Semantics Of Types For Database Objects

MS-CIS-89-15

Atsushi Ohori

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

February 1989

Acknowledgements:
This work was supported by grants from the Army Research Office grants DAA29-84-K-0061, DAA29-84-9-0027. The National Science Foundation IRI86-10617 and AT&T's Telecommunication Program at The University of Pennsylvania. To appear in *Journal of Theoretical Computer Science*.(Special issue dedicated to Second International Conference on "Database Theory").

Semantics of Types for Database Objects*

Atsushi Ohori

Department of Computer and Information Science,
University of Pennsylvania,
Philadelphia, PA 19104-6389, U.S.A.

Abstract

This paper proposes a framework of denotational semantics of database type systems and constructs a type system for complex database objects. Starting with an abstract analysis of the relational model, we develop a mathematical theory for the structures of domains of database objects. Based on this framework, we construct a concrete database type system and its semantic domain. The type system allows arbitrarily complex structures that can be constructed using labeled records, labeled variants, finite sets and recursion. On the semantic domain, in addition to standard operations on records, variants and sets, a *join* and a *projection* are available as polymorphically typed computable functions on arbitrarily complex objects. We then show that both the type system and the semantic domain can be uniformly integrated in an ML-like programming language. This leads us to develop a database programming language that supports rich data structures and powerful operations for databases while enjoying desirable features of modern type systems of programming languages including strong static type-checking, static type inference and ML polymorphism.

1 Introduction

There have been a number of attempts to develop data models to represent complex database objects beyond the first-normal-form relational model. Examples include nested relations [22, 48, 46] and complex object models [31, 8, 2]. (See also [29] for a survey.) However, these complex data structures and associated database operations have not been well integrated in a modern type system of a programming language, creating the problem known as "impedance mismatch" [39, 7]. As a result, database programming cannot share the benefits of recent developments in type theories of programming languages such as *static type inference* [40, 20] and *polymorphism* [40, 47], which should have had apparent practical benefits for many database applications. The problem is seen by simply noting that any existing *polymorphic* type system cannot represent even the relational model – perhaps the simplest form of a "complex object" model. As pointed out in [6], no existing type system can type-check a polymorphic *natural join* operation. Several languages

*This research was supported in part by grants NSF IRI86-10617, ARO DAA6-29-84-k-0061, and by funding from AT&T's Telecommunications Program at the University of Pennsylvania and from OKI Electric Industry Co., Japan.

have been proposed to integrate database structures into a programming language [52, 4, 5, 17, 16, 42]. (See also [6] for a survey.) However, their type systems are either dynamic or rather limited and do not incorporate static type inference nor polymorphism.

The author believes that the major source of this mismatch problem is the poor understanding of the properties of types for databases and the structures of domains of database objects. Traditionally, the theory of types of programming languages has been focussed on function types and domains of functions. Neither the properties of database type systems nor their relationship to type systems of programming languages have been well investigated. The goal of this paper is to construct a theory of database type systems that will serve as a "bridge" between complex data models and type systems of programming languages and to propose a concrete database type system that is rich enough to represent a wide range of complex database objects. These should enable us to develop a strongly typed database programming language that supports rich data structures and powerful operations for databases while enjoying desirable features of modern type systems of programming languages including static type inference and ML polymorphism.

As suggested by Cardelli [14], one way to represent complex objects in a programming language is to use labeled records and labeled disjoint unions (or labeled variants) found in many programming languages such as Pascal, Standard ML [25], Amber [15] and Galileo [4]. The following is an example of a labeled record expression:

$$[Name = [Firstname = "Joe", Lastname = "Doe"], Dept = "Sales", Office = 278]$$

Types for expressions can be easily defined. For example, the above record is given the following type:

$$[Name : [Firstname : string, Lastname : string], Dept : string, Office : int]$$

Tuples in the relational model are regarded as labeled records that contain only atomic values. In programming languages, these data structures are inductively defined allowing arbitrarily nested structures. Some languages also support recursively defined types and expressions. On these complex expressions, various operations are available. Assuming computable equality on each atomic type, equality on expressions that do not contain functions is computable and it is not hard to introduce set expressions on those complex expressions. A database of complex objects could then be represented as a set of these complex expressions.

An obvious problem of this approach is that, in practice, both expressions and sets become very large and contain a great deal of redundancy. This problem is elegantly solved in the relational model by the introduction of the two operations the (*natural*) *join* and the *projection*. Instead of representing a database as one large set (relation) of large tuples, we can first project it onto various small relations and then represent a database as a collections of those small relations. Larger relations are obtained by joining these small relations when needed. In order to support complex database objects in a programming language, it is therefore essential to support a join and a projection on complex expressions. We further believe that properly generalized join and projection together with standard operations on complex expressions form a sufficiently rich set of operations for complex database objects. Furthermore, integration of them into a modern type system of a programming language yields a database programming language in which databases are directly representable as typed data structures and a powerful set of operations are available as typed polymorphic functions. Such a programming language should be also suitable for other data intensive applications such as natural language processing and knowledge representation. We therefore hope that the

integration should also contribute to solve the "high-level" impedance mismatch between database systems and other applications.

The join and the projection in the relational model are based on the underlying operations that compute a join of tuples and a projection of a tuple. By regarding tuples as *partial descriptions* of real-world entities, we can characterize these operations as special cases of very general operations on partial descriptions; the one that *combines* two consistent descriptions and the one that *throws away* part of a given description. For example, if we consider the following non-flat tuples

$$t_1 = [Name = [Firstname = "Joe"]]$$

and

$$t_2 = [Name = [Lastname = "Doe"]]$$

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

$$t = [Name = [Firstname = "Joe", Lastname = "Doe"]]$$

Conversely, the tuple t_1 is considered as the result of the projection of the partial description t on the structure specified by the type

$$[Name : string, [Firstname : string]].$$

Operations that combine partial information also arise in other areas of applications. Examples include the "meet operation" on Aït-Kaci's ψ-terms [3] and the "unification operation" on *feature structures* representing linguistic information (see [55] for a survey).

Based on this general intuition, in this paper, we propose a framework of denotational semantics for database type systems and construct a concrete database type system and its semantic domain. The type system contains arbitrarily complex expressions definable by labeled records, labeled variants, finite sets and recursion. On its semantic domain, a join and a projection are defined as polymorphically typed computable functions. Furthermore, we carry out these construction in a completely effective way. In our framework, we require types and objects to be finitely representable and various properties to be effectively computable. This means that, once we have constructed the type system and its semantic domain based on our framework, it not only provides an uniform and elegant explanation of the properties of type system and the structures of domain of complex database objects, but it also provides representations and algorithms to integrate them into a practical programming language. Based on these results, an experimental programming language, Machiavelli [45], has been developed at University of Pennsylvania.

The rest of this paper is organized as follows. In section 2, we analyze the relational model as a typed data structure and extract the essence of the join and the projection. This analysis will also serve as an introduction to the subsequent abstract characterizations of database type systems and their semantic domains. Based on the analysis of the relational model, in section 3, we characterize the structures of type systems in which a polymorphic join and a polymorphic projection are definable and propose a framework for their semantic domains. In section 4, we define a concrete type system for complex database objects and construct its semantic domain. A part of the construction of the semantic domain (section 4.5) is based on the idea developed in [13] that a certain ordering on powerdomains can be used to generalize the relational join uniformly to complex objects and the idea due to Aït-Kaci [3] that a rich yet computationally feasible

domain of values is nicely represented by *labeled regular trees*. In revising this paper, the author also noticed that Rounds' recent work [49] achieves results similar to the ones presented in section 4.5 using a slightly different framework. Finally in section 5, we show that the type system and its semantic domain can be integrated in an ML-like programming language.

2 Analysis of the Relational Model

We first give a standard definition of the relational model. Since our purpose is to extract the essence of the type structure of the model, we define the model as a typed data structure. We also integrate *null values* in the model. The importance of null values has been widely recognized and several approaches have been proposed [9, 53, 34, 58]. Among them, we adopt the approach that null values represent *non-informative* values [58]. This approach fits well in our paradigm that database objects are partial descriptions and plays a crucial role in our theory of semantic domains of database type systems being developed in the next section.

Let $\mathcal{L}$ be a countably infinite set of labels. We assume that we are given a set $\mathcal{B}$ of base types and a set of atomic objects B_b for each $b \in \mathcal{B}$. For each base type b, we denote by $null_b$ the null value of the type b.

Definition 1 (Tuples and Relations) *A tuple type τ is a term of the form $[l_1 : b_1, \ldots, l_n : b_n]$ where $l_1, \ldots, l_n \in \mathcal{L}$ and $b_1 \ldots, b_n \in \mathcal{B}$. A tuple t of the tuple type $[l_1 : b_1, \ldots, l_n : b_n]$ is a term of the form $[l_1 = c_1, \ldots, l_n = c_n]$ such that $c_i \in B_{b_i}$ or $c_i = null_{b_i}, 1 \leq i \leq n$. A relation type (or relation scheme in the database literature) ρ is a term of the form $\{\!\{\tau\}\!\}$ for some tuple type τ. A relation instance r of the relation type $\{\!\{\tau\}\!\}$ is a term of the form $\{\!\{t_1, \ldots, t_n\}\!\}$ such that each $t_i, 1 \leq i \leq n$ is a tuple of the type τ.*

Regarding a tuple t as a function from a finite subset $L \subset \mathcal{L}$ to $\bigcup_{b \in \mathcal{B}} B_b \cup \{null_b | b \in \mathcal{B}\}$, we write $dom(t)$ for the set of labels in t and $t(l)$ for the value corresponding to the label l.

Relation instances are terms representing sets, for which the following equations hold:

$$\{\!\{t_1, \ldots, t_n\}\!\} = \{\!\{t_{i_1}, \ldots, t_{i_n}\}\!\} \text{ if } i_1, \ldots, i_n \text{ is a permutation of } 1, \ldots, n$$

and

$$\{\!\{t_1, t_2, t_3, \ldots\}\!\} = \{\!\{t_2, t_3, \ldots\}\!\} \text{ if } t_1 = t_2.$$

We consider relation instances as equivalence classes of the above equality. Under this equality, relation instances behave exactly like sets of tuples, on which ordinary set-theoretic operations are defined. Based on this fact, we treat relation instances as sets of tuples and apply ordinary set-theoretic notions directly to them. Readers might think that this strictly syntactic treatment only introduces (trivial but annoying) complication to structures that were simpler and more intuitive if we treated them just as sets. This had been true if we were only interested in sets of flat tuples. However, it is no longer possible to maintain such intuitive treatment when we allow *infinite* structures through recursion. Our syntactic treatment provides a uniform way to treat complex structures involving recursion.

Among the operations in the relational algebra, we only define the join and the projection. As we have argued, these two operations make the model a successful data model for databases. They also distinguish the model from standard type systems of programming languages. Two tuple types τ_1, τ_2 are *consistent* if

Name	Age	Salary
"*Joe Doe*"	21	21000
"*John Smith*"	$null_{int}$	34000

r_1

Name	Age	Office
"*Joe Doe*"	$null_{int}$	103
"*John Smith*"	$null_{int}$	278
"*Mary Jones*"	41	556

r_2

Name	Age	Salary	Office
"*Joe Doe*"	21	21000	103
"*John Smith*"	$null_{int}$	34000	278

$join(r_1, r_2)$

Figure 1: Join of Relations Containing Null Values

for all $l \in dom(\tau_1) \cap dom(\tau_2)$, $\tau_1(l) = \tau_2(l)$. Let τ_1, τ_2 be two consistent tuple types. Define $jointype(\tau_1, \tau_2)$ as the type τ such that $dom(\tau) = dom(\tau_1) \cup dom(\tau_2)$ and $\tau(l) = \tau_1(l)$ if $l \in dom(\tau_1)$ otherwise $\tau(l) = \tau_2(l)$. The two tuples t_1, t_2 are *consistent* if for all $l \in dom(t_1) \cap dom(t_2)$ one of the following hold: (1) $t_1(l) = t_2(l)$, (2) $t_1(l) = null_b$ and $t_2(l) \in B_b$ or (3) $t_1(l) \in B_b$ and $t_2(l) = null_b$. Two relation type $\{\!\{\tau_1\}\!\}, \{\!\{\tau_2\}\!\}$ are *consistent* if τ_1, τ_2 are consistent. For two consistent relation types $\{\!\{\tau_1\}\!\}, \{\!\{\tau_2\}\!\}$, define $jointype(\{\!\{\tau_1\}\!\}, \{\!\{\tau_2\}\!\})$ as the relation type $\{\!\{jointype(\tau_1, \tau_2)\}\!\}$.

Definition 2 (Relational Join) *Let t_1, t_2 be two consistent tuples of the respective types τ_1, τ_2. Then τ_1, τ_2 are also consistent. The join of t_1, t_2, $join(t_1, t_2)$, is the tuple t of the type $jointype(\tau_1, \tau_2)$ such that $dom(t) = dom(t_1) \cup dom(t_2)$, and $t(l) = t_1(l)$ if $l \in dom(t_1)$ and either $l \notin dom(t_2)$ or $t_2(l) = null_b$ otherwise $t(l) = t_2(l)$.*

Let $r_1 = \{\!\{t_1, \ldots, t_n\}\!\}, r_2 = \{\!\{t'_1, \ldots, t'_m\}\!\}$ be two relation instances having the consistent relation types ρ_1, ρ_2 respectively. The (natural) join of r_1, r_2, $join(r_1, r_2)$, is the relation instance r of the type $jointype(\rho_1, \rho_2)$ such that $r = \{\!\{t | \exists t_i \in r_1 \exists t_j \in r_2 . t_i, t_j$ are consistent $, t = join(t_i, t_j)\}\!\}$.

Definition 3 (Relational Projection) *Let $t = [l_1 = c_1, \ldots, l_n = c_n, \ldots]$ be a tuple of a type τ of the form $[l_1 : b_1, \ldots, l_n : b_n, \ldots]$. The* projection of t onto the type $\tau' = [l_1 : b_1, \ldots, l_n : b_n]$, *$project_{\tau'}(t)$, is the tuple $[l_1 = c_1, \ldots, l_n = c_n]$ of type τ'. Let r is a relation instance of the type $\{\!\{\tau\}\!\}$. The projection of r on the type $\{\!\{\tau'\}\!\}$, $project_{\{\!\{\tau'\}\!\}}(r)$, is the relation instance $\{\!\{project_{\tau'}(t) | t \in r\}\!\}$ of the type $\{\!\{\tau'\}\!\}$.*

When restricted to tuples without null values, it is clear that the above definitions are straightforward translations of standard definitions of the relational model found for example in [57, 21, 38]. The operation *join* is extended to relations containing null values. Figure 1 shows an example of a join of relations containing null values. Note that the definition of the join reflects the intended semantics of null values. The projection is specified by a type not just a set of labels. This will allow us to generalize the relational projection to complex structures.

These definitions apparently depend on the underlying structures of flat tuples. There are some efforts to generalize these operations beyond the first normal form relations [48, 1, 22, 32]. (See also [29] for a

survey.) However, their definitions still depend on the underlying tuple structures. Here, we would like to characterize the join and the projection operations independent of the underlying data structures so that we can generalize them uniformly to a wide range of complex data structures and introduce them to a type system of a programming language. Our guiding intuition is the idea exploited in [13] that database objects are *partial descriptions* of real-world entities and are *ordered* in terms of their "goodness" of descriptions. The idea of partial description was originally suggested by Lipski [35]. The corresponding ordered structure was first observed by Zaniolo [58] and is closely related to the ordering on ψ-terms [3] and finite state automata [50].

A *preorder* is a transitive reflexive relation. Let $(P, \leq)$ be a preordered set. Two elements $x, y \in P$ is *consistent* if there is some $z \in P$ such that $x \leq z$ and $y \leq z$. z is called an *upper bound* of x, y. (In what follows, we only need upper bounds of two elements and therefore we restrict the notion of upper bounds to upper bounds of two elements.) A *least upper bound* of x, y is an upper bound z of x, y such that $z \leq w$ for any upper bound w of x, y. A preordered set $(P, \leq)$ has the *pairwise bounded join property* if any two consistent elements has a least upper bound. A *partial order* is an antisymmetric preorder. In a partially ordered set (poset), least upper bounds are unique. We denote by $x \sqcup y$ the least upper bound of x, y (if exists). Any preordered set $(P, \leq)$ induces a poset, called the *quotient poset induced by* $(P, \leq)$, denoted by $[(P, \leq)]$: Let $\equiv$ be the equivalence relation on P defined as $x \equiv y$ iff $x \leq y$ and $y \leq x$. We denote by $[x]$ the equivalence class containing x. Define the set $P/\equiv$ as $\{[x] | x \in P\}$ and the relation $\leq/\equiv$ on $P/\equiv$ as $[x] \leq/\equiv [y]$ iff $x \leq y$. Then $[(P, \leq)]$ is the poset $(P/\equiv, \leq/\equiv)$. The following result is standard.

Lemma 1 *If $(P, \leq)$ is a preordered set with the pairwise bounded join property then $[(P, \leq)]$ is a poset with the pairwise bounded join property.*

For generality and simplicity, we treat tuples and relations uniformly. We call both tuple types and relation types as *flat description types* (ranged over by σ) and tuples and relation instances as *flat descriptions* (ranged over by d). For each flat description type σ, we write D_σ for the set of descriptions of the type σ. A flat description type represents a structure of descriptions. Such structures are naturally *ordered* to represent the intuition that one *contains* the other. For example, if $\sigma_1 = [Name : string, Age : int]$ and $\sigma_2 = [Name : string, Age : int, Office : int]$, then the structure represented by σ_2 contains the structure represented by σ_1. This intuitive idea is formalized by the following ordering:

Definition 4 (Ordering on Flat Description Types) *The* information ordering $\leq$ *on flat description types is the smallest relation satisfying:*

$$\begin{aligned} [l_1 : b_1, \ldots, l_n : b_n] &\leq [l_1 : b_1, \ldots, l_n : b_n, \ldots] \\ \{\!\{\tau_1\}\!\} &\leq \{\!\{\tau_2\}\!\} \text{ if } \tau_1 \leq \tau_2 \end{aligned}$$

Since the relation is based on the inclusion of fields of records, it is clear that it is a partial order. Moreover, this ordering has the following properties:

1. $\leq$ on the set of description types has the pairwise bounded join property, and
2. the ordering relation $\leq$ is decidable and least upper bounds (if they exist) are effectively computable.

The importance of this ordering is that it provides the following characterization of the types of the relational join and the relational projection:

Theorem 1 (Types of Relational Join and Projection) *Let d_1, d_2 be flat descriptions of the types σ_1, σ_2 respectively.*

1. *If $join(d_1, d_2)$ is defined and equal to d then $\sigma_1 \sqcup \sigma_2$ exists and d has the type $\sigma_1 \sqcup \sigma_2$.*
2. *If $project_\sigma(d_1)$ is defined and equal to d then $\sigma \leq \sigma_1$ and d has the type σ.*

Proof The property of *join* is an immediate consequence of the fact that $jointype(\sigma_1, \sigma_2)$ exists and equal to σ iff $\sigma_1 \sqcup \sigma_2$ exists and equal to σ. The property of *project* is an immediate consequence of the definition. ∎

We can then give the following type schemes (polymorphic types) to the join and the projection:

$$\begin{aligned} join &: (\sigma_1 \times \sigma_2) \rightarrow \sigma_1 \sqcup \sigma_2 \qquad \text{for all } \sigma_1, \sigma_2 \text{ such that } \sigma_1 \sqcup \sigma_2 \text{ exists} \\ project &: \sigma_1 \rightarrow \sigma_2 \qquad \text{for all } \sigma_1, \sigma_2 \text{ such that } \sigma_2 \leq \sigma_1 \end{aligned}$$

Since the ordering relation is decidable and least upper bounds are effectively computable, these type schemes allow us to type-check expressions containing joins and projections.

We next characterize these operations themselves using ordering on descriptions. Zaniolo observed [58] that the introduction of null values induces the following ordering on tuples:

$$[l_1 = x_1, \ldots, l_n = x_n] \sqsubseteq [l_1 = y_1, \ldots, l_n = y_n] \text{ iff either } x_i = null_b \text{ or } x_i = y_i, 1 \leq i \leq n$$

This ordering is interpreted as the ordering of "goodness" of descriptions. The following is an example of this ordering.

$$[Name = \text{"}Joe\ Doe\text{"}, Age = null_{int}] \sqsubseteq [Name = \text{"}Joe\ Doe\text{"}, Age = 21]$$

It is clear that for ant tuple type τ this ordering is a partial order on D_τ with the pairwise bounded join property. The join on tuples of a same type is characterized as the least upper bound operation under this ordering, which formalizes our intuition that the join is an operation that combines partial descriptions:

Proposition 1 (Join of Flat Tuples) *If $t_1, t_2 \in D_\tau$ then $join(t_1, t_2) = t$ iff $t_1 \sqcup t_2 = t$.*

Proof By definitions. ∎

For a relation type ρ, an appropriate ordering on D_ρ to characterize the join on D_ρ turns out to be the ordering known as *Smyth powerdomain ordering* [56]. To define the ordering, we first define a preorder $\preceq$:

$$\{\!\{t_1, \ldots, t_n\}\!\} \preceq \{\!\{t'_1, \ldots, t'_m\}\!\} \text{ if } \forall t'_j \in \{t'_1, \ldots, t'_m\} \exists t_i \in \{t_1, \ldots, t_n\}.\, t_i \sqsubseteq t'_j$$

The relation $\preceq$ is not antisymmetric. However, we can take the quotient poset induced by the preorder:

Proposition 2 *For any relation type ρ, $[(D_\rho, \preceq)]$ is a poset with the pairwise bounded join property.*

Proof $\preceq$ is clearly transitive and reflexive and therefore $(D_\rho, \preceq)$ is a preordered set. Let r_1 and r_2 be any elements in D_ρ under $\preceq$. Let $r = \{\!\{t | \exists t_i \in r_1 \exists t_j \in r_2.\ t_i, t_j \text{ are consistent}, t = join(t_i, t_j)\}\!\}$. Since $t_1 \sqcup t_2 = join(t_1, t_2)$, as a special case of the result shown in [56], r is a least upper bound of r_1 and r_2. Then the proposition follows from lemma 1. ∎

We regard a relation instance as a representative of the corresponding equivalence class induced by the above preorder and write $d_1 \sqcup d_2$ for the least upper bound of the corresponding equivalence classes. We also write $(D_\rho, \sqsubseteq)$ for $[(D_\rho, \preceq)]$. Readers are referred to [13] for the intuition and relevance of this ordering in various aspects of databases. [12, 49] also use this ordering in a context of partial information. For us, this ordering provides the following characterization of the join on relations shown in [13]:

Proposition 3 (Join of Flat Relations) *If $r_1, r_2 \in D_\rho$ then $join(r_1, r_2) = r$ iff $r_1 \sqcup r_2 = r$.*

In order to characterize joins of descriptions of different types and projections, we interpret the partially ordered space of flat description types by *coercions* between domains.

Definition 5 (Coercions between Relational Domains) *The set of up-coercions is the set of mappings $\{\phi_{\sigma_1 \to \sigma_2} | \sigma_1 \leq \sigma_2\}$ defined as*

1. *if $\sigma_1 = [l_1 : b_1, \ldots, l_n : b_n]$, $\sigma_2 = [l_1 : b_1, \ldots, l_n : b_n, l_{n+1} : b_{n+1}, \ldots, l_{n+m} : b_{n+m}]$ then*

$$\phi_{\sigma_1 \to \sigma_2}([l_1 = c_1, \ldots, l_n = c_n]) = [l_1 = c_1, \ldots, l_n = c_n, l_{n+1} = null_{b_{n+1}}, \ldots, l_{n+m} = null_{b_{n+m}}],$$

2. *if $\sigma_1 = \{\!\{\tau_1\}\!\}$, $\sigma_2 = \{\!\{\tau_2\}\!\}$ and $\tau_1 \leq \tau_2$ then*

$$\phi_{\sigma_1 \to \sigma_2}(r) = \{\!\{\phi_{\tau_1 \to \tau_2}(t) | t \in r\}\!\}.$$

The set of down-coercions is the set of mappings $\{\psi_{\sigma_1 \to \sigma_2} | \sigma_2 \leq \sigma_1\}$ defined as

1. *if $\sigma_1 = [l_1 : b_1, \ldots, l_n : b_n, \ldots]$ and $\sigma_2 = [l_1 : b_1, \ldots, l_n : b_n]$ then*

$$\psi_{\sigma_1 \to \sigma_2}([l_1 = c_1, \ldots, l_n = c_n, \ldots]) = [l_1 = c_1, \ldots, l_n = c_n],$$

2. *if $\sigma_1 = \{\!\{\tau_1\}\!\}$, $\sigma_2 = \{\!\{\tau_2\}\!\}$ then $\tau_2 \leq \tau_1$ and*

$$\psi_{\sigma_1 \to \sigma_2}(r) = \{\!\{\psi_{\tau_1 \to \tau_2}(t) | t \in r\}\!\}.$$

Intuitively, an up-coercion coerces a description to a description of a larger structure by "padding" extra part of structure with null values. A down-coercion on the other hand coerces a description to a description of a smaller structure by "throwing away" part of its structure. For example, if

$$\begin{aligned}
\tau_1 &= [Name : string, Age : int] \\
\tau_2 &= [Name : string, Office : int] \\
\tau_3 &= [Name : string, Age : int, Office : int] \\
t_1 &= [Name = "Joe", Age = 21] \\
t_2 &= [Name = "Joe", Office = 278] \\
t_3 &= [Name = "Joe", Age = 21, Office = 278]
\end{aligned}$$

then

$$\begin{aligned}
\phi_{\tau_1\to\tau_3}(t_1) &= [Name = \text{"}Joe\text{"}, Age = 21, Office = null_{int}] \\
\phi_{\tau_2\to\tau_3}(t_2) &= [Name = \text{"}Joe\text{"}, Age = null_{int}, Office = 278] \\
\psi_{\tau_3\to\tau_1}(t_3) &= t_1 \\
\psi_{\tau_3\to\tau_2}(t_3) &= t_2
\end{aligned}$$

We then have the following equations:

$$\begin{aligned}
join(t_1, t_2) &= \phi_{\tau_1\to\tau_3}(t_1) \sqcup_{D_{\tau_3}} \phi_{\tau_2\to\tau_3}(t_2) \\
project_{\tau_1}(t_3) &= \psi_{\tau_3\to\tau_1}(t_3) \\
project_{\tau_2}(t_3) &= \psi_{\tau_3\to\tau_2}(t_2)
\end{aligned}$$

This example suggests that computing a join of descriptions of types σ_1, σ_2 corresponds to coercing them to the type $\sigma_1 \sqcup \sigma_2$ followed by computing their least upper bound. The projections correspond to down-coercions. Indeed we have:

Theorem 2 (Relational Join and Projection) *Let d_1 and d_2 be any flat descriptions of types σ_1, σ_2 respectively. $join(d_1, d_2)$ exists and equal to d iff $\sigma_1 \sqcup \sigma_2$ exists and $d = \phi_{\sigma_1\to\sigma}(d_1) \sqcup_{D_\sigma} \phi_{\sigma_2\to\sigma}(d_2)$ where $\sigma = \sigma_1 \sqcup \sigma_2$. $project_\sigma(d_1)$ exists and equal to d iff $\sigma \leq \sigma_1$ and $d = \psi_{\sigma_1\to\sigma}(d_1)$*

Proof By the definitions of ϕ and *join*, for any d_1 of type σ_1 and d_2 of type σ_2 such that $\sigma_1 \sqcup \sigma_2$ exists and equal to σ, $join(d_1, d_2)$ exists and equal to d iff $join(\phi_{\sigma_1\to\sigma}(d_1), \phi_{\sigma_2\to\sigma}(d_2))$ exists and equal to d. Then the property of *join* follows from propositions 1 and 3. The property of projection is by definitions. ∎

The semantic space of the relational model is therefore characterized by the set

$$\{(D_\sigma, \sqsubseteq) | \sigma \text{ is a flat description type}\}$$

connected by the set of pairs of up- and down-coercions

$$\{(\phi_{\sigma_1\to\sigma_2}, \psi_{\sigma_2\to\sigma_1}) | \sigma_1 \leq \sigma_2\}.$$

associated with the set of join operations $\{join_{(\sigma_1\times\sigma_2)\to\sigma} | \sigma_1 \sqcup \sigma_2$ exists and equal to $\sigma\}$ defined as

$$join_{(\sigma_1\times\sigma_2)\to\sigma}(d_1, d_2) = \phi_{\sigma_1\to\sigma}(d_1) \sqcup_{D_\sigma} \phi_{\sigma_2\to\sigma}(d_2)$$

and the set of projection operations $\{project_{\sigma_1\to\sigma_2} | \sigma_2 \leq \sigma_2\}$ defined as

$$project_{\sigma_1\to\sigma_2}(d) = \psi_{\sigma_1\to\sigma_2}(d).$$

The importance of this characterization is that it applies to any set of domains on which we can define information orderings and appropriate sets of coercions. Based on this analysis, in the next section, we formally define the structures of type systems for databases and their semantic domains.

3 Database Domains

As a generalization of the set of flat description types in the relational model, we define a set of types for databases as follows:

Definition 6 (Database Type Systems) *A database type system is a poset of types $(T, \leq)$ such that*

1. *it has the pairwise bounded join property, and*
2. *the ordering relation and least upper bounds (if they exist) are effectively computable.*

We call each element of T a description type.

Each type represents a structure of descriptions and the ordering on types represents the containment ordering of the structures they represent. The pairwise bounded join condition is necessary for the types of joins to be well defined. The decidability conditions is necessary for effective type-checking.

Each description type should denote a domain of descriptions. As a generalization of domains of flat descriptions in the relational model, we require domains of descriptions to satisfy the following conditions:

Definition 7 (Description Domains) *A description domain is a poset $(D, \sqsubseteq)$ satisfying:*

1. *D has the bottom element $null_D$, i.e. for any $d \in D, null_D \sqsubseteq d$,*
2. *D has the pairwise bounded join property,*
3. *the ordering relation $\sqsubseteq$ is decidable and least upper bounds (if they exist) are effectively computable.*

Condition 1 allows us to represent non-informative value which is essential for partial descriptions. Condition 2 states that if we have two consistent descriptions then the combination of the two is also representable as a description. This is necessary for join to be well defined. The necessity of the condition 3 is obvious.

It should be noted that description domains are models of types of database objects and not models of general types in programming languages such as function types. In particular, they should not be confused with *Scott domains* [54] which is used to give semantics to untyped lambda calculus and programming languages with recursively defined functions [51]. Both notions share similar ordered structure and are based on a similar intuition that values are ordered in terms of "goodness of approximation". However, the properties of the two orderings are fundamentally different. The ordering on a description domain is just a computable predicate. On the other hand the *Scott ordering* is regarded as a predicate on the computability and in principle not computable.

By abstracting underlying tuple structures from the definition of up-coercions and down-coercions between relational domains, we interpret an ordering on description types by a special class of mappings between description domains. A function $f : D_1 \rightarrow D_2$ between description domains D_1, D_2 is *monotone* iff for any $x, y \in D_1$, $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$.

Definition 8 (Embeddings and Projections) *A monotone function $\phi : D_1 \rightarrow D_2$ is an embedding if there exists a function $\psi : D_2 \rightarrow D_1$ such that* (1) *for any $x \in D_2$, $\phi(\psi(x)) \sqsubseteq x$ and* (2) *for any $x \in D_1$, $\psi(\phi(x)) = x$. The function ψ is called a projection.*

A pair of embedding and projection is a special case of *Galois connections* (or *adjunctions*), for which the following result is well known [23]:

Lemma 2 *Given an embedding $\phi : D_1 \rightarrow D_2$, the corresponding projection is uniquely determined by ϕ.*

If ϕ is an embedding, we sometimes denote by ϕ^R the corresponding projection.

If a pair of description domains (D_1, D_2) has an embedding-projection pair $(\phi : D_1 \rightarrow D_2, \psi : D_2 \rightarrow D_1)$ then D_2 contains an isomorphic copy $D_1' = \phi(D_1)$ of D_1 and for any element d in D_2 there is a unique maximal element $d' \in D_1'$ such that $d' \sqsubseteq d$. We regard this property as the semantics of the ordering of description types. ϕ maps an element $d \in D_1$ to the least element $d' \in D_2$ such that d' contains all information in d. ψ maps an element $d \in D_2$ to a unique maximal element $d' \in D_1$ that contains only information in d and is regarded as a database projection from D_2 to D_1. The set of up-coercions we have defined on relational domains are indeed the set of embeddings between relational domains. The corresponding projections are exactly down-coercions.

Our characterization of the ordering on types can be regarded as a refinement of one of the characterizations of subtypes proposed by Bruce and Wegner [11], where the notion of subtypes is characterized in three ways; one of them being that the larger set contains an isomorphic copy of the smaller. It is also related to the notion of *information capacity* of data structures studied in [30] where the ordering on various data structures was defined by using mappings between the sets of objects.

Finally we define a semantic space of a database type system as a space of description domains partially ordered by a set of embedding-projection pairs.

Definition 9 (Database Domains) *A database domain is a pair (Dom, Emb) of a set of description domains Dom and a set of embeddings Emb between Dom satisfying the following conditions:*

1. *For any two domains $D_1, D_2 \in Dom$, there is at most one $\phi \in Emb$ such that $\phi : D_1 \rightarrow D_2$. We write $\phi_{D_1 \rightarrow D_2}$ for an embedding of type $D_1 \rightarrow D_2$.*
2. *For any domain $D \in Dom$, $\phi_{D \rightarrow D} \in Emb$.*
3. *Emb is closed under composition.*
4. *For any two domains $D_1, D_2 \in Dom$, if there is some $D \in Dom$ such that $\phi_{D_1 \rightarrow D} \in Emb$ and $\phi_{D_2 \rightarrow D} \in Emb$ then there is a unique $D' \in Dom$ depending only on D_1, D_2 such that $\phi_{D_1 \rightarrow D'} \in Emb$, $\phi_{D_2 \rightarrow D'} \in Emb$ and for any $D'' \in Dom$ if $\phi_{D_1 \rightarrow D''} \in Emb$ and $\phi_{D_2 \rightarrow D''} \in Emb$ then $\phi_{D' \rightarrow D''} \in Emb$.*
5. *For any $\phi \in Emb$, both ϕ and ϕ^R are computable, i.e. there is an algorithm to compute $\phi(d)$ and $\phi^R(d')$ for any given $d \in dom(\phi)$ and $d' \in dom(\phi^R)$.*

The condition 1 means that the set of embeddings defines a relation on Dom. Moreover,

Proposition 4 *The relation defined by Emb is a partial order with the pairwise bounded join property.*

Proof From the condition 2 and 3, the relation is reflexive and transitive. For anti-symmetricity, suppose $\phi_{X\to Y} \in Emb$ and $\phi_{Y\to X} \in Emb$ for some $X, Y \in Dom$. Since $\phi_{X\to X} \in Emb$ and $\phi_{Y\to Y} \in Emb$, the uniqueness of D' in the condition 4 implies $X = Y$. The pairwise bounded join property is an immediate consequence of the condition 4. ∎

Definition 10 (Models of Database Type Systems) *Let $(T, \leq)$ be a database type system. A database domain (Dom, Emb) is a model of $(T, \leq)$ if there is a mapping $\mu : T \to Dom$ such that for any $\tau_1, \tau_2 \in T$, $\tau_1 \leq \tau_2$ iff $\phi_{\mu(\tau_1)\to\mu(\tau_2)} \in Emb$.*

Remember that on description domains we imposed the conditions that the ordering is decidable and least upper bounds are computable. Combined with the computability condition on embeddings and projections, they guarantee that the join and the projection defined as

$$join_{(\sigma_1\times\sigma_2)\to\sigma}(d_1, d_2) = \phi_{\sigma_1\to\sigma}(d_1) \sqcup_{D_\sigma} \phi_{\sigma_2\to\sigma}(d_2) \quad (1)$$

$$project_{\sigma_1\to\sigma_2}(d) = \psi_{\sigma_1\to\sigma_2}(d) \quad (2)$$

are always computable functions. This means that if a database type system has a model, then the join and the projection are available as computable functions with the following polymorphic types:

$$join : (\sigma_1 \times \sigma_2) \to \sigma_1 \sqcup \sigma_2 \quad \text{for all } \sigma_1, \sigma_2 \text{ such that } \sigma_1 \sqcup \sigma_2 \text{ exists} \quad (3)$$

$$project : \sigma_1 \to \sigma_2 \quad \text{for all } \sigma_1, \sigma_2 \text{ such that } \sigma_1 \leq \sigma_2 \quad (4)$$

The relational join and the relational projection are special cases of the above functions on flat tuple structures. Moreover, from the previous results, we have:

Theorem 3 *The set of flat description types with the information ordering $\leq$ is a database type system. The pair of the set of relational domains and the set of up-coercions*

$$(\{(D_\sigma, \sqsubseteq) | \sigma \text{ is a flat description type}\}, \{\phi_{\sigma_1\to\sigma_2} | \sigma_1 \leq \sigma_2\})$$

is a database domain and a model of the poset of flat description types.

We therefore claim that the notions of database type systems and database domains are a proper generalization of the relational model.

The advantage of this characterization is that it is independent of the actual structures of types and objects. This allows us to generalize the relational model to wide range of structures, even those that include recursively defined types and objects. In the next section we construct a database type system and its database domain, which we believe is rich enough to cover virtually all proposed representations of complex database objects.

4 A Type System for Complex Database Objects

In addition to finite structures representable by finite terms, we would like to allow recursively defined structures, which naturally emerge in descriptions of real-word entities. As demonstrated by Aït-Kaci [3],

an appropriate formalism to represent these structures are *regular trees*, which provides a sufficiently rich yet computationally feasible framework for complex data structures. We therefore develop our type system and its domain using regular trees. However, this generality creates a slight technical complication that we cannot use inductive method to define structures and to prove properties. This may yield less intuitive definitions and might decrease the readability of the rest of the paper. In order to prevent the situation, for major definitions and properties, we give equivalent inductive characterizations on finite trees. They will not be used in the subsequent development and we shall omit the proofs of their equivalence to the original definitions restricted to finite trees. They can be proved by usual structural induction.

4.1 Labeled Regular Trees

We gather definitions and standard results on regular trees. Main references on this subject are [19, 18].

Let A be a set of symbols. The set of all strings (finite sequences of symbols) over A is denoted by A^*. The length of a string $a \in A^*$ is denoted by $|a|$. The empty string ϵ is the string of length 0. The concatenation of $a, b \in A^*$ is denoted by $a \cdot b$. A string a is a *prefix* of a string b if there is some c such that $a = b \cdot c$. A prefix a of b is *proper* if $a \neq b$. For $X \subseteq A^*$ and $Y \subseteq A^*$, $X \cdot Y$ is the set $\{x \cdot y | x \in X, y \in Y\}$. We write $x \cdot Y$ for $\{x\} \cdot Y$ and $X \cdot y$ for $X \cdot \{y\}$. For $a \in A^*$ and $X \subseteq A^*$, X/a is the set $\{b | \exists c \in X$ such that $c = a \cdot b\}$. We identify an element $a \in A$ and the corresponding string a of length one.

Instead of using a standard representation of trees based on fixed arity function symbols with ordered arguments, we use *labeled trees* whose node are labeled with function symbols and whose edges are labeled with elements in $\mathcal{L}$ indicating their arguments. This is a generalization of labeled record structures and is particularly suitable for representing complex structures including recursively defined ones. The following definition is due to [3].

Definition 11 (Labeled Trees) *Let $\boldsymbol{F}$ be a (not necessarily finite) set of symbols. A labeled $\boldsymbol{F}$-tree is a function $\alpha : L \to \boldsymbol{F}$ such that L is a prefix-closed subset of $\mathcal{L}^*$, i.e. for any $a, b \in \mathcal{L}^*$, if $a \cdot b \in L$ then $a \in L$. A tree α is finite if its domain $dom(\alpha)$ is finite otherwise it is infinite. The set of all $\boldsymbol{F}$-trees and the set of all finite $\boldsymbol{F}$-trees are denoted respectively by $\boldsymbol{T}^\infty(\boldsymbol{F})$ and $\boldsymbol{T}(\boldsymbol{F})$.*

Note that we do not impose the *arity restriction* on function symbols. However, we can regard each function symbol $f \in \boldsymbol{F}$ as the set of symbols $\{f_{\{l_1,\ldots,l_n\}} | l_1, \ldots, l_n \in \mathcal{L}\}$ indexed by finite sets of labels. By assuming a total order $\ll$ on $\mathcal{L}$, we can then regard our definition of trees as a notational variant of the standard representation of trees found in [19, 18] based on the *tree domains* [24]. We omit formal treatment of the connection.

For any element $f \in \boldsymbol{F}$, we also denote by f the one node tree such that $dom(f) = \{\epsilon\}$ and $f(\epsilon) = f$. Let $\alpha_1, \ldots, \alpha_n \in \boldsymbol{T}^\infty(\boldsymbol{F})$, $l_1, \ldots, l_n \in \mathcal{L}$ and $f \in \boldsymbol{F}$. We write $f(l_1 = \alpha_1, \ldots, l_n = \alpha_n)$ to denote the tree α such that $dom(\alpha) = l_1 \cdot dom(\alpha_1) \cup \cdots \cup l_n \cdot dom(\alpha_n)$, $\alpha(\epsilon) = f$, $\alpha(l_i \cdot a) = \alpha_i(a)$ for all $a \in dom(\alpha_i)$, $1 \leq i \leq n$. If $\alpha \in \boldsymbol{T}^\infty(\boldsymbol{F})$ and $a \in dom(\alpha)$ then the *subtree at* a in α, denoted by α/a, is the tree α' such that $dom(\alpha') = dom(\alpha)/a$, and for all $b \in dom(\alpha')$, $\alpha'(b) = \alpha(a \cdot b)$. The set of all subtrees of a tree α is the set $\boldsymbol{Subtrees}(\alpha) = \{\alpha/a | a \in dom(\alpha)\}$.

Definition 12 (Regular Trees) *A tree $\alpha \in \boldsymbol{T}^{\infty}(\boldsymbol{F})$ is regular iff the set $\boldsymbol{Subtrees}(\alpha)$ is finite. The set of all regular trees in $\boldsymbol{T}^{\infty}(\boldsymbol{F})$ is denoted by $\boldsymbol{R}(\boldsymbol{F})$.*

Intuitively, regular trees are trees that have a finite representation. There are several equivalent representations of regular trees. Following [3], we use Moore machines to represent them.

Definition 13 (Moore Machine) *A More machine is a 5-tuple $(Q, s, \boldsymbol{F}, \delta, \lambda)$, where Q is a set of states, s is a distinguished element in Q called the start state, $\boldsymbol{F}$ is the set of output symbols, δ is a partial function from $Q \times \mathcal{L}$ to Q called the state transition function such that for any $q \in Q$, $\{l \in \mathcal{L} | \delta(q, l)$ is defined$\}$ is finite and λ is the output function from Q to $\boldsymbol{F}$.*

In the above definition, the input alphabet is implicitly assumed as the fixed set $\mathcal{L}$ of labels. Because of the restriction on δ, a Moore machine under the above definition behaves like a Moore machine under a standard definition where the input alphabet $\mathcal{L}$ is finite and δ is defined as a total function on $Q \times \mathcal{L}$. As is done in standard finite state automata [26], we extend δ to δ^* on $Q \times \mathcal{L}^*$. A state $q \in Q$ is *reachable* if there is some $a \in \mathcal{L}^*$ such that $\delta^*(s, a) = q$. Each state $q \in Q$ of a Moore machine $M = (Q, s, \boldsymbol{F}, \delta, \lambda)$ represents a function form a prefix-closed subset of $\mathcal{L}^*$ to $\boldsymbol{F}$. Define $M(q)$ as the function such that $dom(M(q)) = \{a \in \mathcal{L} | \delta^*(q, a) = q$ for some $q' \in Q\}$ and $M(q)(a) = \lambda(\delta^*(q, a))$ for all $a \in dom(M)$.

The following theorem establishes the relationship between Moore machines and regular trees, which is essentially same as the equivalence of regular trees and *regular systems* shown in [19]. The proof can be easily reconstructed from the corresponding proof.

Theorem 4 *For any Moore machine $M = (Q, s, \boldsymbol{F}, \delta, \lambda)$, $M(s) \in \boldsymbol{R}(\boldsymbol{F})$. Conversely, for any regular tree $\alpha \in \boldsymbol{R}(\boldsymbol{F})$ there is a Moore machine $M = (Q, s, \boldsymbol{F}, \delta, \lambda)$ such that $\alpha = M(s)$.*

We say that a regular tree α *is represented by a Moore machine* M if $M(s) = \alpha$.

We use the following term language to represent regular trees via Moore machines:

$$e ::= s \mid f \mid f(l = e, \ldots, l = e) \mid (rec\ s.\, e)$$

where f stands for $\boldsymbol{F}$, l stands for $\mathcal{L}$ and s stands for the set of *state variables* disjoint from other symbols. The state variables are bound variables similar to those in lambda calculi. A term e is *proper* if a state variable occurrence s is either an occurrence of the form $rec\ s$ or in some e' in $(rec\ s.\, e')$.

For a proper term e, define the Moore machine $M_e = (Q, s, \boldsymbol{F}, \delta, \lambda)$ as:

1. $Q = \{q_f |$ for each *occurrence* $f \in \boldsymbol{F}$ in $e\}$,
2. $s = q_f$ where f is the outmost occurrence of output symbol in e,
3. $\delta(q_f, l) = q_g$ iff either f, g are the occurrences in a subterm of the form $f(\ldots, l = g(\ldots), \ldots)$ or f, g are the occurrences in a subterm of the form $(rec\ s.\, g(\ldots f(\ldots, l = s, \ldots)\ldots))$ such that it is the smallest subterm of the form $(rec\ s.\ \cdots)$ surrounding $f(\ldots, l = s, \ldots)$.
4. $\lambda(q_f) = f$.

The regular tree *represented by* a proper term e is then defined as $M_e(s)$. It can be also shown that for any regular tree α there is a proper term e that represents α.

For a technical convenience we assume that the set of labels $\mathcal{L}$ is closed under products, i.e. there is a injective function *prodcode* : $(\mathcal{L} \times \mathcal{L}) \rightarrow \mathcal{L}$. For any given set of labels, we can construct a set satisfying this condition. We use *prodcode* implicitly and treat $\mathcal{L}$ as the set satisfying $\mathcal{L} \times \mathcal{L} \subseteq \mathcal{L}$. In particular $(\mathcal{L} \times \mathcal{L})^* \in \mathcal{L}^*$. On $(\mathcal{L} \times \mathcal{L})^*$ we define the mappings *first*, *second* inductively as follows:

$$\begin{aligned} \mathit{first}(\epsilon) &= \epsilon \\ \mathit{first}(a \cdot (l_1, l_2)) &= \mathit{first}(a) \cdot l_1 \\ \mathit{second}(\epsilon) &= \epsilon \\ \mathit{second}(a \cdot (l_1, l_2)) &= \mathit{second}(a) \cdot l_2 \end{aligned}$$

On $\{(a,b) | a \in \mathcal{L}^*, b \in \mathcal{L}^*, |a| = |b|\}$, we define *pair* as follows:

$$\begin{aligned} \mathit{pair}(\epsilon, \epsilon) &= \epsilon \\ \mathit{pair}(a \cdot l_1, b \cdot l_2) &= \mathit{pair}(a,b) \cdot (l_1, l_2) \end{aligned}$$

For $a \in \mathcal{L}^* \times \mathcal{L}^*$, the following equation always holds:

$$\mathit{pair}(\mathit{first}(a), \mathit{second}(a)) = a$$

Let r be a relation on $\mathcal{L}$. The *extension of* r *on* $\mathcal{L}^*$, denoted by $\hat{r}$, is the relation defined as:

$$\begin{aligned} \epsilon \quad \hat{r} \quad & \epsilon \\ a \cdot l_1 \quad \hat{r} \quad & a \cdot l_2 \text{ if } l_1 \, r \, l_2 \end{aligned}$$

The following construction on Moore machines, which "traces" two Moore machines in "parallel", is often useful to determine various relations on regular trees. This can be regarded as a generalization of the *merged transaction function* used to determine the equivalence of two finite state machines in [27]. The new symbol \$ introduced below represents a "rejecting state" in a standard representation.

Definition 14 (Product Machine) *Let $\sim$ be any equivalence relation on $\mathcal{L}$. Given two Moore machines $M_1 = (Q_1, s_1, \boldsymbol{F}_1, \delta_1, \lambda_1)$ and $M_2 = (Q_2, s_2 \boldsymbol{F}_2, \delta_2, \lambda_2)$, the product machine of M_1 and M_2 modulo $\sim$, write $(M_1 \times M_2)/\!\sim$, is the Moore machine $(Q, s, \boldsymbol{F}, \delta, \lambda)$ such that*

1. $Q = (Q_1 \cup \{\$\}) \times (Q_2 \cup \{\$\})$ *where \$ is a new distinguished symbol that does not appear both in M_1 and M_2,*
2. $s = (s_1, s_2)$,
3. $\boldsymbol{F} = (\boldsymbol{F}_1 \cup \{\$\}) \times (\boldsymbol{F}_2 \cup \{\$\})$
4. $\delta((x,y), l)$ *is defined and equal to (x', y') iff one of the following holds:*
   (a) $l = (l_1, l_2)$, $l_1 \neq \$, l_2 \neq \$$, $l_1 \sim l_2$, $x \in Q_1, y \in Q_2$, *and* $x' = \delta_1(x, l_1), y' = \delta_2(y, l_2)$,

(b) $l = (l_1, l_2)$, $l_1 \neq \$, l_2 = \$$, $x \in Q_1$, *either there is no* l' *such that* $\delta_2(y, l')$ *is defined and* $l_1 \sim l'$ *or* $y = \$$, *and* $x' = \delta_1(x, l_1)$, $y' = \$$,

(c) $l = (l_1, l_2)$, $l_1 = \$, l_2 \neq \$$, $y \in Q_2$, *either there is no* l' *such that* $\delta_1(x, l')$ *is defined and* $l_2 \sim l'$ *or* $x = \$$, *and* $x' = \$$, $y' = \delta_2(y, l_2)$.

5. $\lambda((x_1, x_2)) = (o_1, o_2)$ *where* $o_i = \lambda_i(x_i)$ *if* $x_i \in Q_i$ *otherwise* $o_i = \$$, $i \in \{1, 2\}$.

If $\sim$ *is the identity relation* $=$ *on* $\mathcal{L}$ *then we write* $M_1 \times M_2$ *for* $(M_1 \times M_2)/=$.

The construction of a product machine is clearly effective. The following properties are also immediate consequences of the definition:

Lemma 3 *Let* $M_1 = (Q_1, s_1, \boldsymbol{F}_1, \delta_1, \lambda_1)$, $M_2 = (Q_2, s_2, \boldsymbol{F}_2, \delta_2, \lambda_2)$ *and* $(Q, s, \boldsymbol{F}, \delta, \lambda) = (M_1 \times M_2)/\sim$.

1. *If* $\delta^*(s, a) = (q_1, q_2), q_1 \in Q_1, q_2 \in Q_2$ *then* $first(a) \hat{\sim} second(a)$ *and* $\delta_1^*(s_1, first(a)) = q_1$, $\delta_2^*(s_2, second(a)) = q_2$. *Conversely, if there are* a, b *such that* $a \hat{\sim} b$, $\delta_1^*(s_1, a) = q_1$ *and* $\delta_2^*(s_2, b) = q_2$ *then* $\delta^*(s, pair(a, b)) = (q_1, q_2)$.

2. *If* $\delta^*(s, a) = (q, x), q \in Q_1$ *then* $\delta_1^*(s_1, first(a)) = q$ *and* $first(\lambda((q, x))) = \lambda_1(q)$. *If* $\delta^*(s, a) = (x, q), q \in Q_2$ *then* $\delta_2^*(s_2, second(a)) = q$ *and* $second(\lambda(x, q)) = \lambda_2(q)$.

3. *If* $\delta_1^*(s_1, a) = q$ *then there is some* b *such that* $first(b) = a$ *and* $\delta^*(s, b) = (q, x)$ *and* $\lambda_1(q) = first(\lambda((q, x)))$. *If* $\delta_2^*(s_2, a) = q$ *then there is some* b *such that* $second(b) = a$ *and* $\delta^*(s, b) = (x, q)$ *and* $\lambda_2(q) = second(\lambda((q, x)))$.

4.2 Set of Description Types

Using regular trees, we now define the set of types of our type system:

Definition 15 (Set of Description Types) *The set of description type constructors is the set* $\boldsymbol{F}_\tau = \{Record, Variant, Set\} \cup \mathcal{B}$. *A description type is a tree* $\sigma \in \boldsymbol{R}(\boldsymbol{F}_\tau)$ *satisfying the following conditions:*

1. *if* $\sigma(a) = Set$ *then* $\{l \in \mathcal{L} | a \cdot l \in dom(\sigma)\} = \{elm_1\}$,

2. *if* $\sigma(a) = b \in \mathcal{B}$, *then the set* $\{l \in \mathcal{L} | a \cdot l \in dom(\sigma)\}$ *is empty.*

A description type σ *is finite if it is finite as a tree. The set of all description types and the set of all finite description types are denoted by* $\boldsymbol{Dtype}^\infty$ *and* $\boldsymbol{Dtype}$ *respectively.*

Record, *Variant* and *Set* represent the record, the variant and the set type constructors respectively. The condition (1) restricts set types to be " homogeneous" sets. Let $\sigma_1, \ldots, \sigma_n \in \boldsymbol{Dtype}^\infty$. We use the following notations:

$$
\begin{array}{rcl}
[l_1 : \sigma_1, \ldots, l_n : \sigma_n] & \text{for} & Record(l_1 = \sigma_1, \ldots, l_n = \sigma_n), \\
\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle & \text{for} & Variant(l_1 = \sigma_1, \ldots, l_n = \sigma_n), \\
\{\!\{\sigma\}\!\} & \text{for} & Set(elm_1 = \sigma)
\end{array}
$$

$$
\begin{aligned}
\textit{unit} &= [] \\
\textit{point} &= [\textit{X-cord} : \textit{int}, \textit{Y-cord} : \textit{int}] \\
\textit{intlist} &= (\textit{rec } u.\, \langle \textit{Cons} : [\textit{Head} : \textit{int}, \textit{Tail} : u], \textit{Nil} : \textit{unit} \rangle) \\
\textit{object} &= [\textit{Name} : \textit{string}, \textit{Age} : \textit{int}] \\
\textit{person} &= (\textit{rec } p.\, [\textit{Name} : \textit{string}, \textit{Age} : \textit{int}, \textit{Parents} : \{\!\{p\}\!\}]) \\
\textit{employee} &= (\textit{rec } e.\, [\textit{Name} : \textit{string}, \textit{Age} : \textit{int}, \textit{Parents} : \{\!\{\textit{person}\}\!\}, \textit{Salary} : \textit{int}, \textit{Boss} : e]) \\
\textit{student} &= [\textit{Name} : \textit{string}, \textit{Age} : \textit{int}, \textit{Parents} : \{\!\{\textit{person}\}\!\}, \textit{Course} : \{\!\{\textit{string}\}\!\}] \\
\textit{working-student} &= [\textit{Name} : \textit{string}, \textit{Age} : \textit{int}, \textit{Parents} : \{\!\{\textit{person}\}\!\}, \textit{Course} : \{\!\{\textit{string}\}\!\}, \textit{Salary} : \textit{int}, \\
&\qquad \textit{Boss} : \textit{employee}] \\
\textit{flights} &= \{\!\{[\textit{Flight} : [\textit{F-id} : \textit{int}, \textit{Date} : \textit{string}], \textit{Plane} : \textit{string}]\}\!\} \\
\textit{flown-by} &= \{\!\{[\textit{Plane} : \textit{string}, \textit{Pilots} : \{\!\{[\textit{Name} : \textit{string}, \textit{Emp-id} : \textit{int}]\}\!\}]\}\!\} \\
\textit{schedule-data} &= \{\!\{[\textit{Flight} : [\textit{F-id} : \textit{int}, \textit{Date} : \textit{string}], \textit{Plane} : \textit{string}, \\
&\qquad \textit{Pilots} : \{\!\{[\textit{Name} : \textit{string}, \textit{Emp-id} : \textit{int}]\}\!\}]\}\!\}
\end{aligned}
$$

Figure 2: Examples of Description Types

Similar shorthands are adopted in term representations of regular trees.

The set of finite description types $\boldsymbol{Dtype}$ coincides with the following inductively defined set $\boldsymbol{Dtype}^o$:

1. $b \in \boldsymbol{Dtype}^o$ for any $b \in \mathcal{B}$,
2. if $\sigma_1, \ldots, \sigma_n \in \boldsymbol{Dtype}^o$ and $l_1, \ldots, l_n \in \mathcal{L}$ then $[l_1 : \sigma_1, \ldots, l_n : \sigma_n] \in \boldsymbol{Dtype}^o$,
3. if $\sigma_1, \ldots, \sigma_n \in \boldsymbol{Dtype}^o$ and $l_1, \ldots, l_n \in \mathcal{L}$ then $\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle \in \boldsymbol{Dtype}^o$,
4. if $\sigma \in \boldsymbol{Dtype}^o$, then $\{\!\{\sigma\}\!\} \in \boldsymbol{Dtype}^o$.

Figure 2 shows examples of description types in term representation. In this example, as well as in all other examples we will show later, identifiers such as *unit* are used purely as syntactic shorthands to avoid repetitions and have no significance themselves. As seen in these examples, infinite trees correspond to recursively defined types.

For the set $\boldsymbol{Dtype}^\infty$, we define the following ordering to capture the ordering of the containment of the structures:

Definition 16 (Information Ordering on $\boldsymbol{Dtype}^\infty$) *Let $\sigma_1, \sigma_2 \in \boldsymbol{Dtype}^\infty$. The information ordering $\leq$ on $\boldsymbol{Dtype}^\infty$ is the relation defined as: $\sigma_1 \leq \sigma_2$ iff $dom(\sigma_1) \subseteq dom(\sigma_2)$ and for any $a \in dom(\sigma_1)$, $\sigma_1(a) = \sigma_2(a)$ and if $\sigma_1(a) =$ Varinat then $\{l \in \mathcal{L} | a \cdot l \in dom(\sigma_1)\} = \{l \in \mathcal{L} | a \cdot l \in dom(\sigma_2)\}$.*

This ordering can be regarded as a special case of the subsumption ordering on Aït-Kaci's ψ-terms [3]. The condition on variant nodes means that in order for two variant types to be ordered, they must have the same

$$
\begin{aligned}
\textit{unit} &\leq \textit{point} \\
\textit{unit} &\leq \textit{object} \\
\textit{object} &\leq \textit{person} \\
\textit{person} &\leq \textit{employee} \\
\textit{person} &\leq \textit{student} \\
\textit{employee} &\leq \textit{working-student} \\
\textit{student} &\leq \textit{working-student} \\
\textit{flights} &\leq \textit{schedule-data} \\
\textit{flown-by} &\leq \textit{schedule-data}
\end{aligned}
$$

Figure 3: Examples of Ordering on Description Types

set of variants. The intuition behind this condition is that if a variant type σ has a a component $l : \sigma''$ and σ' has no l-component, then for a value v of the type σ corresponding to the component $l : \sigma''$ there is no value v' of the type σ' that is related in structure to v and therefore σ and σ' are not related.

The ordering $\leq$, when restricted to the set of finite description types $\boldsymbol{Dtype}$, coincides with the following inductively defined relation $\leq^o$:

$$
\begin{aligned}
b &\leq^o b \text{ for all } b \in \mathcal{B} \\
[l_1 : \sigma_1, \ldots, l_n : \sigma_n] &\leq^o [l_1 : \sigma'_1, \ldots, l_n : \sigma'_n, \ldots] \text{ if } \sigma_i \leq^o \sigma'_i, 1 \leq i \leq n \\
\{\!\{\sigma\}\!\} &\leq^o \{\!\{\sigma'\}\!\} \text{ if } \sigma \leq^o \sigma' \\
\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle &\leq^o \langle l_1 : \sigma'_1, \ldots, l_n : \sigma'_n \rangle \text{ if } \sigma_i \leq^o \sigma'_i, 1 \leq i \leq n
\end{aligned}
$$

Figure 3 shows examples of the information ordering on $\boldsymbol{Dtype}^\infty$ among the description types defined in figure 2.

From the inductive characterization of $\leq$, it is easy to check that $(\boldsymbol{Dtype}, \leq)$ is a poset with pairwise bounded join property, $\leq$ is decidable and least upper bounds (if they exist) are effectively computable. The following two propositions show that these properties still hold for general description types, whose proofs can be reconstructed from the proofs of the corresponding properties on ψ-terms [3] by checking the extra condition we imposed on the variant nodes.

Proposition 5 $(\boldsymbol{Dtype}^\infty, \leq)$ *is a poset with the pairwise bounded join property.*

Proposition 6 *The ordering* $\leq$ *on* $\boldsymbol{Dtype}^\infty$ *is decidable and for any description types* σ_1, σ_2*, it is decidable whether* σ_1, σ_2 *are consistent or not and if consistent then their least upper bound is effectively computable.*

Combining proposition 5 and 6, we have:

Theorem 5 $(\boldsymbol{Dtype}^\infty, \leq)$ *is a database type system.*

The following is an example of a least upper bound of description types defined in figure 2:

$$\begin{aligned} \textit{employee} \sqcup \textit{student} &= \textit{working-student} \\ \textit{flights} \sqcup \textit{flown-by} &= \textit{schedule-data} \end{aligned}$$

From examples shown in figure 3 and the above examples, we can see that $\leq$ is a generalization of the information ordering on types in the relational models to complex structures including recursive structures represented by infinite trees.

4.3 Universe of Descriptions

In order to construct a model of $(\boldsymbol{Dtype}^{\infty}, \leq)$, we first define a set of possible descriptions.

Definition 17 (Universe of Descriptions) *The set of description constructors is the set* $\boldsymbol{F}_d = \{Record, Inj, Set\} \cup (\bigcup_{b \in \mathcal{B}} B_b) \cup \{null_b | b \in \mathcal{B}\}$. *A description is a tree* $d \in \boldsymbol{R}(\boldsymbol{F}_d)$ *satisfying the following conditions: for all* $a \in dom(d)$,

1. *if* $d(a) = Set$ *then* $\{l \in \mathcal{L} | a \cdot l \in dom(d)\} = \{elm_1, \ldots, elm_n\}$ *for some* $n \geq 0$,
2. *if* $d(a) = Inj$ *then the set* $\{l \in \mathcal{L} | a \cdot l \in dom(d)\}$ *is either a singleton set or the empty set,*
3. *if* $d(a) \in B_b$ *or* $d(a) = null_b$, *then the set* $\{l \in \mathcal{L} | a \cdot l \in dom(d)\}$ *is the empty set.*

A description d is finite if it is finite as a tree. The set of all descriptions and the set of all finite descriptions are denoted by $\boldsymbol{Dobj}^{\infty}$ *and* $\boldsymbol{Dobj}$ *respectively.*

Inj is a variant constructor (injection to a variant type). *Inj* node with no outgoing edge represents a null value of a variant type.

Let $d_1, \ldots, d_n \in \boldsymbol{Dobj}^{\infty}$. We use the following notations:

$$\begin{aligned} [l_1 = d_1, \ldots, l_n = d_n] \quad &\text{for} \quad Record(l_1 = d_1, \ldots, l_n = d_n), \\ \{\!\{d_1, \ldots, d_n\}\!\} \quad &\text{for} \quad Set(elm_1 = d_1, \ldots, elm_n = d_n). \end{aligned}$$

The set of finite descriptions $\boldsymbol{Dobj}$ coincides with the following inductively defined set $\boldsymbol{Dobj}^{o}$:

1. $c \in \boldsymbol{Dobj}^{o}$, for any $c \in B_b, b \in \mathcal{B}$,
2. $null_b \in \boldsymbol{Dobj}^{o}$ for any $b \in \mathcal{B}$,
3. if $d_1, \ldots, d_n \in \boldsymbol{Dobj}^{o}$ and $l_1, \ldots, l_n \in \mathcal{L}$ then $[l_1 = d_1, \ldots, l_n = d_n] \in \boldsymbol{Dobj}^{o}$,
4. $Inj \in \boldsymbol{Dobj}^{o}$,
5. if $d \in \boldsymbol{Dobj}^{o}$ and $l \in \mathcal{L}$ then $Inj(l = d) \in \boldsymbol{Dobj}^{o}$,
6. if $d_1, \ldots, d_n \in \boldsymbol{Dobj}^{o}$ then $\{\!\{d_1, \ldots, d_n\}\!\} \in \boldsymbol{Dobj}^{o}$, $0 \leq n$.

Figure 4 shows examples of descriptions.

$$
\begin{array}{ll}
\textit{Unity} & = [\,] \\
\textit{Point23} & = [\textit{X-cord} = 2,\ \textit{Y-cord} = 3] \\
\textit{Onelist} & = \textit{Inj}(\textit{Cons} = [\textit{Head} = 1,\ \textit{Tail} = \textit{Inj}(\textit{Nil} = \textit{Unity})]) \\
\textit{Null-person} & = (\textit{rec}\ p.\ [\textit{Name} = \textit{null}_{\textit{string}},\ \textit{Age} = \textit{null}_{\textit{int}},\ \textit{Parents} = \{p\}]) \\
\textit{Null-employee} & = (\textit{rec}\ e.\ [\textit{Name} = \textit{null}_{\textit{string}},\ \textit{Age} = \textit{null}_{\textit{int}},\ \textit{Parents} = \{\!\{\textit{Null-person}\}\!\}, \\
 & \qquad \textit{Salary} = \textit{null}_{\textit{int}},\ \textit{Boss} = e]) \\
\textit{John} & = [\textit{Name} = \text{"}\textit{John Smith}\text{"},\ \textit{Age} = 34,\ \textit{Parent} = \{\!\{\textit{Null-person}\}\!\}, \\
 & \quad \textit{Salary} = 23000,\ \textit{Boss} = \textit{Null-employee}] \\
\textit{Mary1} & = [\textit{Name} = \text{"}\textit{Mary Blake}\text{"},\ \textit{Age} = 21,\ \textit{Parent} = \{\!\{\textit{Null-person}\}\!\}, \\
 & \quad \textit{Courses} = \{\!\{\text{"}\textit{math120}\text{"}, \text{"}\textit{phil340}\text{"}, \text{"}\textit{logic110}\text{"}\}\!\}] \\
\textit{Mary2} & = [\textit{Name} = \text{"}\textit{Mary Blake}\text{"},\ \textit{Age} = 21,\ \textit{Parent} = \{\!\{\textit{Null-person}\}\!\}, \\
 & \quad \textit{Salary} = 9000,\ \textit{Boss} = \textit{John}] \\
\textit{Mary3} & = [\textit{Name} = \text{"}\textit{Mary Blake}\text{"},\ \textit{Age} = 21,\ \textit{Parent} = \{\!\{\textit{Null-person}\}\!\}, \\
 & \quad \textit{Courses} = \{\!\{\text{"}\textit{math120}\text{"}, \text{"}\textit{phil340}\text{"}, \text{"}\textit{logic110}\text{"}\}\!\}, \\
 & \quad \textit{Salary} = 9000,\ \textit{Boss} = \textit{John}]
\end{array}
$$

$$
\begin{array}{ll}
\textit{Flights} & = \{\!\{\ [\textit{Flight} = [\textit{F-id} = 001,\ \textit{Date} = \text{"}\textit{8 Aug}\text{"}],\quad \textit{Plane} = \text{"}\textit{Concord}\text{"}], \\
 & \quad\ [\textit{Flight} = [\textit{F-id} = 83,\ \textit{Date} = \text{"}\textit{9 Aug}\text{"}],\quad \textit{Plane} = \text{"}\textit{707}\text{"}], \\
 & \quad\ [\textit{Flight} = [\textit{F-id} = 116,\ \textit{Date} = \text{"}\textit{10 Aug}\text{"}],\ \textit{Plane} = \text{"}\textit{747}\text{"}]\}\!\}
\end{array}
$$

$$
\begin{array}{ll}
\textit{Flown-by} & = \{\!\{\ [\textit{Plane} = \text{"}\textit{Concord}\text{"},\ \textit{Pilots} = \{\!\{\ [\textit{Name}=\text{"}\textit{Jones}\text{"},\ \textit{Emp-id} = 5566]\}\!\}], \\
 & \quad\ [\textit{Plane} = \text{"}\textit{707}\text{"},\quad \textit{Pilots} = \{\!\{\ [\textit{Name} = \text{"}\textit{Clar}\text{"},\ \textit{Emp-id} = 1122], \\
 & \qquad\qquad\qquad\qquad\ [\textit{Name} = \text{"}\textit{Copely}\text{"},\ \textit{Emp-id} = 2233], \\
 & \qquad\qquad\qquad\qquad\ [\textit{Name} = \text{"}\textit{Chin}\text{"},\ \textit{Emp-id} = 3344]\}\!\}], \\
 & \quad\ [\textit{Plane} = \text{"}\textit{747}\text{"},\quad \textit{Pilots} = \{\!\{\ [\textit{Name} = \text{"}\textit{Clark}\text{"},\ \textit{Emp-id} = 1122], \\
 & \qquad\qquad\qquad\qquad\ [\textit{Name} = \text{"}\textit{Jones}\text{"},\ \textit{Emp-id} = 5566]\}\!\}]\}\!\}
\end{array}
$$

$$
\begin{array}{ll}
\textit{Schedule-data} & = \{\!\{\ [\ \textit{Plane} = \text{"}\textit{Concord}\text{"},\ \textit{Pilots} = \{\!\{[\textit{Name} = \text{"}\textit{Jones}\text{"},\ \textit{Emp-id} = 5566]\}\!\}, \\
 & \qquad \textit{Flight} = [\textit{F-id} = 001,\ \textit{Date} = \text{"}\textit{8 Aug}\text{"}]], \\
 & \quad\ [\ \textit{Plane} = \text{"}\textit{707}\text{"},\quad \textit{Pilots} = \{\!\{[\textit{Name} = \text{"}\textit{Clark}\text{"},\ \textit{Emp-id} = 1122], \\
 & \qquad\qquad\qquad\qquad\ [\textit{Name} = \text{"}\textit{Copely}\text{"},\ \textit{Emp-id} = 2233], \\
 & \qquad\qquad\qquad\qquad\ [\textit{Name} = \text{"}\textit{Chin}\text{"},\ \textit{Emp-id} = 3344]\}\!\}, \\
 & \qquad \textit{Flight} = [\textit{F-id} = 83,\ \textit{Date} = \text{"}\textit{9 Aug}\text{"}]], \\
 & \quad\ [\ \textit{Plane} = \text{"}\textit{747}\text{"},\quad \textit{Pilots} = \{\!\{[\textit{Name} = \text{"}\textit{Clark}\text{"},\ \textit{Emp-id} = 1122], \\
 & \qquad\qquad\qquad\qquad\ [\textit{Name} = \text{"}\textit{Jones}\text{"},\ \textit{Emp-id} = 5566]\}\!\}, \\
 & \qquad \textit{Flight} = [\textit{F-id} = 116,\ \textit{Date} = \text{"}\textit{10 Aug}\text{"}]]\}\!\}
\end{array}
$$

Figure 4: Examples of Descriptions

4.4 Typing Relation

Description types represent structures of descriptions. A description d *has* a description type σ if d has the structure represented by σ. This relationship is formalized by the *typing relation*:

Definition 18 (Typing Relation) *Let $\approx$ be the equivalence relation on $\mathcal{L}$ defined as $l_1 \approx l_2$ iff $l_1 = l_2$ or $l_1 = elm_i, l_2 = elm_j$ for some i, j. Define the consistency relation $:^b$ between $\boldsymbol{F}_d$ and $\boldsymbol{F}_\tau$ as follows: $f :^b g$ iff one of the following holds:*

1. $f = g$,
2. $f = Inj$ *and* $g = Variant$,
3. $f \in B_g$ *and* $g \in \mathcal{B}$,
4. $f = null_g$ *and* $g \in \mathcal{B}$.

The typing relation $d : \sigma$ between $\boldsymbol{Dobj}^\infty$ and $\boldsymbol{Dtype}^\infty$ is defined as: $d : \sigma$ iff for all $a \in dom(d)$,

1. *there is some a' such that $a \hat{\approx} a'$, $d(a) :^b \sigma(a')$,*
2. *if $d(a) = Record$ then $\{l \in \mathcal{L} | a \cdot l \in dom(d)\} = \{l \in \mathcal{L} | a' \cdot l \in dom(\sigma)\}$,*
3. *if $d(a) = Inj$ then if $a \cdot l \in dom(d)$ for some $l \in \mathcal{L}$ then $l \in \{l \in \mathcal{L} | a' \cdot l \in dom(\sigma)\}$,*

The equivalence relation $\approx$ "ignores" the difference due to the positions $elm_1, \ldots, elm_n$ of occurrences of subtrees in the set constructor $Set(elm_1 = d_1, \ldots, elm_n = d_n)$.

When restricted to the set of finite descriptions $\boldsymbol{Dobj}$, the above typing relation coincides with the following relation $:^o$ on $\boldsymbol{Dobj} \times \boldsymbol{Dtype}^\infty$ defined by induction on $\boldsymbol{Dobj}$:

1. $c :^o b$ for all $c \in B_b$,
2. $null_b :^o b$,
3. if $d_1 :^o \sigma_1, \ldots, d_n :^o \sigma_n$ and $l_1, \ldots, l_n \in \mathcal{L}$ then $[l_1 = d_1, \ldots, l_n = d_n] :^o [l_1 : \sigma_1, \ldots, l_n : \sigma_n]$,
4. $Inj :^o \langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle$,
5. if $d :^o \sigma$ then $Inj(l = d) :^o \langle \ldots, l : \sigma, \ldots \rangle$,
6. if $d_1 :^o \sigma, \ldots, d_n :^o \sigma$ then $\{\!\{d_1, \ldots, d_n\}\!\} :^o \{\!\{\sigma\}\!\}$.

Note however that $d \in \boldsymbol{Dobj}$ and $d :^o \sigma$ does not implies that $\sigma \in \boldsymbol{Dtype}$ because of variant types, i.e. the rule 4 in the above definition.

Figure 5 shows examples of typing relations that hold between descriptions defined in figure 4 and description types defined in figure 2.

Unity	:	*unit*
Point23	:	*point*
Onelist	:	*intlist*
Null-person	:	*person*
Null-employee	:	*employee*
John	:	*employee*
Mary1	:	*student*
Mary2	:	*employee*
Mary3	:	*working-student*
Flights	:	*flights*
Flown-by	:	*flown-by*
Schedule-data	:	*schedule-data*

Figure 5: Examples of Typing Relation

From the above inductive characterization of typing relation, it is easy to check that for any finite description d and any description type σ it is decidable whether $d : \sigma$ or not. This property is essential to develop a type inference system. Fortunately, this property still holds for general descriptions:

Proposition 7 *For any $d \in \boldsymbol{Dobj}^{\infty}, \sigma \in \boldsymbol{Dtype}^{\infty}$, the property $d : \sigma$ is decidable.*

Proof Let $M_d = (Q_d, s_d, \boldsymbol{F}_d, \delta_d, \lambda_d)$ and $M_\sigma = (Q_\sigma, s_\sigma, \boldsymbol{F}_\tau, \delta_\sigma, \lambda_\sigma)$ be Moore machines representing d and σ respectively. Let $M = (Q, s, \boldsymbol{F}, \delta, \lambda)$ be the product machine $(M_d \times M_\sigma)/\approx$ where $\approx$ is the equivalence relation on $\mathcal{L}$ defined in definition 18. We show that $d : \sigma$ iff M satisfies the following conditions: for any reachable state q,

1. if $q = (q_1, x), q_1 \in Q_1$ then $x \in Q_2$ and $\lambda(q) = (f, g)$ such that $f :^b g$,
2. if $q = (q_1, q_2), q_1 \in Q_1, q_2 \in Q_2$, $\lambda(q) = (\mathit{Record}, \mathit{Record})$ and $\delta(q, l) = q'$ then $l = (l', l'), l' \neq \$$,
3. if $q = (q_1, q_2), q_1 \in Q_1, q_2 \in Q_2$, $\lambda(q) = (\mathit{Inj}, \mathit{Variant})$ and $\delta(q, l)$ is defined then $l = (l', l')$ or $l = (\$, l')$ for some $l' \neq \$$.

By lemma 3, M satisfies the condition 1 iff for any $a \in dom(M_1(s_1))$, there is some a' such that $a \hat{\approx} a'$, $\delta_1^*(s_1, a) = q_1$, $\delta_2^*(s_2, a') = q_2$, and $\lambda_1(q_1) :^b \lambda_2(q_2)$. Since M_d, M_σ represent d, σ respectively, this condition is equivalent to the condition 1 of the definition of the typing relation. The equivalences of the conditions 2, 3 of the propositions and the conditions 2, 3 of the definition of the typing relation are immediate consequences of their definitions.

Since M is effectively constructed and the above property is clearly decidable, the proposition is proved. ∎

4.5 Description Domains

By the typing relation, we can identify for each description type the corresponding set of descriptions. By defining a proper ordering, we turn this set into a description domain. For a pair of trees d_1, d_2, Courcelle described [19] the notion of a *coherent* and *simplifiable* relation on $\boldsymbol{Subtrees}(d_1) \times \boldsymbol{Subtrees}(d_2)$ as a relation $\simeq$ satisfying the condition that if

$$f(l_1 = d_1, \ldots, l_n = d_n) \simeq g(l_1 = d'_1, \ldots, l_n = d'_n)$$

then $d_i \simeq d'_i$, $0 \leq i \leq n$ and $f = g$. By generalizing this and combining it with Smyth powerdomain preorder, we can generalize the information ordering on flat descriptions to $\boldsymbol{Dobj}^\infty$:

Definition 19 (Information Preorder on $\boldsymbol{Dobj}^\infty$) *The information ordering on the set $\boldsymbol{F}_d$ of description constructors is the following partial ordering $\sqsubseteq^b$:*

$$f \sqsubseteq^b g \text{ iff } f = g \text{ or } f = null_b \text{ and } g \in B_b$$

The information preorder $\preceq$ on $\boldsymbol{Dobj}^\infty$ is the relation defined as: $d_1 \preceq d_2$ iff there is a relation $\simeq$, called substructure relation, on $\boldsymbol{Subtrees}(d_1) \times \boldsymbol{Subtrees}(d_2)$ satisfying the following properties:

1. $d_1 \simeq d_2$,
2. *if* $d \simeq d'$ *then* $d(\epsilon) \sqsubseteq^b d'(\epsilon)$,
3. *if* $d \simeq d'$ *and* $d(\epsilon) = Record$ *then* $\{l \in \mathcal{L} | l \in dom(d)\} = \{l \in \mathcal{L} | \mathcal{L}l \in dom(d')\}$ *and for all* $l \in \{l \in \mathcal{L} | l \in dom(d)\}$ $d/l \simeq d'/l$,
4. *if* $d \simeq d'$, $d(\epsilon) = Variant$ *and* $l \in dom(d)$ *then* $l \in dom(d')$ *and* $d/l \simeq d'/l$,
5. *if* $d \simeq d'$, $d(\epsilon) = Set$ *then for all* $l \in \{l \in \mathcal{L} | l \in dom(d')\}$ *there is some* $l' \in \{l \in \mathcal{L} | l \in dom(d)\}$ *such that* $d/l' \simeq d'/l$.

This ordering is also closely related to *Smyth simulation* on a certain class of directed graphs defined in [49].

The relation $\preceq$, when restricted to the set of finite descriptions $\boldsymbol{Dobj}$, coincides with the following inductively defined relation $\preceq^o$:

$$\begin{aligned}
c &\preceq^o c \text{ for all } c \in B_b, \\
null_b &\preceq^o c \text{ for all } c \in B_b, \\
null_b &\preceq^o null_b, \\
[l_1 = d_1, \ldots, l_n = d_n] &\preceq^o [l_1 = d'_1, \ldots, l_n = d'_n] \text{ if } d_i \preceq^o d'_i, 1 \leq i \leq n, \\
Inj &\preceq^o Inj(l = d) \text{ for all } d, \\
Inj(l = d) &\preceq^o Inj(l = d') \text{ if } d \preceq^o d', \\
\{\!\{d_1, \ldots, d_n\}\!\} &\preceq^o \{\!\{d'_1, \ldots, d'_m\}\!\} \text{ if } \forall d' \in \{d'_1, \ldots, d'_m\}. \exists d \in \{d_1, \ldots, d_n\}. d \preceq^o d'
\end{aligned}$$

On a substructure relation $\simeq$, the following property hold:

Lemma 4 *Let $d_1 \preceq d_2$ and $\simeq$ be a substructure relation on $\boldsymbol{Subtrees}(d_1) \times \boldsymbol{Subtrees}(d_2)$. For $d_1' \in \boldsymbol{Subtrees}(d_1), d_2' \in \boldsymbol{Subtrees}(d_2)$, if $d_1' \simeq d_2'$ then $d_1' \preceq d_2'$.*

Proof Immediate consequence of the fact that the restriction of a substructure relation $\simeq$ to $\boldsymbol{Subtrees}(d_1') \times \boldsymbol{Subtrees}(d_2')$ is also a substructure relation. ∎

We next show that $\preceq$ is a preorder having the desired properties. Rounds' recent work [49] also shows a similar results in a slightly different framework.

Proposition 8 *The relation $\preceq$ is a preorder on $\boldsymbol{Dobj}^\infty$ with the pairwise bounded join property.*

The strategy of the following rather long proof is the combination of the technique suggested in [3] to construct a least upper bound of two regular trees by tracing the moves of two Moore machines representing them in parallel and the property of Smyth powerdomain preorder shown in [56] that if s_1 and s_2 are finite subset of a a poset then $\{\!\{d_1 \sqcup d_2 | d_1 \in s_1, d_2 \in s_2 \text{ and } d_1 \sqcup d_2 \text{ exists}\}\!\}$ is a least upper bound of s_1 and s_2 under the Smyth preorder.

Proof For any description d, the identity relation on $\boldsymbol{Subtrees}(d)$ is a substructure relation and $d \preceq d$. Suppose $d_1 \preceq d_2$ and $d_2 \preceq d_3$. Let $\simeq_1$ and $\simeq_2$ be substructure relations on $\boldsymbol{Subtrees}(d_1) \times \boldsymbol{Subtrees}(d_2)$ and $\boldsymbol{Subtrees}(d_2) \times \boldsymbol{Subtrees}(d_3)$ respectively. Then the composition of the two relations r_1, r_2 also satisfies the conditions of substructure relation. Therefore $d_1 \preceq d_3$ and $\preceq$ is a preorder.

We next show that $\preceq$ has the pairwise bounded join property by showing the following stronger property: There is an algorithm taking any two descriptions d_1, d_2 that determines whether d_1, d_2 have an upper bound or not and that if d_1, d_2 have an upper bound then computes (one of) their least upper bound. Let $M_{d_1} = (Q_1, s_1, \boldsymbol{F}_d, \delta_1, \lambda_1)$ and $M_{d_2} = (Q_2, s_2, \boldsymbol{F}_d, \delta_2, \lambda_2)$ be Moore machine representing d_1, d_2 respectively. Let M be the product machine $(M_1 \times M_2)/\approx$. We say that a state q in M is *consistent* iff it satisfies the condition that if $q = (q_1, q_2)$ for some $q_1 \in Q_1, q_2 \in Q_2$ then $\lambda(q) = (f, g)$ for some $f, g \in \boldsymbol{F}_d$ such that f, g has an upper bound and the following conditions are satisfied:

1. if $\lambda(q) =$ (*Record, Record*) then for all l if $\delta(q, l)$ is defined and equal q' then $l = (l', l')$ for some l' and q' is consistent,

2. if $\lambda(q) = (Inj, Inj)$ then there is at most one l such that $\delta(q, l) = q'$ and if $\delta(q, (l', l')) = q'$ for some l' then q' is consistent.

We first show that if d_1, d_2 has an upper bound then s is consistent. Suppose s is not consistent. Then there is some $a \in \mathcal{L}^*$ such that (1) $\delta^*(s, a) = (q_1, q_2), q_1 \in Q_1, q_2 \in Q_2$ and (2) for any prefix b of a $\lambda(s, b)$ is either (*Record, Record*) or (Inj, Inj) and (3) one of the following hold: (a) $\lambda((q_1, q_2)) = (f, g)$ such that $\{f, g\}$ has no upper bound, (b) $\lambda((q_1, q_2)) =$ (*Record, Record*) and there is some $(l_1, l_2), l_1 \neq l_2$ such that $\delta((q_1, q_2), (l_1, l_2))$ is defined, (c) $\lambda((q_1, q_2)) = (Inj, Inj)$ and there are at least two distinct l_1, l_2 such that both $\delta((q_1, q_2), l_1)$ and $\delta((q_1, q_2), l_2)$ are defined. Now suppose to the contrary that there is some d such that $d_1 \preceq d$ and $d_2 \preceq d$. Let a be a string satisfying the condition (1) and (2). Then by lemma 4, $d_1/a \preceq d/a$ and $d_2/a \preceq d/a$, which contradicts the condition (3).

Next we show that if s is consistent then d_1, d_2 has a least upper bound by constructing one. Suppose s is consistent. Define $M' = (Q, s, \boldsymbol{F}_d, \delta', \lambda')$ from M as follows:

1. Q, s are same as M,
2. $\delta'(q, l)$ is defined and equal to q' iff one of the following hold:
   (a) $\lambda(q) = (\mathit{Record}, \mathit{Record})$ and $\delta(q, (l, l)) = q'$,
   (b) $\lambda(q) = (\mathit{Set}, \mathit{Set})$, $l = \mathit{elm}_i$ and $\delta(q, (\mathit{elm}_j, \mathit{elm}_k)) = q'$ where $(\mathit{elm}_j, \mathit{elm}_k)$ is the i^{th} smallest symbol under the total order $\ll$ on $\mathcal{L}$ in the set $\{(\mathit{elm}_l, \mathit{elm}_n) | \delta(q, (\mathit{elm}_l, \mathit{elm}_n))$ is defined and consistent$\}$,
   (c) $\lambda(q) = (\mathit{Inj}, \mathit{Inj})$ and one of the following hold: (i) $\delta(q, (l, l)) = q'$, (ii) $\delta(q, (l, \$)) = q'$ or (iii) $\delta(q, (\$, l)) = q'$,
   (d) $q = (q_1, \$)$ and $l = (l, \$)$ or $q = (\$, q_2)$ and $l = (\$, l)$.
3. λ' is defined as

$$\lambda'(q) = \begin{cases} x \sqcup y & \text{if } \lambda(q) = (x, y), x, y \in \boldsymbol{F}_d \text{ and } x \sqcup y \text{ exists} \\ x & \text{if } \lambda(q) = (x, \$) \\ y & \text{if } \lambda(q) = (\$, y) \\ \$ & \text{otherwise} \end{cases}$$

We show that $M'(s)$ is a least upper bound of d_1, d_2. Let $S_1 = \{M_1(q) | q \in Q_1, q$ reachable$\}$, $S_2 = \{M_2(q) | q \in Q_2, q$ reachable$\}$, and $S = \{M'(q) | q \in Q, q$ reachable$\}$. Then $S_1 = \boldsymbol{Subtrees}(d_1), S_2 = \boldsymbol{Subtrees}(d_2)$ and $S = \boldsymbol{Subtrees}(M'(s))$. Define the relation $\simeq_1$ between S_1 and S as $M_1(q) \simeq_1 M'(q')$ iff $q' = (q, x)$ for some x. Then it is easily checked that this relation satisfies the conditions of substructure relation and therefore $d_1 \preceq M'(s)$. Similarly $d_2 \preceq M'(s)$. Let d be any upper bound of d_1, d_2. Let $\simeq'_1, \simeq'_2$ be substructure relations on $\boldsymbol{Subtrees}(d_1) \times \boldsymbol{Subtrees}(d)$ and $\boldsymbol{Subtrees}(d_2) \times \boldsymbol{Subtrees}(d)$ respectively. Define the relation $\simeq$ on $S \times \boldsymbol{Subtrees}(d)$ as $M'(q) \simeq d'$ iff one of the following hold: (1) $q = (q_1, \$), M_1(q_1) \simeq'_1 d'$, (2) $q = (\$, q_2), M_2(q_2) \simeq'_2 d'$, or (3) $q = (q_1, q_2), M_1(q_1) \simeq'_1 d', M_2(q_2) \simeq'_2 d'$. Then $\simeq$ clearly satisfies the conditions 1,2,3,4 of the definition of substructure relation. For the condition 5 of substructure relation, suppose $M'(q) \simeq d'$ and $M'(q) = \mathit{Set}$. If $q = (q_1, \$)$ or $q = (\$, q_2)$ then the condition 5 follows from the fact that $\simeq'_1, \simeq'_2$ are substructure relations. Suppose $q = (q_1, q_2)$. Then $M_1(q_1) \simeq'_1 d'$ and $M_2(q_2) \simeq'_2 d'$. If $l \in \mathit{dom}(d')$ for some $l \in \mathcal{L}$, then there is some $l_1, l_2 \in \mathcal{L}$ such that $\delta_1(q_1, l_1) = q'_1, \delta_2(q_2, l_2) = q'_2$, $M_1(q'_1) \simeq'_1 d'/l$ and $M_2(q'_2) \simeq'_2 d'/l$. By lemma 4, $M_1(q'_1) \preceq d'/l$ and $M_2(q'_2) \preceq d'/l$. Let M'_1, M'_2, M'' be respectively Moore machines obtained from M_1, M_2, M' by respectively replacing their start states with $q'_1, q'_2, (q'_1, q'_2)$. Clearly $M_1(q'_1) = M'_1(q'_1)$, $M_2(q'_2) = M'_2(q'_2)$ and $M'' = (M'_1 \times M'_2)/\approx$. Since $M'_1(q'_1)$ and $M_2(q'_2)$ has an upper bound, (q'_1, q'_2) is consistent. By definition, $l_1 = \mathit{elm}_i$ and $l_2 = \mathit{elm}_j$ for some i, j. Then by definition of M' there is some l' such that $\delta'(q, l) = (q'_1, q'_2)$ and therefore $M'(q)/l' \simeq d'/l$.

Since M' is effectively constructed, the proposition was proved. ∎

The above proof also establishes that least upper bounds of $\preceq$ are effectively computable. For the Moore machine M' defined in the above proof, the following property can be also easily shown: $d_1 \preceq d_2$ iff M' satisfies the condition that for all reachable state q in M' if q is consistent then it is of the form $q = (x, q_2)$ and if $q = (q_1, q_2), q_1 \in Q_1, q_2 \in Q_2$ then $\lambda_1(q_1) \sqsubseteq^b \lambda_2(q_2)$. Therefore we have:

Proposition 9 *The relation $\preceq$ on $\boldsymbol{Dobj}^\infty$ is decidable and least upper bounds (if they exist) are effectively computable.*

The next proposition show that the typing is preserved by least upper bound.

Proposition 10 *If $d_1 : \sigma$, $d_2 : \sigma$ and d is a least upper bound of d_1, d_2 then $d : \sigma$.*

Proof Let d_1, d_2 be any descriptions and M' be the Moore machine representing a least upper bound of d_1 and d_2 constructed in the proof of proposition 8. By the construction of M', for any $a \in dom(M'(s))$ either there is some $b \in dom(d_1)$ such that $a \approx b$ and $d_1(b) \sqsubseteq M'(s)(a)$ or there is some $c \in dom(d_2)$ such that $a \approx c$ and $d_2(c) \sqsubseteq M'(s)(a)$. Since for some $x, y \in \boldsymbol{F}_d$, if $x \sqsubseteq y$ and $x :^b f$ for some $f \in \boldsymbol{F}_\tau$ then $y :^b f$, in either case a satisfies the conditions of the definition of the typing relation $d : \sigma$. ∎

Definition 20 *For any description type $\sigma \in \boldsymbol{Dtype}^\infty$, the description domain D_σ associated with σ is the poset $[(\{d | d : \sigma\}, \preceq)]$.*

Theorem 6 *For any σ, D_σ is a description domain.*

Proof We show that D_σ has a bottom element. By definition of D_σ, it is suffices to show the existence of a description d such that $d \preceq d'$ for all $d' \in \{d | d : \sigma\}$. Define a mapping $nullval : \boldsymbol{F}_\tau \to \boldsymbol{F}_d$ as

$$nullval(f) = \begin{cases} null_b & \text{if } f \in \mathcal{B} \\ Inj & \text{if } f = Varinat \\ f & \text{otherwise} \end{cases}$$

For any σ, define the description $Null(\sigma)$ as follows:

1. $a \in dom(Null(\sigma))$ iff $a \in dom(\sigma)$ and there is no proper prefix b of a such that $\sigma(b) = Varinat$, and
2. for all $a \in dom(Null(\sigma))$, $Null(\sigma)(a) = nullval(\sigma(a))$.

From this definition, it is easy to check that $Null(\sigma) : \sigma$ and $Null(\sigma) \preceq d$ for any description $d : \sigma$. Then the theorem follows from propositions 8, 9, 10 and lemma 1. ∎

4.6 A Model of the Type System

We now define the set of embedding-projection pairs to connect the set of description domains and turn them into a database domain.

For defining functions and properties on D_σ, the following definitions and results are useful. Let $(P_1, \leq_1)$, $(P_2, \leq_2)$ be a preordered sets. A function $f : P_1 \to P_2$ is *monotone* iff for any $p_1, p_2 \in P_1$, if $p_1 \leq_1 p_2$ then $f(p_1) \leq_2 f(p_2)$. For a monotone function $f : P_1 \to P_2$, define $[f] : P_1/\equiv \to P_2/\equiv$ as $[f]([x]) = [f(x)]$. Since f is monotone, $[f]$ is well defined. It is also clear that $[f]$ is monotone. The following lemma is an immediate consequence of the definition.

Lemma 5 *Let $(P_1, \leq_1), (P_2, \leq_2)$ be a preordered sets and $f : P_1 \to P_2$, $g : P_2 \to P_1$ be monotone functions. If for all $p \in P_1$, $g(f(p)) = p$ and for all $p \in P_2$, $f(g(p)) \leq_2 p$ then $([f], [g])$ is an embedding-projection pair between $[(P_1, \leq_1)]$ and $[(P_2, \leq_2)]$.*

Definition 21 *Let $\sigma_1, \sigma_2 \in \boldsymbol{Dtype}^\infty$ such that $\sigma_1 \leq \sigma_2$. $\phi_{\sigma_1 \to \sigma_2}$ is a function from D_{σ_1} to D_{σ_2} defined as follows: $a \in dom(\phi_{\sigma_1 \to \sigma_2}(d))$ iff either (1) $a \in dom(d)$ or (2) $a \in dom(\sigma_2)$ satisfying the following conditions:*

1. *if b is the longest prefix of a such that $b \in dom(d)$ then $d(b) = Record$, and*
2. *a has no proper prefix b such that $b \notin dom(d)$ and $\sigma_2(b) = Varinat$,*

and for any $a \in dom(\phi_{\sigma_1 \to \sigma_2}(d))$,

$$\phi_{\sigma_1 \to \sigma_2}(d)(a) = \begin{cases} d(a) & \text{if } a \in dom(d) \\ nullval(\sigma_2(a)) & \text{otherwise} \end{cases}$$

where nullval is a function from $\boldsymbol{F}_\tau$ to $\boldsymbol{F}_d$ defined in the proof of theorem 6.

$\psi_{\sigma_2 \to \sigma_1}$ is a mapping from D_{σ_2} to D_{σ_1} defined as follows: for any $d \in D_{\sigma_2}$, $\psi_{\sigma_2 \to \sigma_1}(d)$ is the restriction of d such that $a \in dom(\psi_{\sigma_2 \to \sigma_1})(d)$ iff $a \in dom(d)$ and there is some $b \in dom(\sigma_2)$ such that $a \approx b$.

Define

$$\begin{aligned} \boldsymbol{Emb}^\infty &= \{\phi_{\sigma_1 \to \sigma_2} | \sigma_1, \sigma_2 \in \boldsymbol{Dtype}^\infty, \sigma_1 \leq \sigma_2\} \\ \boldsymbol{Emb} &= \{\phi_{\sigma_1 \to \sigma_2} | \sigma_1, \sigma_2 \in \boldsymbol{Dtype}, \sigma_1 \leq \sigma_2\} \\ \boldsymbol{Proj}^\infty &= \{\psi_{\sigma_1 \to \sigma_2} | \sigma_1, \sigma_2 \in \boldsymbol{Dtype}^\infty, \sigma_2 \leq \sigma_1\} \\ \boldsymbol{Proj} &= \{\psi_{\sigma_1 \to \sigma_2} | \sigma_1, \sigma_2 \in \boldsymbol{Dtype}, \sigma_2 \leq \sigma_1\} \end{aligned}$$

For $\boldsymbol{Emb}$ and $\boldsymbol{Proj}$, there are inductive definitions. We first define functors (function constructions) for records, variants and sets.

1. *Records.*
   Let $f_1 : \sigma_1^1 \to \sigma_1^2, \ldots, f_n : \sigma_n^1 \to \sigma_n^2$ be any functions and $c_{n+1}, \ldots, c_{n+m}$ be any constants. $[l_1 = f_1, \ldots, l_n = f_n, l_{n+1} = c_{n+1}, \ldots, l_{n+m} = c_{n+m}]$ is the function on records of type $[l_1 : \sigma_1^1, \ldots, l_n : \sigma_n^1]$ defined as

   $$[l_1 = f_1, \ldots, l_n = f_n, l_{n+1} = c_{n+1}, \ldots, l_{n+m} = c_{n+m}]([l_1 = d_1, \ldots, l_n = d_n]) = [l_1 = f_1(d_1), \ldots, l_n = f_n(d_n), l_{n+1} = c_{n+1}, \ldots, l_{n+m} = c_{n+m}]$$

   and $[l_1 = f_1, \ldots, l_k = f_k, l_{k+1} = \$, \ldots, l_n = \$], k \leq n$ is the function on records of type $[l_1 : \sigma_1^1, \ldots, l_k = \sigma_k^1, l_{k+1} = \sigma_{k+1}, \ldots, l_n = \sigma_n]$ for some $\sigma_{k+1}, \ldots, \sigma_n$ defined as

   $$[l_1 = f_1, \ldots, l_k = f_k, l_{k+1} = \$, \ldots, l_n = \$]([l_1 = d_1, \ldots, l_n = d_n]) = [l_1 = f_1(d_1), \ldots, l_k = f_k(d_k)]$$

2. *Variants.*
   Let $f_1 : \sigma_1^1 \to \sigma_1^2, \ldots, f_n : \sigma_n^1 \to \sigma_n^2$ be any functions. $\langle l_1 = f_1, \ldots, l_n = f_n \rangle$ is the function on variants

of type $\langle l_1 : \sigma_1^1, \ldots, l_n : \sigma_n^1 \rangle$ defined as

$$\begin{aligned} \langle l_1 = f_1, \ldots, l_n = f_n \rangle(Inj) &= Inj \\ \langle l_1 = f_1, \ldots, l_n = f_n \rangle(Inj(l_i = d)) &= Inj(l_i = f_i(d)), 1 \leq i \leq n \end{aligned}$$

3. *Sets.*
   Let $f : \sigma_1 \rightarrow \sigma_2$ be any function. $\{\!\{f\}\!\}$ is the function on sets of type $\{\!\{\sigma_1\}\!\}$ defined as

$$\{\!\{f\}\!\}(\{\!\{d_1, \ldots, d_n\}\!\}) = \{\!\{f(d_1), \ldots, f(d_n)\}\!\}$$

Then $\boldsymbol{Emb}$ coincides with the following inductively defined set $\boldsymbol{Emb}^o$:

1. $id_b \in \boldsymbol{Emb}^o$ for any $b \in \mathcal{B}$ where id_b is the identity function on B_b,
2. if $\phi_{\sigma_1^1 \rightarrow \sigma_1^2}, \ldots, \phi_{\sigma_n^1 \rightarrow \sigma_n^2} \in \boldsymbol{Emb}^o$ and $\sigma_{n+1}, \ldots, \sigma_{n+m} \in \boldsymbol{Dtype}$ then $[l_1 = \phi_{\sigma_1^1 \rightarrow \sigma_1^2}, \ldots l_n = \phi_{\sigma_n^1 \rightarrow \sigma_n^2}, l_{n+1} = Null(\sigma_{n+1}), \ldots, l_{n+m} = Null(\sigma_{n+m})] \in \boldsymbol{Emb}^o$ where $Null(\sigma_i)$ is the value defined in theorem 6,
3. if $\phi_{\sigma_1^1 \rightarrow \sigma_1^2}, \ldots, \phi_{\sigma_n^1 \rightarrow \sigma_n^2} \in \boldsymbol{Emb}^o$ then $\langle l_1 = \phi_{\sigma_1^1 \rightarrow \sigma_1^2}, \ldots l_n = \phi_{\sigma_n^1 \rightarrow \sigma_n^2} \rangle \in \boldsymbol{Emb}^o$,
4. if $\phi_{\sigma_1 \rightarrow \sigma_2} \in \boldsymbol{Emb}^o$ then $\{\!\{\phi_{\sigma_1 \rightarrow \sigma_2}\}\!\} \in \boldsymbol{Emb}^o$.

The $\boldsymbol{Proj}$ coincides with the following inductively defined set:

1. $id_b \in \boldsymbol{Proj}$ where id_b is the identity function on B_b,
2. if $\psi_{\sigma_1^1 \rightarrow \sigma_1^2}, \ldots, \psi_{\sigma_n^1 \rightarrow \sigma_n^2} \in \boldsymbol{Proj}$ then $[l_1 = \psi_{\sigma_1^1 \rightarrow \sigma_1^2}, \ldots l_n = \psi_{\sigma_n^1 \rightarrow \sigma_n^2}, l_{n+1} = \$, \ldots, l_{n+m} = \$] \in \boldsymbol{Proj}$,
3. if $\psi_{\sigma_1^1 \rightarrow \sigma_1^2}, \ldots, \psi_{\sigma_n^1 \rightarrow \sigma_n^2} \in \boldsymbol{Proj}$ then $\langle l_1 = \psi_{\sigma_1^1 \rightarrow \sigma_1^2}, \ldots l_n = \psi_{\sigma_n^1 \rightarrow \sigma_n^2} \rangle \in \boldsymbol{Proj}$,
4. if $\psi_{\sigma_1 \rightarrow \sigma_2} \in \boldsymbol{Proj}$ then $\{\!\{\psi_{\sigma_1 \rightarrow \sigma_2}\}\!\} \in \boldsymbol{Proj}$.

Proposition 11 *For any σ_1, σ_2 such that $\sigma_1 \leq \sigma_2$, $([\phi_{\sigma_1 \rightarrow \sigma_2}], [\psi_{\sigma_2 \rightarrow \sigma_1}])$ is an embedding-projection pair between D_{σ_1} and D_{σ_2}.*

Proof For any element $d \in D_{\sigma_1}$, let $d' = \phi_{\sigma_1 \rightarrow \sigma_2}(d)$ and $d'' = \psi_{\sigma_2 \rightarrow \sigma_1}(d')$. By definition of $\phi_{\sigma_1 \rightarrow \sigma_2}$, $dom(d) \subseteq dom(d')$ and for any $a \in dom(d), d'(a) = d(a)$. By definition of $\psi_{\sigma_2 \rightarrow \sigma_1}$, $a \in dom(d'')$ iff $a \in dom(d')$ and there is some b such that $a \approx b$, $b \in dom(\sigma_1)$. Also for any $a \in dom(d''), d''(a) = d'(a)$. But by definition of D_{σ_1}, $a \in dom(d)$ iff there is some b such that $a \approx b$, $b \in dom(\sigma_1)$. Therefore $d = d''$ and hence $\psi_{\sigma_2 \rightarrow \sigma_1}(\phi_{\sigma_1 \rightarrow \sigma_2})(d) = d$.

For any element $d \in D_{\sigma_2}$, let $d' = \psi_{\sigma_2 \rightarrow \sigma_1}(d)$ and $d'' = \phi_{\sigma_1 \rightarrow \sigma_2}(d')$. Define a relation $\simeq$ on $\boldsymbol{Subtrees}(d'') \times \boldsymbol{Subtrees}(d)$ as follows: for $d_1 \in \boldsymbol{Subtrees}(d''), d_2 \in \boldsymbol{Subtrees}(d)$, $d_1 \simeq d_2$ iff either there is some $a \in dom(d')$ such that $d_1 = d''/a$ and $d_2 = d/a$, or there is some a, b such that $a \notin dom(d')$, $a \approx b$, $d_1 = d''/b$ and $d_2 = d/a$. Since $\epsilon \in dom(d')$, $d'' \simeq d$. Suppose $d_1 = d''/a, d_2 = d/a$ for some $a \in dom(d')$. By definition of $\phi_{\sigma_1 \rightarrow \sigma_2}$ and $\psi_{\sigma_2 \rightarrow \sigma_1}$, $d''(a) = d'(a) = d(a)$. Suppose $d_1 = d''/b, d_2 = d/a$ for some $a \notin dom(d')$, $a \approx b$. Then by definition of $\phi_{\sigma_1 \rightarrow \sigma_2}$, $b \in dom(\sigma_2)$ and $d''(b) = nullval(\sigma_2(b))$. By the property of *nullval*,

$d''(b) \sqsubseteq^b d(a)$. Therefore in both case $d_1(\epsilon) \sqsubseteq^b d_2(\epsilon)$. The other conditions of substructure relation (condition 3–5) can be easily checked by distinguishing cases whether $a \in dom(d')$ or not and using the property of the typing relation in the latter case.

For the monotonicity of $\phi_{\sigma_1 \to \sigma_2}$, let $d_1, d_2 \in D_{\sigma_1}$ and $d_1' = \phi_{\sigma_1 \to \sigma_2}(d_1)$, $d_2' = \phi_{\sigma_1 \to \sigma_2}(d_2)$. Suppose there is a substructure relation $\simeq$ on $\boldsymbol{Subtrees}(d_1) \times \boldsymbol{Subtrees}(d_2)$. Define a relation $\simeq'$ on $\boldsymbol{Subtrees}(d_1') \times \boldsymbol{Subtrees}(d_2')$ as follows: $d \simeq' d'$ iff either (1) there are a, b such that $d_1/a \simeq d_2/b$ and $d = d_1'/a, d' = d_2'/a$ or (2) there are a, b, c such that $d_1/a \simeq d_2/b$, $a \cdot c \notin dom(d_1)$, $b \cdot c \notin dom(d_2)$, $d = d_1'/a \cdot c$ and $d' = d_2/b \cdot c$. It can then checked that $\simeq'$ is a substructure relation. For the monotonicity of $\psi_{\sigma_2 \to \sigma_1}$, let $d_1, d_2 \in D_{\sigma_2}$ and $d_1' = \psi_{\sigma_2 \to \sigma_1}(d_1)$, $d_2' = \psi_{\sigma_2 \to \sigma_1}(d_2)$. Suppose there is a substructure relation $\simeq$ on $\boldsymbol{Subtrees}(d_1) \times \boldsymbol{Subtrees}(d_2)$. Define a relation $\simeq'$ on $\boldsymbol{Subtrees}(d_1') \times \boldsymbol{Subtrees}(d_2')$ as: $d \simeq' d'$ iff there are a, b such that $d_1/a \simeq d_2/b$ and $d = d_1'/a, d' = d_2'/a$ or Then it is easily verify that $\simeq'$ is a substructure relation. Then the proposition follows from lemma 5. ∎

From the inductive characterization of $\boldsymbol{Emb}$ and $\boldsymbol{Proj}$ it is easy to see that all embeddings and projections between finite types are computable functions. This necessary property still hold for general embeddings and projections.

Proposition 12 *Elements of* $\boldsymbol{Emb}^\infty$ *and* $\boldsymbol{Proj}^\infty$ *are all computable functions.*

Proof We first show for the embeddings in $\boldsymbol{Emb}^\infty$. Let $\sigma_1 \leq \sigma_2$ and $d : \sigma_1$. Let $M_d = (Q_d, s_d, \boldsymbol{F}_d, \delta_d, \lambda_d)$ and $M_{\sigma_2} = (Q_{\sigma_2}, \boldsymbol{F}_\tau, \delta_{\sigma_2}, \lambda_{\sigma_2})$ be Moore machines representing d, σ_2 respectively. Let $M = (Q, s, \boldsymbol{F}, \delta, \lambda) = (M_d \times M_{\sigma_2})/\approx$ be the product machines modulo the equivalence relation $\approx$ defined in definition 18. Define $M' = (Q, s, \boldsymbol{F}_d, \delta', \lambda')$ from M as follows:

1. Q, s are same as M,
2. $\delta'(q, l)$ is defined and equal to q' iff either $\delta(q, (l, l')) = q$ and $l \neq \$$, or $\delta(q, (\$, l)) = q'$ and $\lambda(q) \neq (x, \textit{Variant})$ for some x.
3. $\lambda'(q) = f$ iff either $\lambda(q) = (f, g)$ for some g or $\lambda(q) = (\$, g)$ and $f = nullval(g)$.

It can then be checked that $M'(s) = \phi_{\sigma_1 \to \sigma_2}(d)$.

For projections in $\boldsymbol{Proj}^\infty$, let $\sigma_2 \leq \sigma_1$ and $d : \sigma_1$. Let $M_d = (Q_d, s_d, \boldsymbol{F}_d, \delta_d, \lambda_d)$ and $M_{\sigma_2} = (Q_{\sigma_2}, \boldsymbol{F}_\tau, \delta_{\sigma_2}, \lambda_{\sigma_2})$ be Moore machines representing d, σ_2 respectively. Let $M = (Q, s, \boldsymbol{F}, \delta, \lambda) = (M_d \times M_{\sigma_2})/\approx$. Define $M' = (Q, s, \boldsymbol{F}_d, \delta', \lambda')$ from M as follows:

1. Q, s are same as M,
2. $\delta'(q, l)$ is defined and equal to q' iff $\delta(q, (l, l')) = q$ and $l' \neq \$$.
3. $\lambda'(q) = f$ iff $\lambda(q) = (f, g)$ for some g.

Then by lemma 3, $M'(s) = \psi_{\sigma_1 \to \sigma_2}(d)$. ∎

We now have the following theorem:

Theorem 7 *$(\{D_\sigma | \sigma \in \boldsymbol{Dtype}^\infty\}, \{[\phi] | \phi \in \boldsymbol{Emb}^\infty\})$ is a database domain and a model of $(\boldsymbol{Dtype}^\infty, \leq)$.*

Proof By proposition 11, for all $\phi \in \boldsymbol{Emb}^\infty$, $[\phi]$ is an embedding. Since $\boldsymbol{Dtype}^\infty$ is a poset with pairwise bounded join property, the conditions 1 – 4 of database domain are satisfied by the set $\{[\phi] | \phi \in \boldsymbol{Emb}^\infty\}$. The condition 5 is shown by proposition 12. ∎

This theorem says that we have successfully completed the constructions of a type system for complex database objects and its semantic domain. The type system allows arbitrarily complex objects constructed by records, variants, finite sets and recursion. A join and a projection are available as computable functions on *arbitrarily* complex structures as given by the equations (1) and (2). Moreover, since these operations have polymorphic type schemes (3) and (4), the result types can be always computed from the types of their arguments without actually computing them. The following are examples of joins of descriptions in figure 4:

$$join(Mary1, Mary2) = Mary3$$

$$join(Flights, Flown\text{-}by) = Schedule\text{-}data$$

The types of the above two joins are *working-student* and *schedule-data* respectively, which are computed from the types of their arguments. This property enables us to develop a static type inference system. The another important implication of the theorem 7 is that it provides an elegant semantic formulation of the domain of complex database objects endowed with the join and the projection.

5 A Polymorphic Language for Databases

We now show that the entire type system and the semantic domain we have just constructed can be integrated in an ML-like programming language. Such integration yields a strongly typed polymorphic programming language suitable for databases. An experimental programming language, Machiavelli [45], embodying the integration has been developed at University of Pennsylvania. In this section, we show how the integration is done by defining a subset of Machiavelli. Redders are refer to [45] for discussions of the advantages of such a polymorphic database language and many examples of database programming in the language.

5.1 Types and Expressions

The first step of the integration is to define the set of types and the set of expressions of the language in such a way that the set of description types $\boldsymbol{Dtype}^\infty$ and the set of descriptions $\boldsymbol{Dobj}^\infty$ can be freely mixed with the other constructs of the language. This is done by simply extending term languages for $\boldsymbol{Dtype}^\infty$ and $\boldsymbol{Dobj}^\infty$ we have defined to include function type constructors and function expressions.

The set of types $\boldsymbol{Type}$ (ranged over by t) of the language is given by the following abstract syntax:

$$t ::= b \mid t \to t \mid [l:t, \ldots, l:t] \mid \langle l:t, \ldots, l:t \rangle \mid \{\!\{t\}\!\} \mid (rec\ v.\, t(v))$$

where b stands for the set of base types $\mathcal{B}$, $t(v)$ stands for type expressions possibly containing symbol v. Regarding $t_1 \rightarrow t_2$ as a shorthand for $Fun(Domain = t_1, Range = t_2)$, each type expression t denotes the regular tree $M_t(s) \in R(F_\tau \cup \{Fun\})$ where M_s is the Moore machine defined in section 4.1. We regard $t \in \mathbf{Type}$ as the corresponding regular tree $M_t(s)$. The set of types that do not contain the function type constructor $\rightarrow$ is exactly the set of description types $\mathbf{Dtype}^\infty$.

For expressions of the language, in addition to expressions that denote descriptions, we introduce the following expression constructors:

$(fn(x) => e)$	:	function abstraction
$f(a)$	:	function application
$r.l$	:	field selection form a record r
$modify(r, l, e)$	:	modification of l field in r with e
$(case\ v\ of\ l_1\ of\ x_1 => e_1, \ldots, l_n\ of\ x_n => e_n)$	:	case analysis for a variant
$union(s_1, s_2)$	:	set union
$prod(s_1, s_2)$	:	cartesian product of two sets
$map(f, s)$	:	map a function f to a set s

The set of expressions $\mathbf{Expr}$ (ranged over by e) of the language is then given by the following abstract syntax:

$$\begin{aligned} e \ ::= \ & c \mid x \mid e(e) \mid (fn(x) => e) \mid let\ x = e\ in\ e \mid \\ & [l = e, \ldots, l = e] \mid e.l \mid modify(e, l, e) \mid \\ & Inj(l = e) \mid (case\ e\ of\ l\ of\ x => e, \ldots, l\ of\ x => e) \mid \\ & \{\!\{e, \ldots, e\}\!\} \mid union(e, e) \mid prod(e, e) \mid map(e, e) \mid \\ & join(e, e) \mid project(e, \sigma) \mid (rec\ x.\ e) \end{aligned}$$

where c stands for constants, x stands for variables, $let\ x = e\ in\ e$ stands for ML *let*-expressions. The subset of $\mathbf{Expr}$ defined by by the following abstract syntax

$$d ::= c \mid [l = e, \ldots, l = e] \mid Inj(l = e) \mid \{\!\{e, \ldots, e\}\!\} \mid (rec\ x.\ d(x))$$

denote regular trees and corresponds exactly to the set $\mathbf{Dobj}^\infty$. We identify an expression d and the corresponding description in $\mathbf{Dobj}^\infty$ if d is in the subset specified by the above grammar.

5.2 Type Inference

Different from the explicitly typed language, the expressions we have defined carry no explicit type information. Types of expressions are completely *inferred* by a proof system called a *type inference system*. In [44], a complete type inference system for a language containing database objects without variants and recursive objects is defined. By using the typing relation defined in section 4.4, the type inference system defined in [44] can be extended to the entire set of the above expressions.

In [44], it is also shown that by extending Milner's method [40] for ML type inference with *conditions* on substitutions, we can define a complete type inference algorithm. The method relays on the solvability of unification of type expressions, the decidability of the ordering on description types, and the computability of least upper bounds of description types. Huet showed [28] the solvability of unification problem of regular trees and define a unification algorithm. In section 4.4, we have shown the decidability of the ordering relation and the computability of least upper bounds of description types. Using these two results, the method described in [44] can be extended to the entire language.

5.3 Equality and Reduction Relations on Expressions

On expressions, sets of rules should be defined to represents the equality and the reduction relation on expressions, which determine the dynamic properties of the language. These relations are defined on *typed expressions* derived in the type inference system. The equality rule (rule scheme) for expressions corresponding to descriptions is given as:

(description) $\quad d_1 : t = d_2 : t \quad$ if $d_1, d_2 \in \boldsymbol{Dobj}^\infty, d_1 \preceq d_2, d_2 \preceq d_1$

Since $\preceq$ represent the goodness of descriptions, this rule correctly captures the intended equality on descriptions. The equality rules for the join and the projection are defined as:

(join) $\quad join(d_1 : t_1, d_2 : t_2) \ : \ t = \phi_{t_1 \to t}(d_1) \sqcup_{D_t} \phi_{t_2 \to t}(d_2) \quad$ if $t_1, t_2 \in \boldsymbol{Dtype}^\infty, t_1 \sqcup t_2 = t$

(project) $\quad project(d : t_1, t_2) : t_2 = \psi_{t_1 \to t_2}(d) \quad$ if $t_1, t_2 \in \boldsymbol{Dtype}^\infty, t_2 \leq t_1$

Combining them with the rules for standard equational reasoning, the standard rules for function applications (the rule β), *let*-expressions and primitive operations other than *join* and *project*, we have a complete equality relation for the language.

In order to define a reduction relation, we define the notion of normal form on descriptions. For $d \in \boldsymbol{Dobj}^\infty$, d is *in description normal form* if for any element $d' \in \boldsymbol{Subtrees}(d)$ if $d'(\epsilon) = Set$ then there is no d_1, d_2 such that $d_1 = d'/elm_i$, $d_2 = d'/elm_j$ for some i, j and $d_1 \preceq d_2$. It can be shown that for any $d \in \boldsymbol{Dobj}^\infty$, there is some d' such that d' is in description normal form and $d = d'$ in the sense of the above equality relation. Moreover, such d' can be effectively computed. The reduction rule for descriptions is then given as:

(description) $\quad d_1 : t \to d_2 : t \quad$ if $d_1, d_2 \in \boldsymbol{Dobj}^\infty$ and d_2 is in a description normal form

The reduction rules for the join and the projection are same as the rules for equality. Combining with the rules for standard equational reasoning *except* the rule for symmetry, the standard rules for function applications (the rule β), *let*-expressions and primitive operations other than *join* and *project*, we have a complete reduction relation for the language. Based on this reduction relation, an operational semantics of the language is defined. Actual evaluation algorithm for the language can be defined by using the algorithms for computing least upper bounds of descriptions, embeddings and projections that have been defined in the proofs of their computabilities in the previous section.

5.4 Semantics of the Entire Programming Language

In practice, it is sufficient to have a type inference algorithm to type-check programs and an evaluation algorithm to compute results of programs. For a better understanding of the language, however, it is highly desirable to construct a complete semantics of the entire programming language. Such a semantics should be useful for reasoning about various properties of programs and for further enhancement of the language.

In addition to the semantics of database type system we have constructed, a semantics of the entire language requires a semantics of ML polymorphism. Milner proposed one such semantics [40] using a universal value domain of an untyped language. In his semantics, a type denotes a subset of the universal domain. MacQueen et. al. extended this semantics to recursive types [36]. However, this semantics does not agree with behavior of implicitly typed expressions. (See a [43] for an analysis of this problem.) Recently, Mitchell and Harper showed that [41] there is a one-to-one correspondence between a typing derivation in ML type inference system and a term in a explicitly typed language. Along the line of this connection, it is shown in [43] that a semantics of explicitly typed language yields a semantics of the corresponding implicitly typed language supporting ML polymorphism. It is therefore sufficient to construct a semantics of the explicitly typed version of the language.

We regard expression constructors other than function abstraction and function application as ("curried") constant functions. For example, the record $[Name = "Joe"]$ is regarded as the application $[Name = \_]("Joe")$ of the constant function $[Name = \_]$ to the constant $"Joe"$ and a join $join(d_1, d_2)$ is regarded as the curried application $join(d_1)(d_2)$ of the constant function $join$ to d_1, d_2. Recursive descriptions can be also treated in this way. The explicitly typed language corresponding to our language is then obtained by explicitly specifying the type of parameter in function abstraction as in $(fn(x : t) => e)$ and replace each constant by the corresponding set of *typed* constants. The resulting language is a typed lambda calculus with constants. In [10], a framework for a semantics of typed lambda calculi was given. In the framework, the set of types is generalized to a *type algebra* allowing arbitrary equations (or *constraints*). Since the set of regular trees satisfies their definition of a type algebra, we can use this framework to construct a semantics of the explicitly typed language. In the framework, a semantic space of a language is a *frame* $(\mathcal{F}, \bullet, \gamma)$ where: $\mathcal{F}$ is a $\boldsymbol{Type}$-indexed set such that each $F_t \in \mathcal{F}$ is non-empty, $\bullet$ is a binary operation $\bullet : F_{t_1 \to t_2} \to F_{t_2}$ representing the function application and γ is a function that interprets constants. For our language, we impose the following conditions on a frame $(\mathcal{F}, \bullet, \gamma)$:

1. for each $t \in \boldsymbol{Dtype}^{\infty}$, $(D_t \cup \{\top\}) \subseteq F_t$, where D_t is the description domain we have constructed and $\top$ is a distinguished value representing the exception of the computation of join,
2. for $c \in B_b$, $\gamma(c : b) = c$,
3. for $null_b$, $\gamma(null_b : b) = null_b$,
4. for a constant $f : t$ introduced for a term constructor, $\gamma(f : t)$ is the element in F_t satisfying the intended equations. Such equations are easily defined for each constant based on the structures of database domain we have developed in the previous section. For example, the necessary equations for

$\gamma(join : t_1 \to t_2 \to t) \in F_t$, where $t = t_1 \sqcup t_2$, are given as:

$$(j \bullet (d_1)) \bullet (d_2) = \begin{cases} \phi_{t_1 \to t}(d_1) \sqcup_{D_t} \phi_{t_2 \to t}(d_2) & \text{if lub exists} \\ \top & \text{otherwise} \end{cases}$$

for all $d_1 \in D_{t_1}, d_2 \in D_{t_2}$.

The method described in [10] can then be applied to construct a semantics of the explicitly typed version of our language. A semantics of the implicitly typed language supporting ML polymorphism can be constructed by using the method described in [43]. Based on the semantics, we can show the strong soundness and completeness theorem for the equational theory of our language as is done in [43].

6 Conclusion and Future Works

Based on an abstract analysis of the relational data mode, we have proposed a framework for semantics of types for databases. We characterized a semantic space of individual type as a poset of descriptions, which we called a description domain, and a semantic space of the entire type system as a poset of description domains, which we called a database domain. Based on this framework, we have constructed a concrete database type system and its semantic domain using regular trees supporting arbitrary complex structure constructed from records, variants, finite sets and recursive definitions. On these complex structures, a join and a projection are available as typed polymorphic operations. We have also shown that both the type system and the semantic domain can be uniformly integrated in an ML-like polymorphic programming language.

In our study of database type system, we have implicitly assumed that database objects are *values*. Tow objects are equal if they are equal as values. As we have demonstrated, these value-based database systems are fit nicely to a paradigm of functional programming languages. However, value-based systems have a disadvantage that it is rather difficult to represent *sharing* and *mutability*, which are also important aspects of database objects. In order to overcome this disadvantage, the notion of "object-identities" has been proposed [7, 37, 33]. In an identity-based system, database objects are represented by their unique identities associated with their attribute values. For the same reason as we wanted to integrate value-based database system into a modern type system of a programming language, we would like to integrate identity-based database objects in a types system of a programming language. Although the notion of object identities is intuitively clear and appealing, integrating it into a programming language type system constitutes a challenge. As demonstrated in [45], the major properties of object identities seems to be captured by ML *reference type* when integrated in a database type system like the one we have developed in this paper. However, a uniform and elegant integration will require an analysis of the properties of object identities analogous to what we have done to the structure of value-based complex database objects.

Acknowledgement

I would like to thank Peter Buneman and Val Breazu-Tannen for discussions and many helpful comments on this paper.

References

[1] S. Abiteboul and N. Bidoit. Non First Normal Form Relations to Represent Hierarchically Organized Data. In *Proc. 3rd ACM PODS*, Waterloo, Ontario, Canada, 1984.

[2] S. Abiteboul and R. Hull. Restructuring of Complex Objects and Office Forms. In *Proc. International Conference on Database Theory, Lecture Notes in Computer Science 243*, Springer-Verlag, Rome, Italy, September 1986.

[3] H. Aït-Kaci. *A Lattice Theoretic Approach to Computation based on Calculus of Prtially Ordered Type Structures.* PhD thesis, University of Pennsylvania, 1985.

[4] A. Albano, L. Cardelli, and R. Orsini. Galileo: A Strongly Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[5] 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.

[6] M.P. Atkinson and O.P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, June 1987.

[7] F. Bancilhon. Object-Oriented Database Systems. In *Symposium on Principles of Database Systems*, ACM SIGART-SIGMOD-SIGACT, Austin, Texas, March 1988.

[8] F. Bancilhon and S. Khoshafin. A Calsulus for Complex Objects. In *Proc. ACM Conference on Principles of Database Systems*, 1986.

[9] J. Biskup. A Formal Approach to Null Values in Database Relations. In *Advances in Data Base Theory Vol 1*, Prenum Press, New York, 1981.

[10] V. Breazu-Tannen and A. R. Meyer. Lambda calculus with constrained types (extended abstract). In R. Parikh, editor, *Proceedings of the Conference on Logics of Programs, Brooklyn, June 1985*, pages 23–40, *Lecture Notes in Computer Science*, Vol. 193, Springer-Verlag, 1985.

[11] K. B. Bruce and P. Wegner. An Algebraic Model of Subtypes in Object-Oriented Languages. *SIGPLAN Notices*, 21(10):163–172, October 1986.

[12] P. Buneman, S. Davidson, and A. Watters. A Semantics for Complex Objects and Approximate Queries. In *Proceedings 1988 ACM Symposium of Principles of Database Systems*, March 1988. Full version to appear in Journal of Computer and System Sciences.

[13] P. Buneman and A. Ohori. Using Powerdomains to Generalize Relational Databases. *Theoreical Computer Science*, To Appear, 1988. Available as a technical report from Department of Computer and Information Science, University of Pennsylvania.

[14] L. Cardelli. A Semantics of Multiple Inheritance. In *Semantics of Data Types, Lecture Notes in Computer Science 173*, Springer-Verlag, 1984.

[15] L. Cardelli. *Amber.* Technical Memorandum TM 11271-840924-10, AT&T Bell Laboratories, 1984.

[16] M. Carey, D. DeWitt, Richardson J., and E Sheikta. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of the 12th VLDB Conference, Kyoto, Japan*, August 1986.

[17] G. Copeland and D. Maier. Making Smalltalk a Database System. In *Proceedings of ACM SIGMOD*, pages 316–325, ACM, June 1984.

[18] B. Courcelle. Equivalences and Transformations of Regular Systems – Applications to Recursive Program Schemes and Grammars. *Theoretical Computer Science*, 42:1–122, 1986.

[19] B. Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983.

[20] 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.

[21] C.J. Date. *An Introduction to Database System Vol 1.* Addison-Wesley, third edition, 1981.

[22] P.C. Fischer and S.J. Thomas. Operators for Non-First-Normal-Form Relations. In *Proc. IEEE COMPSAC*, 1983.

[23] G. Gierz, H.K. Hofmann, K. Keimel, J.D. Lawson, M. Mislove, and D.S. Scott. *A Compendium of Continuous Lattices.* Springer-Verlag, 1980.

[24] S. Gorn. Explicit definitions and linguistic dominoes. In J. Hart and S. Takasu, editors, *Systems and Computer Science*, pages 77–105, University of Toronto Press, 1967.

[25] R. Harper, D. B. MacQueen, and R. Milner. *Standard ML.* LFCS Report Series ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, March 1986.

[26] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 1979.

[27] J.E. Hopcroft and R.M. Karp. *An Algorithm for Testing the Equivalence of Finite Automata.* Technical Report TR-71-114, Department of Computer Science, Cornell University, Ithaca, NY., 1971.

[28] G Huet. *Résolution d'équations dans les langages d'ordre 1,2,...ω.* PhD thesis, University Paris, 1976.

[29] R. Hull. *A survey of theoretical research on typed complex database objects*, chapter 5, pages 193–253. Academic Press, 1987.

[30] R. Hull. Relative information capacity of simple relational database schemata. *SIAM J. Computing*, 15(3):856–886, August 1986.

[31] R. Hull and C.K. Yap. The Format Model: a theory of database organization. *Journal of the ACM*, 31(3):518–537, 1984.

[32] G Jaeschke and H. Schek. Remarks on algebra on non first normal forn relations. In *Proc. of the ACM-SIGACT-SIGMOD Symposiumon Principles of Database Systems*, pages 124–138, Los Angeles, March 1982.

[33] S.N. Khoshafian and G.p. Copeland. Object Identity. In *Proc. OOPSLA*, pages 406–416, September 1986.

[34] Y. Lien. On the Equivalence of Database Models. *Journal of the ACM*, 29(2):333–362, April 1982.

[35] W. Lipski. On Semantic Issues Connected with Incomplete Information Databases. *ACM Transactions on Database Systems*, 4(3):262–296, September 1979.

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

[37] D. Maier. A Logic for Objects. In *Proceedings of Workshop on Deductive Database and Logic Programming*, Washington, D.C., August 1986.

[38] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[39] D Maier. Why Database Langauges Are a Bad Idea. In F. Bancilhon and P. Buneman, editors, *Workshop on Database Programming Languages*, Addison-Wesley, 1989. To Appear.

[40] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[41] 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, California, January 1988.

[42] P O'Brien, B Bullis, and C. Schaffert. Persistent and Shared Objects in Trellis/Owl. In *Proc. of 1986 IEEE International Workshop on Object-Oriented Database Systems.*, 1986.

[43] A. Ohori. Semantics of ML Polymorphism. 1988. Unpublished Manuscript, University of Pennsylvania.

[44] A. Ohori and P. Buneman. Type Inference in a Database Programming Language. In *Proc. ACM Conference on LISP and Functional Programming*, pages 174–183, Snowbird, Utah, July 1988.

[45] A. Ohori, P. Buneman, and V. Breazu-Tannen. *Database Programming in Machiavelli – a Polymorphic Language with Static Type Inference.* Technical Report, Department of Computer and Information Sience, University of Pennsylvania, December 1988.

[46] Z. Özsoyoğlu and L. Yuan. A new normal form for nested relations. *ACM Transactions on Database Systems*, 12(1):111–136, March 1987.

[47] J.C. Reynolds. Towards a Theory of Type Structure. In *Paris Colloq. on Programming*, pages 408–425, Springer-Verlag, 1974.

[48] A.M. Roth, H.F. Korth, and A. Silberschatz. *Extended Algebra and Calculus for ¬1NF Relational Databases.* Technical Report TR-84-36, Department of Computer Sciences, The University of Texas at Austin, 1984. revised 1985.

[49] W. Rounds. *Set values for unification-based grammar formalisms and logic programming.* CSLI Technical Report CSLI-88-129, Center for Study of Language and Information, June 1988.

[50] W. C. Rounds and R. Kasper. A complete Logical Calculus for Record Structures Representing Linguistic Information. In *IEEE Symposium on Logic in Computer Science*, 1986.

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

[52] J.W. Schmidt. Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems*, 5(2), 1977.

[53] E. Sciore. *Null Values, Updates, and Incomplete Information.* Technical Report, Department of Electrical Engineering and Computer Science, Princeton University, 1979.

[54] D. Scott. Domains for Denotational Semantics. In *ICALP*, July 1982.

[55] S.M. Shieber. An Introduction to Unification-Based Approaches to Grammar. In *Proc. 23rd Annual Meeting of the Association for Computational Linguistics*, 1985.

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

[57] J.D. Ullman. *Principle of Database Systems.* Pittman, second edition, 1982.

[58] C. Zaniolo. Database Relation with Null Values. *Journal of Computer and System Sciences*, 28(1):142–166, 1984.