation like (6) can be solved. In what follows we use $\mathcal{D}$ for a domain satisfying (6). We also use injections of component domains $\mathcal{B}_1, \ldots, \mathcal{P}(\mathcal{D})$ into $\mathcal{D}$ implicitly and treat them as if they were actual inclusions.

We use the following notations to represent elements in $\mathcal{D}$.

1. $(l_1 \mapsto d_1, \ldots, l_n \mapsto d_n)$ for the function $f \in (L \to \mathcal{D})$ defined as $f(l) =$ if $l = l_i, 1 \leq i \leq n$ then d_i else $\bot_{\mathcal{D}}$, where we assume that $d_i \neq \bot_{\mathcal{D}}$.
2. $[d_1, \ldots, d_n]$ for the element $d \in \mathcal{P}(\mathcal{D})$ such that $\{d_1, \ldots, d_n\} \in d$, i.e. the equivalence class containing $\{d_1, \ldots, d_n\}$.

It should be noted that the domain $\mathcal{D}$ is equipped with the ordering $\sqsubseteq$. This ordering was originally introduced to model computation. However, if we regard values in $\mathcal{D}$ as descriptions then this ordering corresponds to the approximation ordering on descriptions we discussed in section 2. We therefore believe that the domain $\mathcal{D}$ is an appropriate model of our language.

5.2 Semantics of Expressions

Let *Expr* be the set of expressions. We define a semantics of expressions by the semantic function:

$$\mathcal{E} : Expr \to \mathcal{D}$$

as follows:

$$\begin{aligned}
\mathcal{E}[\![b]\!] &= b \text{ for all } b \in B_i \\
\mathcal{E}[\![null_{B_i}]\!] &= \bot_{\mathcal{B}_i} \\
\mathcal{E}[\![(l_1 \to e_1, \ldots, l_n \to e_n)]\!] &= (l_1 \mapsto \mathcal{E}[\![e_1]\!], \ldots, l_n \mapsto \mathcal{E}[\![e_n]\!]) \\
\mathcal{E}[\![e.l]\!] &= \text{ if } \mathcal{E}[\![e]\!] = (\ldots, l \mapsto d, \ldots) \text{ then } d \text{ else } w \\
\mathcal{E}[\![\{e_1, \ldots, e_m\}]\!] &= [\mathcal{E}[\![e_1]\!], \ldots, \mathcal{E}[\![e_m]\!]] \\
\mathcal{E}[\![e \bowtie e']\!] &= \text{ if } \mathcal{E}[\![e]\!] \sqcup \mathcal{E}[\![e']\!] \text{ exists then } \mathcal{E}[\![e]\!] \sqcup \mathcal{E}[\![e']\!] \text{ else } w
\end{aligned}$$

From this definition, we can easily show, by induction on the structures of expressions, the soundness of the ordering relation on expressions:

Theorem 10 *1. If $e \sqsubseteq e'$ then $\mathcal{E}[\![e]\!] \sqsubseteq \mathcal{E}[\![e']\!]$.*

2. If $e \sqcup e'$ exists then $\mathcal{E}[\![e \sqcup e']\!] = \mathcal{E}[\![e]\!] \sqcup \mathcal{E}[\![e']\!]$.

3. If $e \sqcap e'$ exists then $\mathcal{E}[\![e \sqcap e']\!] = \mathcal{E}[\![e]\!] \sqcap \mathcal{E}[\![e']\!]$.

The equations (1), (2) and (3) between expressions are also sound with respect to this semantics, i.e. the following equations hold:

$$\mathcal{E}[\![(\ldots, l \to e, \ldots).l]\!] = \mathcal{E}[\![e]\!] \tag{7}$$

$$\mathcal{E}[\![\{e_1, e_2, e_3, \ldots\}]\!] = \mathcal{E}[\![\{e_1, e_3, \ldots\}]\!] \text{ if } e_1 \sqsubseteq e_2 \tag{8}$$

$$\mathcal{E}[\![\{e_1, \ldots, e_n\} \bowtie \{e'_1, \ldots, e'_m\}]\!] = \mathcal{E}[\![\{e_i \sqcup e'_j | 1 \leq i \leq n, 1 \leq i \leq m, e_i \sqcup e'_j \text{ exists}\}]\!] \tag{9}$$

(7) is shown by the definition of $\mathcal{E}$. (8) and (9) are shown by the definition of $\mathcal{E}$ and theorem 10.

5.3 Semantics of Types

Types correspond to sets of expressions and expressions denote values in $\mathcal{D}$. Therefore types should be interpreted as subsets of $\mathcal{D}$. In order to give semantics to types, we should first determine what kind of subsets correspond to types. One such model of types was proposed by MacQueen, Plotkin and Sethi in [MPS86] where types were interpreted as *ideals* in $\mathcal{D}$. Cardelli used this ideal model to give semantics to a type system supporting records, variants and subtype relation (*inheritance*) [Car84]. However, the ideal

model is not suitable for our language; (i) it is not suitable for types for partial objects such as partial descriptions and (ii) the ordering on ideals does not agree with the ordering on our types.

To see (i) consider the expression $e = (Name \rightarrow 'J.\ Doe', Age \rightarrow 21, Emp\# \rightarrow 1234)$. This expression should have the type $\sigma = (Name : string, Age : int, Emp\# : int)$. If σ corresponds to a downward closed set of values, then an expression such as $(Name \rightarrow 'J.\ Doe')$ also has the type σ. Then the type system cannot eliminate expressions like $(Name \rightarrow 'J.\ Doe').Age$.

To see (ii) consider the two types $\sigma_1 = (Name : string, Sex : string)$ and $\sigma_2 = (Name : string, Emp\# : int)$. The lub of these two is $(Name : string)$. Then for their semantics we expect the following property should hold:

$$[\![\sigma_1]\!] \sqcup [\![\sigma_2]\!] = [\![(Name : string)]\!]$$

If we interpret types as ideals then $[\![\sigma_1]\!] \sqcup [\![\sigma_2]\!] = [\![\sigma_1]\!] \cup [\![\sigma_2]\!]$. However, the type $(Name : string)$ does not correspond to the union of σ_1 and σ_2. For example $(Name \rightarrow 'J.\ Doe', Sex \rightarrow 1, Emp\# \rightarrow 'ABC123')$ has the type $(Name : string)$ but has neither the type σ_1 nor σ_2. This problem arises even if a language does not contain partial values such as the language described in [Car84].

We regard types as subsets of values having common structures. As we have noted in section 2, subsets having common structures are upward closed sets. In addition to this, we also require that common structures are preserved by finite glb's, the intuition being that the glb $d \sqcap d'$ of two descriptions d, d' corresponds to the description common to d and d' and therefore has all structures common to d, d'. These observation lead us to define types as *filters* in $\mathcal{D}$ which do not contain w.

A non-empty subset $F \subseteq \mathcal{D}$ is a filter iff

1. F is upward closed; for any $d \in F, d \sqsubseteq d'$ implies $d' \in F$.
2. F is closed under pairwise glb; for any $d, d' \in F, d \sqcap d' \in F$.

If filter has a minimal element d them it is a *principal* filter and written as $d\uparrow$. Let $\mathcal{F}(\mathcal{D})$ denote the set of all filters in $\mathcal{D}$ that do not contain w. $\mathcal{F}(\mathcal{D})$ is ordered by set inclusion. Lub and glb are defined as:

1. $F \sqcup F' = \{d | \exists f \in F \exists f' \in F'.f \sqcap f' \sqsubseteq d\}$.
2. $F \sqcap F' = F \cap F'$.

Note that $F \sqcap F'$ dose not necessarily exist.

In order to interpret types in $\mathcal{F}(\mathcal{D})$, we define filter constructors corresponding to type constructors.

1. *Records.*
   Let $F_1, \ldots, F_n$ be filters in $\mathcal{D}$. Define $(l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n) = \{(l_1 \mapsto f_1, \ldots, l_m \mapsto f_m) | n \leq m, f_i \in F_i, 1 \leq i \leq n\}$.

   Prop. 11 *$(l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n)$ is a filter in $(L \rightarrow \mathcal{D})$.*

   Proof. It is clear that $(l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n)$ is upward closed. To see that this set is closed under pairwise glb, we note that glb in $(L \rightarrow \mathcal{D})$ is pointwise.□
   From the definition, we have:

   Prop. 12

   $$(l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n) \sqsubseteq (l_1 \Rightarrow F'_1, \ldots, l_m \Rightarrow F'_m) \text{ if } m \leq n, F_i \sqsubseteq F'_i, 1 \leq i \leq m$$

2. *Sets.*
   Let $F_1, \ldots, F_m$ be filters in $\mathcal{D}$. Define $[F_1, \ldots, F_m] = \{[f_1, \ldots, f_k] | \forall f \in \{f_1, \ldots, f_k\} \exists F \in \{F_1, \ldots, F_m\}.f \in F\}$

   Prop. 13 *$[F_1, \ldots, F_m]$ is a filter in $\mathcal{P}(\mathcal{D})$.*

Proof. It is clear that $[F_1, \ldots, F_m]$ is upward closed. Let $[f_1, \ldots, f_k], [f'_1, \ldots, f'_l] \in [F_1, \ldots, F_m]$. Since $[f_1, \ldots, f_k] \sqcap [f'_1, \ldots, f'_l] = [f_1, \ldots, f_k, f'_1, \ldots, f'_l]$ and $[f_1, \ldots, f_k, f'_1, \ldots, f'_l] \in [F_1, \ldots, F_m]$, $[F_1, \ldots, F_m]$ is closed under pairwise glb.□
From the definition, we have:

Prop. 14 *(a)* $[F_1, \ldots, F_n] \sqsubseteq [F'_1, \ldots, F'_m]$ *if* $\forall F \in \{F_1, \ldots, F_n\} \exists F' \in \{F'_1, \ldots, F'_m\}. F \sqsubseteq F'$.
(b) $[F_1, F_2, F_3, \ldots, F_n] = [F_1, F_3, \ldots, F_n]$ *if* $F_2 \sqsubseteq F_1$.
(c) $[F_1, \ldots, F_n] \sqcap [F'_1, \ldots, F'_m] = [F_i \sqcap F'_j | 1 \leq i \leq n, 1 \leq j \leq m, F_i \sqcap F_j$ *exists*$]$ *where we define* $[] = \{\emptyset\}$.

We now give semantics to types by the semantic function $\mathcal{T} : Texp \to \mathcal{F}(\mathcal{D})$ where $Texp$ is the set of types defined in the previous section:

$$\begin{aligned} \mathcal{T}[\![\tau_i]\!] &= \mathcal{B}_i \\ \mathcal{T}[\![(l_1 : \sigma_1, \ldots, l_n : \sigma_n)]\!] &= (l_1 \Rightarrow \mathcal{T}[\![\sigma_1]\!], \ldots, l_n \Rightarrow \mathcal{T}[\![\sigma_n]\!]) \\ \mathcal{T}[\![\{\sigma_1, \ldots, \sigma_m\}]\!] &= [\mathcal{T}[\![\sigma_1]\!], \ldots, \mathcal{T}[\![\sigma_m]\!]] \\ \mathcal{T}[\![\sigma \wedge \sigma']\!] &= \mathcal{T}[\![\sigma]\!] \sqcap \mathcal{T}[\![\sigma']\!] \end{aligned}$$

Since we interpret the domain constructor + as the separated sum domain constructor, $\mathcal{B}_i = (\perp_{B_i}) \uparrow$ and is a filter in $\mathcal{D}$ not containing w. Then by propositions 11 and 13 it is immediately seen that $\mathcal{T}$ is well defined. Proposition 14 shows that equations between types are sound with respect to this semantics. Using propositions 12 and 14, we can show the soundness of ordering on types by induction on the structure of types:

Theorem 15 *1. If* $\sigma \preceq \sigma'$ *then* $\mathcal{T}[\![\sigma]\!] \sqsubseteq \mathcal{T}[\![\sigma']\!]$.

2. If $\sigma \sqcup \sigma'$ *exists then* $\mathcal{T}[\![\sigma \sqcup \sigma']\!] = \mathcal{T}[\![\sigma]\!] \sqcup \mathcal{T}[\![\sigma']\!]$.

3. If $\sigma \sqcap \sigma'$ *exists then* $\mathcal{T}[\![\sigma \sqcap \sigma']\!] = \mathcal{T}[\![\sigma]\!] \sqcap \mathcal{T}[\![\sigma']\!]$.

From this theorem, we see that the filter model is an appropriate model for our type system. It should be also noted that the filter model can give precise semantics to multiple inheritance. The problem of join types we have mentioned in the beginning of section 5 does not arise in this model.

Based on the semantics of types, we show that the type inference system is correct.

Theorem 16 $\vdash e : \sigma$ *implies* $\mathcal{E}[\![e]\!] \in \mathcal{T}[\![\sigma]\!]$.

Proof. By induction on the hight of the proof tree for $\vdash e : \sigma$□.

Since our typechecking function is syntactically sound (theorem 9) with respect to the type inference system, we also get:

Corollary 17 *If* e *is well typed expression (type*$(e) \neq error$*) then* $\mathcal{E}[\![e]\!] \neq w$.

6 Conclusion and Future Work

By interpreting database objects as descriptions of real-world objects ordered by the goodness of descriptions, we have shown that non-flat records, higher-order relations, and natural joins can be represented as typed expressions in programming languages. We have then defined a simple typed language supporting these data structures and have presented a denotational semantics of the language. In order to give semantics to types, we have proposed a filter model of types. Using this model we have shown that the type system of the language is sound.

In order to develop a practical programming language based on this study, we need to extend the language to include function expressions. One simple way to do this extension is to define a new language using our language. Let e, σ denote expressions and types defined in the previous section. Then a syntax of the extended expressions (ranged over by E) can be given as:

$$E \ ::= \ x(variable) \mid e \mid \lambda x : \Sigma.E \mid EE$$

with the extended types:

$$\Sigma \ ::= \ \sigma \ | \Sigma \to \Sigma.$$

The extended expressions can be interpreted in a domain satisfying:

$$\mathcal{V} = \mathcal{D} + (\mathcal{V} \to \mathcal{V})$$

where $\mathcal{D}$ is a domain satisfying (6) and $\to$ is the continuous function space constructor. Since the space of filters of a domain is closed under the following function type constructor,

$$F \Rightarrow F' = \{f \in (\mathcal{D} \to \mathcal{D}) | \forall x \in F. f(x) \in F'\}$$

a semantics of the extended types can be also given.

We hope that this study provides a basis to implement typed programming languages for partial objects, including languages for databases, knowledge representations, and natural language processing.

Acknowledgements

This work is a continuation of [BO86, BJO91]. I would like to thank Peter Buneman for discussions and suggestions.

References

[AB84] S. Abiteboul and N. Bidoit. Non first normal form relations to represent hierarchically organized data. In *Proc. ACM Symposium on Principles of Database Systems*, Waterloo, Ontario, Canada, 1984.

[Bar84] H.P. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984. revised edition.

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

[BK86] F. Bancilhon and S. Khoshafin. A calculus for complex objects. In *Proc. ACM Symposium on Principles of Database Systems*, 1986.

[BO86] P. Buneman and A. Ohori. A domain theoretic approach to higher-order relations. In *International Conference on Database Theory, Lecture Notes in Computer Science 243*, pages 91 – 104, Rome, Italy, 1986. Springer-Verlag.

[Car84] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types, Lecture Notes in Computer Science 173*. Springer-Verlag, 1984.

[FT83] P.C. Fischer and S.J. Thomas. Operators for non-first-normal-form relations. In *Proc. IEEE COMPSAC*, 1983.

[MPS86] D.B. MacQueen, G.D. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, 1986.

[ÖY85] Z. Özsoyoğlu and L. Yuan. A normal form for nested relations. In *Proc. ACM Symposium on Principles of Database Systems*, pages 251–260, Portland, March 1985.

[RKS85] M.A. Roth, H.F. Korth, and A. Silberschatz. Null values in ¬1nf relational databases. Technical Report TR-85-32, Department of Computer Sciences, The University of Texas at Austin, December 1985. To appear in ACM Trans. on Database Systems.

[Sch86] D.A. Schmidt. *Denotational Semantics, A Methodology for Language Development*. Allyn and Bacon, 1986.

[Shi85] S.M. Shieber. An introduction to unification-based approaches to grammar. In *Proc. 23rd Annual Meeting of the Association for Computational Linguistics*, 1985.

[Smy78] M.B. Smyth. Power domains. *Journal of Computer and System Sciences*, 16(1):23–36, 1978.

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