# EP Data Model - a Language for Higher-Order Functions

Kevin Houzhi Xu[1]

**Abstract**

Searching for a better data model is not a new topic, but it should not become old one either before a satisfaction can be reached. Improving the productivity and quality of developing database systems is the motivation driving the research activities. This paper is approaching this direction by taking sub function spaces from λ-models. The developed data model, called Enterprise-Participant (EP) data model, is to store and construct effectively computable functions from λ-models; and its *db*-terms, similar to combinatory-logic terms or λ-terms, are the only syntactical form of the expressions referencing the functions, and the expressions computing the functions. The ordering relations among the *db*-terms, as built-in logic functions, express queries and updates which are usually called "fixpoint queries" in the relational query languages. As both a language and a data type, the EP data model connects the paradigm of functional programming languages with database management. The EP data model is a "universal" data abstraction for data integration and distribution.

**Key Words**: data model, data structure, language, λ-model, computable function, normal form, consistence, soundness, completeness, applicative structure, active domain, ordering relation, first order language, functional programming language.

## 1 Introduction

The less implementation details developers have to deal with, the faster and easier it will be in software development and maintenance. In other words, higher-level descriptive languages are preferred to specify "what", but not "how". This is called layering in network computing, applicative or functional in programming languages, abstract in data types, and semantic in data models. While a language should be highly descriptive to be simple, however, it must be expressive to be useful in certain application areas. In this paper, we will discuss the descriptiveness and expressiveness of data models in the database application area, and propose a data model with higher descriptiveness and expressiveness. A data model normally offers data structures and the operations against the data structures [35]. Instead of the operations, the data structures of data models are focused in regard to the descriptiveness and expressiveness.

What is expressiveness? In database query languages, the expressiveness is to categorize the classes of queries or functions a language can express. For example, Datalog is more expressive than the relational calculus because the former can express fixpoint queries against relations, but the later cannot [26], [5. Similar to the expressiveness in database query languages, the pure λ-calculus would be said more expressive than a typed λ-calculus in programming languages because the former can express partially recursive functions, but the later only recursive functions [15]. When a language can express partially recursive functions, it is said Turing-machine equivalent. For example, The mathematical models of all the machine languages, assembly languages, imperative languages, functional languages are Turing-machine equivalent. On the other hand, the data structures of data models are normally not rigorously characterized by classes of functions, or expressiveness. For example, the term "flatness" of relational model, and the term "more semantic" in semantic models and object-oriented data models are the primary terms of characterizing data models [10], [17], [12]. However, when one would like to view data

[1] 164 West High Street, Bound Brook, New Jersey 08805.

structures as languages, the data structures could be ranked by expressiveness. For example, relational databases were ranked as functions of order 3 when they are mapped to typed λ-terms in [14].

What is descriptiveness? The author would not take any risk to propose an absolutely definition of it. But the common sense was that the closer to machines a language need to reference to, the less descriptive the language is [3], [6]. The degree of descriptiveness is increased in the order of machine languages, assembly languages, imperative languages, and applicative languages. This is why functional and logic programming languages are intensely focused in the programming language research. The tables in relational databases, where finite atoms are the values of attributes, are more descriptive than imperative programming languages in traditional business-related database applications. This was why database management systems emerged from imperative programming languages in database applications in 1960's. This common sense would satisfy us up to now, but we will come back to this point at the end of the paper for a more measurable term.

It is unfortunate that traditional data models are usually applied to some database application areas beyond the scopes of what they are supposed to do. When this is happened, the data model must be accompanied with remedies to fully support its tasks. The remedies are languages other than data models themselves. Then the overall descriptiveness of developing database application systems is degraded. The degradation of the descriptiveness is happening with the relational data model and object-oriented data models. For example, constraint programming is a remedy for infinite data [11]; constraints and triggers of relational database management systems for functional dependency [8]; and methods in object-oriented data model partly for data constructions.

Remedies are required too in data communication. Data models for storing persistent data may not be appropriate for data communication, and common data protocols (or data models) for data communication may not be appropriate for storing persistent data. Therefore, data interpreters become critical components in distributed or integrated database management systems [38]. X.500 is an example of the data communication protocols [31]. CORBA [30] in object-oriented database management systems seems the practical solution for data communication, but it is a purely imperative paradigm.

Another example of remedying the shortage of traditional data models would be managing application data with ordering relations such as organization hierarchies, family trees, and graphs. The relational data model is able to flatten them, but more expensive query languages like Datalog are required to recover the ordering relations of data, which are so called fixpoint or transitive-closure queries [1], [2]. To maintain the consistence of the "flattened" ordering data with the dynamic world, a remedy is required from host programming languages to support update operations. The notion of composite objects with hierarchical structures is useful in certain applications 18. But it is indeed a remedy because it is not universal.

It is the purpose of this paper to propose a new data model, called Enterprise-Participant (EP) Data model, to overcome the shortages of traditional data models. Let's "forget" the traditional data models and their applications for a while. We know that a computer is semantically constructing nothing, but functions. And a λ-model offers all the continuous functions including effectively computable functions [28], [37], [27], [33], [21]. The EP data model would be said expressive if it can manage an arbitrary finite subset of the elements from a λ-model.

The EP data model is not the first in applying functions to data models or data structures. Structural data like those in the record type in programming languages was interpreted as (aggregate) functions [22]; A functional data model was proposed for database applications [29]; and Relations were interpreted as the functions of order 3 in a typed lambda calculus [14]. However, it is only the EP data model that takes full advantages of the applicative behavior of λ–models, that is, an element in a λ–model could act as a function, an argument, and a value of an application. This leads to the ordering relations of the EP data model introduced in this paper for the purpose of database queries and updates. When the other

functional approaches only say that a data could be interpreted as functions, the EP data model also says that any function (any element) of a λ–model could be managed by the EP data model as long as it is λ–definable (or equivalently expressible by a Turing-Machine equivalent language).

Taking a λ–model as the function domain is not new because it has been done for λ–calculi and practical programming languages. The new is the EP data model itself as a language. The EP data model offers the uniformed syntactical form of sentences – *db*-terms. Similar to *CL*-terms of Combinatory Logic or λ–terms of λ–calculi, a *db*-term represents a function from a λ–model, and is an expression from which a normal form may be reduced. Different from them which are self-defined if they are closed terms, a *db*-term has either an assigned meaning, or a derived meaning from other relevant *db*-terms defined in an EP database. A *db*-term identifies a function and identifies the relationships of the function with others. Ordering relations among the *db*-terms become built-in operators in queries and updates against functions managed by EP databases. It is our intention in this paper to say that all the data in database applications is viewed as functions; and all the functions managed by EP databases are application data. Simple atomic data such as integers and strings (or files), has the counterpart of atom functions; and complex data such as hierarchical organizations, graphs, multimedia data, and 3$^{rd}$-party applications to higher-order functions.

As a language, the EP data model is introduced in Section 2, 3, 4, 5, and 6. Section 2 defined the EP data model and EP databases. A few examples follows in Section 3. Sections 4, 5, and 6 give the definition of the normal forms of *db*-terms, the semantics of the EP data model, and the computation of the EP data model. Starting from Section 7, we try to understand more about *db*-terms as sets, from which we dig out a few ordering relations for further database queries and update operations. In Section 11, we informally discuss that the EP data model is just the first step toward the complete solution of database applications. An extended functional programming languages sugared by the EP data model is necessary, as if the predicate calculus had its foundation of the propositional logic. We further give a more measurable definition of descriptiveness to analyzing why the "descriptive" functional languages are not popular in database application practice, and how the EP data model has a chance to energize the popularity of functional programming languages. Section 12 concludes this paper.

We will feel free to use logical symbols from first-order language [7], [32] and λ–terms from λ–calculi [4], [7], [24] to express what we need in this paper. Since well-formed formulas (wffs) in the first order logic are logic functions, we sometime also use the form of the wffs in lambda terms as long as there is no ambiguity in context. And also the equation symbol = is used as equivalence relations across many domains, such as between two sets, two *db*-terms, two λ–terms, and two elements in a λ–models.

## 2 EP Data Model

The EP data model was first proposed in [38]. It was intended to be more powerful in practice than the traditional data models. For more information, readers can see the examples in the next section, and reference [38]. The EP data model discussed in this paper is an improvement from its origin after a close study on the relationship of the EP data model with the theories of functions, λ–models, and λ–calculi.

A data model is traditionally viewed as an algebra consisting of data structures (values) and operations against the data structures as we pointed out earlier. The EP data model, however, is developed as a language with a set of symbols, *db*-terms as sentances, normal forms, semantics, and computational rules. An EP database, analogous to an environment in languages, assigns values to a finite subset of the *db*-terms, from which other db-terms have derived values.

**2.1 Definition** The following symbols are allowed in the EP data model:

A countable set of constant symbols, denoted as $\mathcal{C} = \{\perp, c, c_1, c_2, \ldots\}$;

~~A countable set of constant symbols, denoted as $\mathcal{C} = \{\perp, c, c_1, c_2, \ldots\}$;~~

A countable set of proposition letters, denoted as $\mathcal{P} = \{a, a_1, a_2, \ldots\}$;

Special constant symbols *tag*, $\phi$.

Parentheses: $(_*)$.

Like the propositional logic, proposition letters are named to mean that they have fixed meanings under a single interpretation (an EP database). The special constant symbol $\perp$, a symbol normally representing the least defined element in a $\lambda$–model [4], is used to mean undefined, or meaningless. And $\perp_E$ will be used in this paper to stand for the least defined element in a $\lambda$–model. The special symbols *tag* and $\phi$ will be explained later when the definition of EP databases is given.

$\mathcal{C}$ is a countable set of constant symbols, each of which will represent a function. A function can be simple typed values like integers, built-in higher-level functions like +, Multimedia objects, an existing EP database, or an existing (or called 3rd-party) computing application.

**2.2 Definition**. The set of terms $\mathcal{T}$, called *db*-terms, is defined inductively as follows:

$a \in \mathcal{P} \Rightarrow a \in \mathcal{T}; c \in \mathcal{C} \Rightarrow c \in \mathcal{T}$

$M, N \in \mathcal{T}$, then $(M\,N) \in \mathcal{T}$

**2.3 Notation**. 1. $M, N, L, \ldots$ denote arbitrary *db*-terms.

2. The symbol $\equiv$ denotes syntactic equality
3. Outermost parentheses are not written. Let $M\,N_1 \ldots Nn \equiv (\ldots((M\,N1)\,N2) \ldots Nn)$.
4. Let $\mathcal{T}(\mathcal{P})$ denotes the *db*-terms with proposition letters only. For example, $a$, $a\,a_1 \in \mathcal{P}$, but $c\,a \notin \mathcal{P}$.

□

Like the combinatory logic or the lambda calculus [4], the combination $(M\,N)$ of *db*-terms is the main syntactical form of constructing *db*-terms. And each *db*-term represents a function, and each combination $(M\,N)$ a function application. Different from the combinatory logic, where variable-free terms have self-defined meanings, some *db*-terms in $\mathcal{T}(\mathcal{P})$ have assigned meanings, and the others in $\mathcal{T}$ have derived ones. The collection of assignments will form an environment (or valuation), that is, a mapping function

$tag: \mathcal{T}(\mathcal{P}) \rightarrow \mathcal{T} \cup \mathcal{C}$.

Since certain restrictions are imposed on environments, we will give the definition of EP databases which carries the restrictions and the environments.

**2.4 Definition** An EP database is a finite set $\mathcal{D} \subset \mathcal{T}(\mathcal{P})$ such that

1. if $M\,N \in \mathcal{D}$, then $M \in \mathcal{D}, N \in \mathcal{D}$.
2. if $M \in \mathcal{D}$, and there is no other $N \in \mathcal{D}$ such that $M\,N \in \mathcal{D}$, then $M$ may be (but don't have to be) assigned a *db*-term $Q \in \mathcal{T}$, denoted $tag(M) \equiv Q$.

An environment usually assigns meanings to variables in regular languages. But here, an EP database assigns meanings to some *db*-terms $M \in \mathcal{T}(\mathcal{P})$ and $M \in \mathcal{D}$. The *db*-terms in $\mathcal{D}$, which have not assignments, will have derived meanings as we will see in the following sections. An EP database imposes a restriction that if $M\,N \in \mathcal{D}$, then $M \in \mathcal{D}$, $N \in \mathcal{D}$, and $M$ and $N$ must not have assigned meanings, but derived ones.

To distinguish *db*-terms with assigned meanings from those with derived meanings, we give the following notations.

**2.5 Notation** 1. If a *db*-term $M$ in $\mathcal{D}$ is not assigned a value, or has not tag, then we denote $tag\,(M) \equiv \phi$, and $M$ is called a non-leaf *db*-term. The symbol $\phi$ means empty, no tag, or no assignment.

2. If $M$ has a tag $N$, that is, $tag\ (M) \equiv N$, then $M$ is written as $M\ (\ tag \equiv N)$ in the definition of a EP database. And $tag\ (M) \equiv N$ may also be interchangeably written as $M.tag \equiv N$ in this paper. □

Similar to λ–calculi, the subterms is defined as

**2.6 Definition** 1. $M$ is a subterm of a *db*-term $N$ (notation $M \subset N$) if $M \in Sub(N)$, where $Sub(N)$, the collection of subterms of $N$, is defined inductively as follows:

$N \equiv a$ and $a \in \mathcal{P} \Rightarrow Sub(N) \equiv \{a\}$;

$N \equiv c$ and $c \in \mathcal{C} \Rightarrow Sub(N) \equiv \{c\}$;

$L_1\ L_2 \equiv N \Rightarrow Sub(N) = Sub(L_1) \cup Sub(L_2) \cup \{\ L_1\ L_2\ \}$

2. A subterm occurrence $M$ of $N$ is active if $M$ occurs as $(MZ) \subset N$ for some $Z \in \mathcal{I}$; otherwise $M$ is passive.

Before we give the interpretation of *db*-terms, the following section will give some examples to show what a EP database looks like. As a collection of meaningful *db*-terms, the concept of EP databases is the central of the EP data model, and the central of applying the EP data model to database application practice.

## 3 Database Design I

Designing databases in the EP data model may not need to understand all the formal notations presented in this paper. The idea, as presented in [38], was that any object in the real world, called enterprise, can be viewed as a collection of sub objects (its components). On the other hand, the participation of an object to other objects partially defines or determines the definition of the other objects. This philosophy matches the applicative characteristics of higher-order functions. That is, each object in the world can be (approximately) represented by a function. Its components are the set of argument-value pairs defined by the function. When object $M$ participate to the "activity" of object $N$, $N$ is the function, $M$ the argument, and $N\ M$ the value. Then we say that $<M, N\ M>$ is the argument-value pair which partially determines the definition of the function $N$.

Before we give examples, a graphical presentation of EP databases will be described. The graphical presentation makes EP databases easier to read, and it is the traditional way of presenting a data model. Or it could be a data type in programming languages.

**3.1 Notation** The definition of a EP database $\mathcal{D}$, as defined in Definition 2.4, can be graphically described as a set of nodes, a set of up-down solid lines, a set of dash arrows, and a set of solid arrows:

1. Nodes. A node is a circle representing an element $M$ in $\mathcal{D}$. If $M \equiv a \in \mathcal{P}$, then the node is labeled a name '$a$'. And if $M = N_1\ N_2$, then the node $M$ is labeled a name '$N_2$'. If $M$ has a tag, then display the tag below the name. Note that constants including $\perp$, which are in $\mathcal{D}$, are not displayed in the graph.
2. Solid lines. For any $M$, $M\ N \in \mathcal{D}$, an up-down solid line is used to link Node $M$ with Node $M\ N$. The solid lines represents rator-application relationships which will be defined late.
3. Dash arrows. For $N$, $MN \in \mathcal{D}$, a dash arrow may be optionally used to link Node $N$ from Node $MN$. In this case, the name of $MN$ may be renamed for convenience. Dash arrows reflect rand-application relationships which will be precisely defined late. When $N$ is not displayed, there is no dash arrows. But from context, the rand-application relationship of $N$ and $MN$ is clear.
4. Solid arrow. For $M \in \mathcal{D}$ and $M.tag$ exists, instead spelling the whole tag of $M$ in the circle, we draw a solid arrow from $M$ to $M.tag$.

Since EP databases will be interpreted as functions, the first example will show how finite functions are defined by an EP database.

**3.2 Example** Extensional definitions of functions. We define the four functions: the square function $\lambda n.\ n^2$, symbolized as *SQ*; the square root function $\lambda n.\ \sqrt{n}$ , symbolized as *Root*; the identity function $\lambda n.\ n$, symbolized as *I*; and composite function $\lambda f,g,n,$ symoblized as *C*. The *n* in functions *SQ*, *I* , and *C* is ranged over integers *2* and *3*; the function *Root* over 4 and 9. And in the function *C*, *f* ranges over *SQ*, *g* over *SQ* and *Root*.

The textual presentation of the database would be: { *SQ*, *SQ 2* (*tag*≡*4*), *SQ 3* (*tag* ≡*9*), *Root*, *Root 4* (*tag* ≡*2*), *Root 9* (*tag* ≡*3*), *I*, *I 2* (*tag* ≡*2*), *I 3* (*tag* ≡*3*), *C*, *C SQ*, *C SQ Root* (*tag* ≡*I*), *C SQ SQ*, *C SQ SQ 2* (*tag* ≡*16*), *C SQ SQ 3* (*tag* ≡*81*)}.

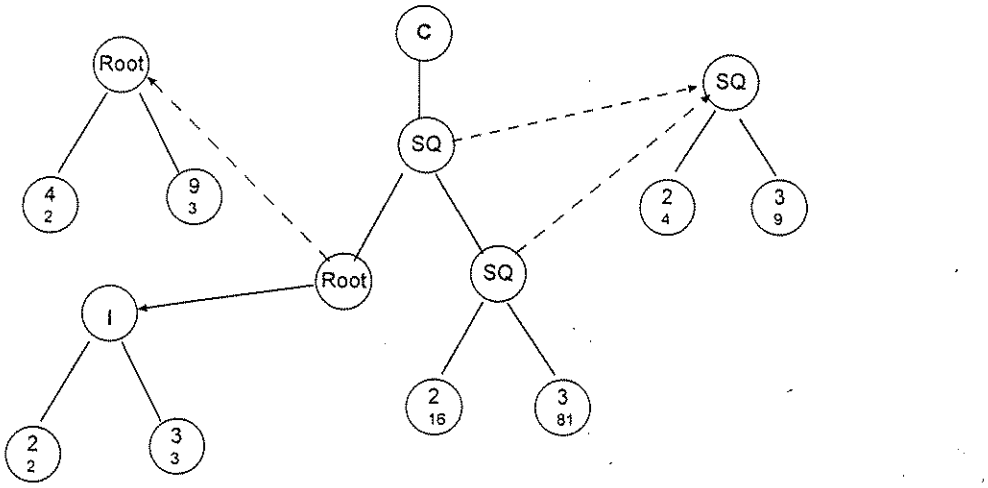


Fig. 1. Extensional Definition of Functions in EP data model

In this example, *SQ*, *Root*, and *I* are defined independently as shown in Fig. 1. Dependent on the definitions of *SQ*, *Root*, and *I*, the composite function *C* is defined as the single pair { <*SQ*, *C SQ*> }, where *C SQ* is further defined by its sub hierarchical structure. While *C SQ SQ* is extensionally spelled out as the set {<*2*, *16*>, <*3*, *81*>}, *C SQ Root* is defined to be equal to the function *I* by assigning a tag. Note that the function *I* could have been spelled out under *C SQ Root*, and the *C SQ SQ* could have been assigned a tag pointing to another function $\lambda n.\ n^4$. The choices of either tag assignments or the extensional constructions of sub hierarchical structures allow different data relationships to be handled differently. We will come back to this point when we talk about database updates in Section 10.

This example showed what a EP data model looks like. How to apply the EP data model into database application practice is the real issue of database designs. We assumed that any database application is effectively Turing computable, and the EP model can be used to capture effectively computable functions as we will see in late sections. Is it intuitive for database design even for a programmer who has no knowledge of functional programming or the lambda calculus? The next example shows an EP database for a typical traditional database application.

**3.3 Example** A school administrative database application. Before the school administrative database system is constructed, the first object is the function for Social Security Department, where residents nation-wise have records about social security numbers, birth dates, and others. As a function, the college consists of multiple departments and an administration office. The administration office records the registration number, enrollment date, and others of each student. And the department of computer science offers multiple classes including *CS 100*, in which John studies with grade *'A'*. Like nodes in the example above, each node in this example also represents a function. And the functions have relationships among them indicated by lines and arrows in Fig. 2. John is a resident as defined under *SSD John* that has a birth date and a social security number. John is a student in a College as given under *College Admin*, and he takes a class *CS100*. Note that two occurrences of names "*John*" are renamed. The complete names should be (*College Admin John*) and (*College CS CS100 John*) respectively. Nevertheless, the dash arrows make the relationships not having ambiguity.

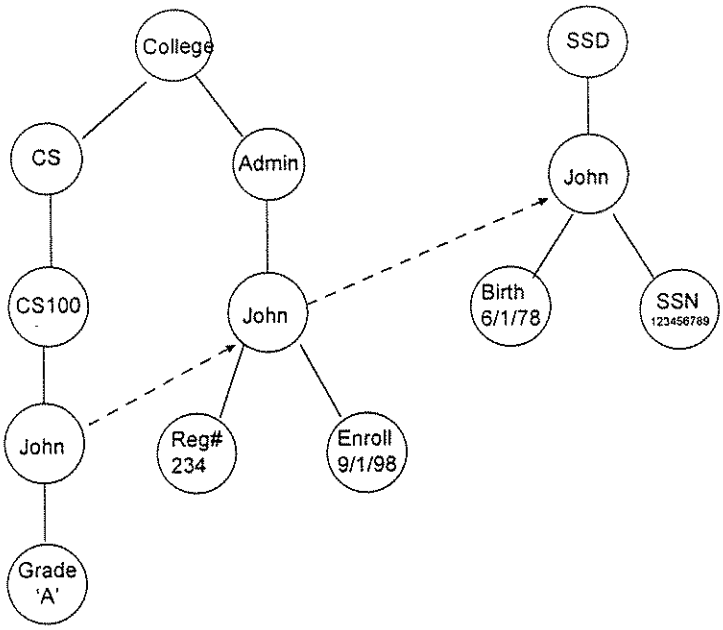


Fig. 2. A School Administrative Database inEP Data Model

What are *CS* (Computer Science), *Admin* (Administration), *Grade*, and etc.? They are distinguished constants representing certain meanings as they do in English. Since this database is not concerning the definitions of these constants, they are not displayed in Fig. 2 and do not have to be precisely defined in the database. But conceptually they are interpreted as functions as we will see in Section 5.

From the above two examples, we see that db-terms offer a naming scheme referencing individual functions, and each function is referenced at least by one *db*-term. For example, *College* references the entire organization of a college; and *College Admin John Enroll* references the enrollment date of John.

**3.4 Example** Infinite data. We use an example similar to one given in 1, a plane in which there are a black square and a white circle (see fig. 3). The shapes of the two objects *obj1* and *obj2* are given by

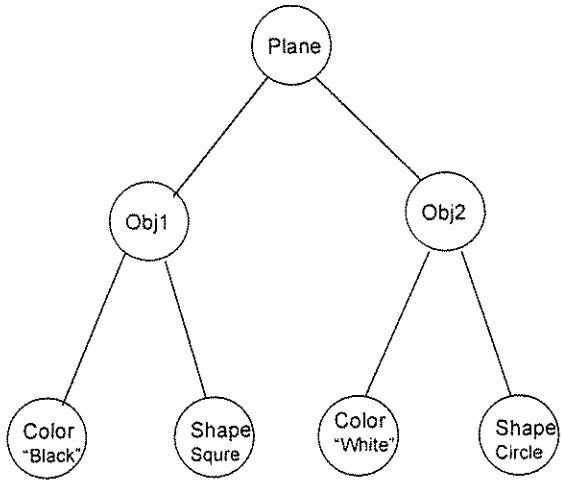


Fig. 3. A EP database for a plane with infinite data

tags:

$$Square \equiv \lambda\, x, y.\ 0 \le x \le 10 \wedge 0 \le y \le 10.$$

$$Circle \equiv \lambda\, x, y.\ (x - 10)^2 + (y - 10)^2 \le 5^2.$$

Here, the functions are given by a "sugared" lambda calculus, where the integer constants, mathematical operators, and boolean operators can be viewed as constant functions. The both functions return true or false value by taking a pair of numbers (coordinators) as arguments.

From this example, we can see that the EP model is able to construct finite presentations for infinite data. Further more, it takes advantage of the applicative characteristics of higher-order functions and offers the uniformly syntactical naming scheme referencing individual infinite data - *db*-terms. For example, given the coordinate <*5*, *5*>, the query of "if the coordinate <*5*, *5*> is in *obj1*" is given by the *db*-term: *Plane Obj1 Shape 5 5*.

## 4 Normal Form

*The db*-terms of the EP data model not only provide the naming scheme of data (objects), they themselves are also query expressions. The following three examples show the necessity of reducing a db-term to another:

1. What is the grade John get in the class *CS 100* in the *CS* department of the *College* in Fig. 2? The query expression in the EP data model is *College CS CS100 John Grade*, and the answer is *'A'*.
2. What is the result of *C* after applying to *SQ*, *Root*, and *2* consecutively in Fig 1? The query expression and the answer in *db*-terms for this query are *C SQ Root 2* and *2* respectively.
3. Is the coordinator <*5*, *5*> in *obj1* of *Plane* in Fig. 3? The query expression and the answer in *db*-terms for this query are *Plane Obj1 Shape 5 5* and *true* respectively.

Before we can think of normal forms and reductions for arbitrary *db*-terms in $\mathcal{J}$, let's focus on the *db*-terms in a EP database $\mathcal{D}$. We will define the equality formula and the normal forms. Two *db*-terms in $\mathcal{D}$ are equal if they are identical or one is the tag of another.

**4.1 Definition** Let $\mathcal{D}$ is an EP Database. $M =_{EP} N$ if $M, N \in \mathcal{D}$ and,
1. $M.tag \equiv N$. Or,
2. $M \equiv N$

*Notation*. If $M =_{EP} N$, it is convenient to write $M = N$.

It is possible that there exists tag circles in $\mathcal{D}$, that is,

**4.2 Definition** A EP database $\mathcal{D}$ has a *tag circle* if there is a sequence of elements $N_1, N_2, \ldots, N_n$ in $\mathcal{D}$, where $n \geq 1$, such that each element has a tag, and $N_1.tag \equiv N_2$, $N_2.tag \equiv N_3$, ..., $N_{n-1}.tag \equiv N_n$, and $N_n.tag \equiv N_1$.

Tag circles should be prohibited in EP database application practice because it defines nothing, but wastes computer space. Therefore, it is not allowed in EP databases. We give two more restrictions on the definition 2.4 of EP databases.

**4.3 Definition** An EP database $\mathcal{D}$ given in Definition 2.4 must further satisfy the following restrictions:
1. there is no tag circle in $\mathcal{D}$.
2. if two db-terms $N_1$ and $N_2$ are not identical, but $N_1 = N_2$, and $M N_1 \in \mathcal{D}$, then $M N_2 \notin \mathcal{D}$.

The restriction 2 above said that an EP database should not give a conflict or redundant definition to a function. Since each *db*-term will be interpreted as a function. And the value of a function by applying to an argument is unique. Therefore, if $M N_1$ and $M N_2$ were *db*-terms in $\mathcal{D}$, and $N_1 = N_2$, then we must request that $M N_1 = M N_2$, which is redundant. Or $M N_1$ and $M N_2$ give the conflict definition for $M$.

Now, let's introduce the normal forms of *db*-terms. Like the normal forms of lambda terms, we intend the normal forms of db-terms to be the unique values of *db*-terms, which can not be reduced further.

**4.4 Definition** Let $\mathcal{D}$ is an EP database of the EP data model.
1. A *db*-term $M \in \mathcal{J}$ is a normal form (or say in normal form, denoted as *db-nf*) if $M$ is either a constant $c \in \mathcal{C}$ or a non-leaf *db*-term ($M$ has no tag) in $\mathcal{D}$.
2. $M$ has a normal form if there exists an $N$ such that $N = M$ and $N$ is a normal form.
3. Let *DB-NF* ($\mathcal{D}$), or interchangeably *DB-NF*, is the set of the *db*-terms in normal forms. That is, *DB-NF* ($\mathcal{D}$) = $\mathcal{C} \cup \{M \mid M$ is a non-leaf in $D\}$.

For example, all the integers, constant functions like *plus* +, $\perp$, and *College CS* in Fig. 2 are in normal form. But + *2 2*, *College CS CS100 John Grade* in Fig. 2, and *SQ 2* in Fig. 1 are not in normal form.

Like the normal forms of λ–terms, *db-nfs* uniquely represent functions in a λ–model. But differently, The latter represents arbitrary functions including the least defined element $\perp_E$ in a λ-model, a partially defined (recursively enumerable) functions, while the former represents only recursive functions. Another difference is that the set *DB-NF*, or precisely *DB-NF*($\mathcal{D}$), relies on a specific EP database $\mathcal{D}$. The set *DB-NF* varies from one EP database to another.

In Section 6, we will see that an arbitrary *db*-term $M$ in $\mathcal{J}$ may be uniquely converted to one of the normal forms under a EP database $\mathcal{D}$.

## 5 Semantics

With *db-nfs* defined, we can assign meanings to them. All the *db*-terms in normal form will be interpreted as an element (or a function) in a λ–model; and each element in the lambda model can be either pre-constructed as a constant, or constructed as a non-leaf db-term in an EP database $\mathcal{D}$. Therefore, the function space (or domain) in a λ–model is the one for the EP data model.

We choose λ–models for the interpretation of the EP data model because *db*-terms in $\mathcal{J}$ have the similar applicative characteristics of lambda-calculi, and lambda models effectively interprets the behavior of the EP data model. It is well-known that a λ–model defines the upper bound for all the computable functions a computer can do. There are many λ–models such as the Scott $D_\infty$ and the graph model Pω for the type-free lambda calculus (Chapter 18 of [4]); and $E \approx A + [E \to E]$ for an applied lambda calculus with constants [37], [33]. In this paper, the following notions from the previous work are taken for granted:

**5.1 Definition**. 1. Let $C_A$ be a set of primary (atomic) constants. $\Lambda(C_A)$ is the λ-terms possibly containing constants from $C_A$. The lambda calculus with λ-terms $\Lambda(C_A)$ has its obvious syntax, axioms, and rules. as described in [19], and the chapter 5 of [4].
2. Let $E$ be the λ–model for the lambda calculus with $\Lambda(C_A)$ as its λ-terms, where the domain $E$ is isomorphic with its continuous function space $A + [E \to E]$, denoted $E \approx A + [E \to E]$, and $A$ has a one-one mapping with $C_A$ in $\Lambda(C_A)$.
3. $\forall c \in C_A$, $\forall M \in \Lambda(C_A)$, $c\ M = \Omega$. And correspondingly, $\forall e \in A$, $\forall M \in E$, $e\ M = \perp_E$ (suggested by [37]). Here Ω is an unsolved λ–term as it will be given in the following definition

Closed λ–terms $\Lambda^0(C_A)$ are interpreted as elements in a λ–model $E$. In this section, instead of dealing with λ–models themselves, we will first map *db*-terms to closed λ–terms $\Lambda^0(C_A)$. Then the meanings of the closed lambda terms under a λ–model are the meanings of the *db*-terms.

To be more confident in mapping λ–terms from *db-nfs*, we start from some standard combinators of the pure λ–calculus (Chapter 6 of 4), from which other λ–terms can be constructed.

**5.2 Definition** (some standard combinators)
1. *Truth* values: $\boldsymbol{T} \equiv \lambda\, xy.x$, $\boldsymbol{F} \equiv \lambda\, xy.y$,
2. The *identity* function: $\boldsymbol{I} \equiv \lambda\, x.x$.
3. *Pairing*: $[M, N] \equiv \lambda\, x.x\, M N$.
4. *Numerals*: $\lceil 0 \rceil \equiv \boldsymbol{I}$, $\lceil n+1 \rceil \equiv [\boldsymbol{F}, \lceil n \rceil]$.
5. *Predecessor*, and the test for *zero* of *numerals*:
   $\boldsymbol{P} \equiv \lambda x.x\boldsymbol{F}$; $\boldsymbol{Zero} \equiv \lambda\, x.x\boldsymbol{T}$
6. *Fixed Point Combinator*: $\boldsymbol{Y} \equiv \lambda f.\ (\lambda x.\ f(xx))\ (\lambda x.f(xx))$
7. $\Omega = (\lambda x.xx)\ (\lambda x.xx)$, an unsolvable term.

**5.3 Definition** 1. To identify the syntactical forms of lambda terms,
   $\# : \Lambda(C_A) \to \mathbf{N}$ (Numerals)
was used as an effective one-one mapping function, that is, $\# M$ is called the *Godel* number of $M$ for all $M \in \Lambda(C_A)$ (Definition 6.5.6 of 4).
2. Similarly, there is an effective one-one map between *db*-terms and the numerals:
   $\# : \mathcal{J} \to \mathbf{N}$(Numerals).
3. Let *id* is a special constant symbol in $\mathcal{C}$, and the corresponding lambda term is represented as $id^\lambda$.

To compare a λ–term with another, we introduce the λ–term $\boldsymbol{EQ}$:

**5.4 Definition** $\boldsymbol{EQ} \equiv \boldsymbol{Y}\, \boldsymbol{C}$, where

$$\boldsymbol{C} \equiv \lambda f.\ \lambda xy.\ (\boldsymbol{Zero}\ \#x)\ ((\boldsymbol{Zero}\ \#y)\ \boldsymbol{T}\ \boldsymbol{F})\ ((\boldsymbol{Zero}\ \#y)\ \boldsymbol{F}\ f(\boldsymbol{P}\ \#x)\ (\boldsymbol{P}\ \#y))$$

Then we can map all the *db*-terms in $\mathcal{J}$ into closed lambda terms $\Lambda^0(C_A)$. First, we assume that each constant in $\mathcal{J}$ is mapped to a $\lambda$–term, the $\lambda$–term of a *db*-term with tag is the one of the tag, and a proposition letter not in $\mathcal{D}$ is mapped to $\Omega$. Then the rest of the *db*-terms have derived mappings.

**5.5 Definition** Given a database $\mathcal{D}$, define a map $\lambda$: $\mathcal{J} \rightarrow \Lambda^0(C_A)$, as follows, where $M^\lambda$ is written for $\lambda(M)$.

1. If $M \equiv c$, where $c \in \mathcal{C}$, then $c^\lambda \in C_A$, or $c^\lambda \in \Lambda^0(C_A)$, where a special case: $\perp^\lambda = \Omega$.
2. If $M \in \mathcal{D}$, and $M$ has a tag, then $M^\lambda = M.tag^\lambda$
3. If $M \in \mathcal{D}$, and $M$ is a non-leaf *db*-term, then find all $M\,N_i$, where $i = 0, \ldots, n$ for an integer $n \geq 0$, such that $M\,N_i$ is in $\mathcal{D}$. By induction, we assume that $N_i^\lambda$ and $(M\,N_i)^\lambda$ have been defined. Then

   $M^\lambda = \boldsymbol{Y}\,\boldsymbol{C}$, where

   $\boldsymbol{C} \equiv \lambda f,\lambda x.\ ((\boldsymbol{EQ}\ x\ id^\lambda)\ \#M\ ((\boldsymbol{EQ}\ x\ N_1^\lambda)\ ((M\ N_1)/M)\ldots((\boldsymbol{EQ}\ x\ N_n^\lambda)\ ((M\ N_n)/M)\ \Omega)\ldots)$,

   where $(M\,N_i)/M \equiv f$ if $M\,N_i =_{EP} M$, or $(M\,N_i)/M \equiv (M\,N_i)^\lambda$ otherwise.
4. If $M$ is a proposition letter $a$, and $a \notin \mathcal{D}$, then $a^\lambda = \Omega$.
5. If $M\,N \notin \mathcal{D}$, then $(M\,N)^\lambda = M^\lambda\,N^\lambda$.

**5.6 Definition** $\forall M \in \mathcal{J},\ [\![M]\!]^E = [\![M^\lambda]\!]^E$. Or simply denote $[\![M]\!] = [\![M^\lambda]\!]$

In order words, the meaning of $M$ under $E$ is exactly the meaning of $M^\lambda$ under $E$. To better understand the definition 5.5.3, let's see the extensional definition of the function $[\![M^\lambda]\!]$. Given a non-leaf *db*-term $M$ in $\mathcal{D}$, find all $M\,N_i$, where $i = 0, \ldots, n$ for an integer $n$, such that each $M\,N_i$ is in $\mathcal{D}$. By induction, we assume that $[\![N_i^\lambda]\!]$ and $[\![(M\,N_i)^\lambda]\!]$ have been defined. Then

$$[\![M^\lambda]\!] = \{<[\![id^\lambda]\!>, [\![\#M]\!]>,\ <[\![N_1^\lambda]\!], [\![M\,N_1{}^\lambda]\!]>\ldots<[\![N_n^\lambda]\!], [\![M\,N_n]\!]>\}$$

The lambda expression in the definition 5.5.3 was given with the fixpoint combinator $\boldsymbol{Y}$ because it is possible that $M$ is equal ($=_{EP}$) to one or more $M\,N_i$. Allowing an application of a function equal to itself may not be interesting in practice, but it is valid theoretically. We will give a formal notion on this in Section 8.

For each non-leaf *db*-term $M \in \mathcal{D}$, the corresponding $\lambda$–term $M^\lambda$ is defined to have an application $M^\lambda\ id^\lambda$ equal to $\#\,M$. This semantically enforces that $M$ is unique in $\mathcal{D}$, and different from each other, even though there is no $N_i$ such that $M\,N_i \in \mathcal{D}$. The analogs of non-leaf *db*-terms are variables in programming languages. No matter the variables are initialized or not, they are different data containers for potential data.

We further show that the property of the applicative behavior of a $\lambda$–model is preserved in the EP data model.

**5.7 Lemma.** $\forall\,(M\,N) \in \mathcal{J},\ [\![M\,N]\!] = [\![M]\!]\ [\![N]\!]$.

*Proof.* 1. We show that $\forall M\,N \in \mathcal{J}, (M\,N)^\lambda = M^\lambda\,N^\lambda$. It is obvious from the definition 5.5.3 and 5.5.5.

2. $(M\,N)^\lambda = M^\lambda\,N^\lambda \Rightarrow [\![M\,N]\!] = [\![M]\!]\ [\![N]\!]$.

| | | |
|---|---|---|
| $[\![M\,N]\!]$ | $= [\![(MN)^\lambda]\!]$ | by the definition 5.6 |
| | $= [\![M^\lambda\,N^\lambda]\!]$ | by the first conclusion of this proof. |
| | $= [\![M^\lambda]\!]\ [\![N^\lambda]\!]$ | by the applicative behavior of $\lambda$–models (Chapter 18 of 4) |
| | $= [\![M]\!]\ [\![N]\!]$ | by the definition 5.6 |

Recall that the definition 4.3 imposed additional constraints on EP databases. This is important in supporting the definition 5.5. The definition 5.5.2 would never touch the ground of λ–terms without the restriction 4.3.1; and the definition 5.5.3 could give a conflict meaning for a non-leaf *db*-term without the restriction 4.3.2.

## 6 Reduction

The computing rules defined in 4.1 are not sufficient to convert arbitrary *db*-terms to *db-nfs*. Since the set of the constants $\mathcal{C}$ is a part of the EP data model, we must first give the computing behavior of the constants before the further inference rules for the *db*-terms can be given. For example, for any $c$ in $\mathcal{C}$ and any *db*-term $M$, what does $c\ M$ come up with? The *db*-term $c\ M$ is not a *db-nf*, then it, if it is decidable, should be computed to a normal form in a language governing the behavior of the constants $\mathcal{C}$.

The constants $\mathcal{C}$ and their applicative behaviors are governed by other languages other than the EP data model. They are needed because atomic values like integers, higher-order functions with infinite data like +, and certain queries against EP databases must be supported by a Turing-machine equivalent language. The EP data model is eventually to be said Turing-machine equivalent because the computing behavior of the constants is counted as axioms of the EP data model.

In the discussion of the reduction of the EP data model, We could use an extended λ–calculus sugared by *db*-terms to be the language governing the constants $\mathcal{C}$ and their computing behavior, as further discussed in Section 11. But we only request a legitimate computing behavior of the language, instead of imposing more specific syntactical definition and inference rules on the language. This will allow a heterogeneous distributed computing environment supporting multiple computing systems, instead of a single dictating system like a lambda calculus, a functional programming language, or a imperative language.

Firstly, we iterate the concept of applicative structures in (Definition 5.1.1 of [4]).

**6.1 Definition** $\mathcal{M} = (X, \cdot)$ is an applicative structure if $\cdot$ is a binary operations on $X$.

When $\mathcal{M}$ is viewed as, or supported by a language, then $\forall x, y \in X$, $\exists\ z \in X$, $\mathcal{M}$ will have a terminating procedure which computes $z$ from $x\ .\ y$ if $x\ .\ y$ is computable, or $\mathcal{M}$ will have no conclusion about $x\ .\ y$. The notion of applicative structures is exactly a part of what we need. We don't care how the result $z$ is computed, but what is the result $z$, which gives the legitimate computing behavior of a language.

**6.2 Examples** 1. Let $X \equiv \{0, 1, \perp\}$. For any objects $c_1, c_2 \in X$, a $\mathcal{M}$ axomitizes $c_1 \cdot c_2$ to be $\perp$.
2. Let $X \equiv \{Integers, Strings, Characters, \perp\}$. For arbitrary objects $c_1, c_2 \in X$, a $\mathcal{M}$ axomitizes $c_1 \cdot c_2$ to be $\perp$.
3. Let $X \equiv \{Integers, Plus, \perp\}$, a common sense language $\mathcal{M}$ will have: $Plus\ n =_{\mathcal{M}} n + 1$ for all $n$ in integers, and $\perp\ c =_{\mathcal{M}} \perp$; $c\ Plus =_{\mathcal{M}} \perp$ for all $c$ in $X$. Here $=_{\mathcal{M}}$ denotes equality formula in the language $\mathcal{M}$.
4. Let $X \equiv \Lambda^0(C_A)$, and $\forall x \in X$, $x$ is either a normal form, a head normal form, or an unsolvable term which has a distinct interpretation in $E$. Then the lambda calculus with $\Lambda^0(C_A)$ as lambda terms is the language such that $\forall\ c_1, c_2 \in \Lambda^0(C_A)$, $\exists\ c_3 \in \Lambda^0(C_A)$, the computing $c_1 \cdot c_2$ may terminate with the result of $c_3$, but it is not guaranteed.
5. Let $X \equiv \mathcal{C}$, then there is a language $\mathcal{M}$: $\forall\ c_1, c_2 \in \mathcal{C}$, $\exists\ c_3 \in \mathcal{C}$ such that $c_3$ can be computed if $c_1\ .\ c_2$ is computable.

The definition 6.1 cannot be used to completely explore the computing behavior of the constants $\mathcal{C}$, because the computing behavior of $c\ M$ was not covered if $M$ is a *db*-term in a EP database. The issue is

that the db-terms are different from the constants in terms of syntactical forms although both of them are interpreted as functions. The EP data model adjust itself to accept constants $\mathcal{C}$ and their legitimate computing behavior. The system or the language which supports the constants $\mathcal{C}$ and their computing behavior is also required to accept db-terms as its own constants. Therefore, we modify the notion of the applicative structures a little and impose the applicative behavior of $\mathcal{C}$ as the following.

**6.3 Definition.** 1. A system $\mathcal{M} = (X, \cdot, Y)$, where $Y \subseteq X$, is an semi-applicative structure if

$\forall c \in Y, \forall M \in X, \exists N \in X$, such that $c \cdot M = {}_{\mathcal{M}} N$.

2. In a semi-applicative structure $\mathcal{M}$, $c \cdot M$, called a $\mathcal{M}$-redex , is computable if $\mathcal{M}$ has an effective reduction such that $c\ M$ can be reduced to $N$, denoted as

$c\ M \rightarrow_{\mathcal{M}} N$.

$\mathcal{M}$ is recursive if $\forall c \in \mathcal{C}, \forall M \in DB\text{-}NF$, $c\ M$ is computable.

3. $\mathcal{M}$ is non-trivial if $\mathcal{M}$ has at least two distinguished elements

*Notation.* 1. $a \cdot b$ is usually written as $a\ b$. The elements in a (semi-)applicative structure are curried functions.

Now we can see that the definition 6.3 satisfies us in the regard of the computing behavior of the constants $\mathcal{C}$ We take the computing behaviors as the axioms of the EP data model.

**6.4 Axiom** Let $\mathcal{M} = (DB\text{-}NF, \cdot, \mathcal{C})$ is a non-trivial semi-applicative structure, and let $c \in \mathcal{C}, M, N \in DB\text{-}NF$.

$c\ M \rightarrow_{\mathcal{M}} N \Rightarrow c\ M =_{EP} N$

Just corresponding to the interpretation of *db*-terms in Definition 5.5, we give the following inference rules.

**6.5 Rules of inference**:

1. $\forall a \notin \mathcal{D} \Rightarrow a = \perp$.
2. Let $M\ L \in \mathcal{T}$, and $M \in \mathcal{D}$. Find all the $N_i \in \mathcal{D}$, here $i \geq 0$, such that $M\ N_i \in \mathcal{D}$. If there is no $N_i$ such that $N_i = L$, then $M\ L = \perp$.
3. $N_1 = N_2 \Rightarrow M\ N_1 = M\ N_2$.
4. $N_1 = N_2 \Rightarrow N_1\ M = N_2\ M$.

All the proposition letters not in the database are equal to $\perp$ by the rule 6.5.1. By the rule 6.5.2, all the combinations of two *db*-terms $M\ L$ are equal to $\perp$ if $M$ is in the database, but $L$ can not be converted to one of $N_i$, for all the $i \geq 0$ such that $M\ N_i$ is in the database. The rules 6.5.3 and 6.5.4 reflect the applicative behavior of $\lambda$–models.

**Examples.**

1. *College CS CS100 John Grade = 'A'*, by 4.1.1.
2. *C SQ Root 2* $= \boldsymbol{I}\ 2$, by 4.1.1
   $= 2$ by 4.1.1
3. *Plane Obj1 Shape 5 5* $= (\lambda x, y.\ 0 \leq x \leq 10 \wedge 0 \leq y \leq 10)\ 5\ 5$ by 4.1.1
   $= (\lambda y.\ 0 \leq y \leq 10)\ 5$ by 6.4
   $=$ true by 6.4

Similar to the reductions of other formal theories, let's give the definition of the reduction between *db*-terms.

**6.6 Definition** Let $M, N \in \mathcal{J}$ with a $\mathcal{C}$ and a $\mathcal{D}$. If there is a (finite) sequence $L_0, \ldots, L_n \in \mathcal{J}$, where $n \geq 0$, such that $M \equiv L_0$, $L_0 = L_1, \ldots, L_{n-1} = L_n$, $L_n \equiv N$, then

1. $M$ is (effectively) convertible (reducible, or computable) to $N$, written as $M \rightarrow_{EP} N$.
2. $L_i$, where $i = 0, \ldots,$ or $n$, is called an intermediate term of the reduction from $M$ to $N$.

Now with the inference rules given in 4.1 and 6.5, and the axioms given in 6.4, we would like to see that an arbitrary *db*-term can be reduced to a *db-nf* as long as the relevant $\mathcal{M}$-redexes are computable. To clarify the term "relevant $\mathcal{M}$-redexes", we have to introduce more tedious notions.

**6.7 Definition.** Let $M \in \mathcal{J}$ with a $\mathcal{C}$ and a $\mathcal{D}$.

1. If a sub term $\Delta \subset M$ is not in normal form by itself, $\Delta$ is called a redex.
2. In $M$ where each subterm is parenthesized, a subterm $\Delta_1$ is to left of a subterm $\Delta_2$ if the right-parenthesis ')' of $\Delta_1$ is left to the left-parenthesis '(' of $\Delta_2$.
3. A redex $\Delta$of $M$ is the left-most redex if there is no other redex $\Delta$'of $M$, such that $\Delta$' is to left of $\Delta$ in $M$. For example in Fig. 1, (*C SQ Root*) and (*I 2*) are the left-most redexes of ((*C SQ Root*) *2*) and (*I* (*I 2*)) respectively.
4. The reduction procedure of a *db*-term M is in left-most order if each intermediate term of the reduction is reduced by starting from the left-most redex.
5. The relevant $\mathcal{M}$-redexes of $M$ are the left-most $\mathcal{M}$-redexes of the intermediate terms that can be reduced starting from $M$ in the left-most reduction order.

**6.8 Theorem** $\forall M \in \mathcal{J}$ with a $\mathcal{C}$ and a $\mathcal{D}$, $M$ can be effectively reduced to one and only one *db-nf* $N$ if the relevant $\mathcal{M}$-redexes of M are computable.

*Proof.* Given a $M \in \mathcal{J}$,

1. If $M \in \mathcal{D}$,
   a). if $M$ has a tag, then the normal form of $M$ is the normal form of *M.tag*. Or
   b). if $M$ has no tag, then $M$ is a non-leaf *db*-term. Then $M$ itself is the normal form.
2. If $M \notin \mathcal{D}$,
   a). if $M \equiv a$, and $a \notin \mathcal{D}$, then $a \rightarrow_{EP} \perp$ by 6.5.1. Then $M$ has the unique normal form $\perp$.
   b). if $M \equiv c$, and $c \in \mathcal{C}$, then $c$ itself is the unique normal form.
   c). if $M \equiv N_1 N_2$, by induction, assume that $N_1$, $N_2$ are effectively reduced to two normal forms $N_1'$, $N_2'$ separately.
      i). If $N_1' \equiv c$, $c \in \mathcal{C}$, then $N_1' N_2' \rightarrow_{\mathcal{M}} N'$, here $N'$ is a normal form by 6.4. Then $M \rightarrow_{EP} N'$ by Axiom 6.4. The reduction on the $c$ $N_2'$, the left-most $\mathcal{M}$-redex of itself, is effective according to the condition of the theorem.
      ii). If $N_1'$ is a non-leaf *db*-term in $\mathcal{D}$, we can search all the $M L_i$, where $i \geq 0$, such that $N_1' L_i \in \mathcal{D}$. If there is a $L_i$ such that $L_i = N_2'$, then $N_1' L_i$ will inductively has a unique normal form according to the Proof 1. If there is no $Li$ such that $Li = N_2'$, then $M \rightarrow_{EP} \perp$ by 6.5.2. Since $i$ is a finite number, the entire reduction process is effective. Finding all $N_i' L_i$ is effective. Reducing all the $L_i$ to normal forms inductively is effective, then comparing $N_2'$ with each $L_i$ is effective. Therefore, the entire reduction process is effective.

Discussing the effectiveness of computing *db*-terms has satisfied us for the purpose of this paper. It is out of the scope to discuss the efficiency of computing *db*-terms by analyzing different reduction orders like applicative order and parallelism although the author see the potential of the EP data model in this area. The next issue is the consistence of the EP data model as a language. We will borrow the notion of consistence from (Definition 2.1.30 of [4]).

**6.9 Definition.** A formal theory with equations as formulas is consistent if the formal theory doesn't prove every equation.

*Notation.* Definition 6.6 can be re-written as $M \neq N$ if there is no reduction sequence such that $M \rightarrow_{EP} N$.

**6.10 Theorem** (consistence). 1. Let $M$, $N$ are two distinct (not identical) normal forms, then $M \neq N$.
2. The EP data model is consistent.

*Proof.* 1. If $M$, $N$ are distinct (syntacticaly different), and $M = N$, then $M$ would have two normal forms, itself and $N$. However, there is no any inference rule in the EP data model which reduce one normal form $M$ to another distinct normal form $N$. $M = N$ is contradict to the EP data model theory, and it must be true that $M \neq N$.
2. If $\mathcal{T}$ was inconsistent, then $M = N$ for all $M$, $N$. But the condition given in Axiom 6.4 required that the semi-applicative structure $\mathcal{M}$ is non-trivial, that is, there are at least two distinguished constants $c_1$ and $c_2$ in M. Since $c_1$ can not be $\mathcal{M}$ -reduced to $c_2$, $c_1$ and $c_2$ are the two distinguished normal forms in the EP data model. It is a contradiction that $\mathcal{T}$ is inconsistent.

**6.11 Theorem.** $M \rightarrow_{EP} N \Rightarrow [\![M]\!] = [\![N]\!]$ (soundness)

*Proof.* 4.1.1 is true by the definition 5.5.2.
4.1.2 is true by itself.
6.4 is true by itself as an axiom.
6.5.1is true by 5.5.1.

| | | |
|---|---|---|
| 6.5.2: $[\![M\,N]\!]$ | $= [\![M]\!]\,[\![N]\!]$ | by Lemma 5.7 |
| | $= [\![\Omega]\!]$ | by Definition 5.5.3 and the condition of 6.5.2 |
| | $= \perp_E$ | By the semantics of $\Omega$. |
| 6.5.3: $[\![M\,N_1]\!]$ | $= [\![M]\!]\,[\![N_1]\!]$ | by Lemma 5.7 |
| | $= [\![M]\!]\,[\![N_2]\!]$ | by induction on the structure of db-terms with the assumption that $N_1 = N_1 \rightarrow [\![N_1]\!] = [\![N_2]\!]$ |
| | $= [\![M\,N_2]\!]$ | by Lemma 5.7 |

6.5.4: Similar to the proof for 6.5.3.

The last issue with regard to the EP data model as a language is completeness. That is, is it true that $[\![M]\!] = [\![N]\!] \Rightarrow M = N$? Or $M \neq N \Rightarrow [\![M]\!] \neq [\![N]\!]$? The answer is no in general simply because of the "weak" reduction systems of a Turing-machine equivalent formal language like $\lambda$–calculi. For example in the pure $\lambda$–calculus, for any normal form $N \in \Lambda$, there is a $\lambda$–term $X$ with no normal form such that $[\![M]\!]^{D_\infty} = [\![X]\!]^{D_\infty}$, here $D_\infty$ is the Scott' $\lambda$–model [37], [36]. This will be true for a $\lambda$–calculus sugared by the constants $C_A$, and further would be true for the EP data model when $\mathcal{C} \subseteq \Lambda^0(C_A)$. Another possibility of the incompleteness in the EP data model is when $[\![c]\!] = [\![M]\!]$, where $c \in \mathcal{C}$ and $M$ is a non-leaf *db*-term in a database $\mathcal{D}$. In this case, however, $c \neq M$ according to the inference rules of the EP data model.

Fortunately, it is not the practical concern to construct two expressions for a single function. Therefore, the following conclusion should satisfy the computing practice.

**6.12 Corollary** Let $M, N \in DB\text{-}NF$, $[\![M]\!] \neq [\![N]\!]$ if
1. for any two distinguished constants $c_1, c_2 \in \mathcal{C}$, $[\![c_1]\!] \neq [\![c_2]\!]$.
2. for any non-leaf *db*-term $Q$ in $\mathcal{D}$, and any constant $c \in \mathcal{C}$, $[\![Q]\!] \neq [\![c]\!]$.

The condition 1 requests that each constant uniquely represents a function; and the condition 2 requests that a non-leaf *db*-terms is constructing a new function on the base of the constants $\mathcal{C}$

*Proof.* Case 1. If $M$ and $N$ are two constants, $[\![M]\!] \neq [\![N]\!]$ by the condition 1.
Case 2. If $M$ and $N$ are two distinguished non-leaf *db*-terms, $[\![M]\!] \neq [\![N]\!]$ by the definition 5.5.3.
Case 3. If $M$ is a constant, and $N$ a non-leaf *db*-terms in $\mathcal{D}$, $[\![M]\!] \neq [\![N]\!]$ by the condition 2.

Now, let's summarize what is meant by the EP data model after a serial augmentations of notions. First of all, The EP data model adopts the constants $\mathcal{C}$ and their computing behavior, represented by the semi-applicative structure (*DB-NF*, ·, $\mathcal{C}$ ). $\mathcal{C}$ could virtually reference the entire set of the effectively computable functions. This is why the EP data model will be shown Turing-machine equivalent. But it would be not real unless they have been constructed by other languages. Therefore, another view of the EP data model would be an extended λ–calculus with the flavor of the EP data model, as we will discuss more in Section 11. How to construct constants is not the concern of the EP data model. What really makes the EP data model interesting is storing and manipulating the constants, and constructing higher-order functions on the top of the constants. Secondly, the EP data model can be characterized as another semi-applicative structure ($\mathcal{J}$, ·, $\mathcal{P}$) by ignoring the constants. Combining the two portions ($\mathcal{J}$, ·, $\mathcal{P}$) and (*DB-NF*, ·, $\mathcal{C}$) together, the EP data model is an applicative structure.

**6.13 Corollary** The EP data model ($\mathcal{J}$, · ) is an applicative structure. That is, $\forall M, N \in \mathcal{J}, \exists L \in \mathcal{J}$ such that $MN = L$.

*Proof.* 1. If all the sub terms of $MN$ are computable, $MN$ has a unique normal form, which must be $L$ by 6.7. In other words, $MN \rightarrow_{EP} L \Rightarrow MN = L$.
2. If some relevant $\mathcal{M}$-redexes $cQ$ of $MN$ are not computable. But according to the definition 6.3.1, $cQ =_{\mathcal{M}} Q'$, where $Q'$ is a *db-nf*, the sub term $cQ$ in $MN$ can be replaced by $Q'$. Inductively, $MN$ should be equal to a *db-nf*. We may not have EP reductions for this reasoning, and we may never know which db-nf M N is equal to, but $MN$ do have a *db-nf*.

Before finishing this section, we would like to highlight a obvious, but signification conclusion: all the effectively computable functions can be managed by the EP data model as long as they can be finitely expressed by a Turing machine equivalent language. Similar to the concept of λ-definability, let's give the definition of *EP*-definability. In the following, we assume that the numerals $\lceil n_i \rceil$ in λ–terms are the constants in $\mathcal{C}$

**6.14 Definition** Let φ be a numeric function with $p$ arguments. φ is called *EP*-definable if there is a *db*-term $M$ in a EP database $\mathcal{D}$ such that

$$\forall \lceil n_1 \rceil, \ldots, \lceil n_p \rceil \in \mathbf{N},\ M\lceil n_1 \rceil \ldots \lceil n_p \rceil \rightarrow_{EP} \lceil \varphi(n_1, \ldots, n_p) \rceil$$

**6.15 Theorem** A partially recursive numeric function φ is *EP*-definable.

*Proof.* 1. The partially recursive numeric function φ is λ-definable according to Kleene Theorem (8.4.13 of 4. In other words, there is a λ–term $F$, such that

$$\forall \lceil n_1 \rceil, \ldots, \lceil n_p \rceil \in \mathbf{N},\ F\lceil n_1 \rceil \ldots \lceil n_p \rceil \rightarrow_{\lambda} \lceil \varphi(n_1, \ldots, n_p) \rceil$$

2. Let $M$ is a *db*-term in a database $\mathcal{D}$, and assign $M.tag \equiv F$.
Then we have $M\lceil n_1 \rceil \ldots \lceil n_p \rceil = (M.tag)\lceil n_1 \rceil \ldots \lceil n_p \rceil = F\lceil n_1 \rceil \ldots \lceil n_p \rceil = \lceil \varphi(n_1, \ldots, n_p) \rceil$
Therefore, φ is *EP*-definable.

## 7 Active Domain and Recursive Enumeration

It could be the end of discussing the EP data model as a language after we have finished its syntactical and semantic definitions. However, The EP data model is introduced for database applications, where

queries and updating operations have to be constructed against databases. Without exceptions, these queries and updating operations will be viewed as higher-order functions, and they are expected to be Turing-machine equivalent. The immediate candidate of descriptive query languages is an extended lambda calculus, in which the *db*-terms in the EP data model are possibly λ-terms $\Lambda(\mathcal{J})$. On the other hand, a first-order language will be very useful in providing "ad-hoc" queries against EP databases. In stead of developing these two languages, we will try to discover a few ordering relations in the rest of the paper which will act as primary database manipulating operators, and as constants in the extended lambda calculus and the first order language. To facilitate the coming sections, we explore the properties of the *db*-terms as sets in this section, and introduce the notions of active domain and recursive enumeration.

We know from the previous sections that a EP database $\mathcal{D}$ defines nothing but a set of higher-order functions. While a constant function gives the definition of the function in intension, a *db*-term in $\mathcal{D}$ gives the definition of a function in extension. Regardless the ways of the function definitions, a function conceptually defines a set of argument/value pairs (or called function enumeration); and recursively each argument, as a function again, defines another set of argument/value pairs. When we recursively union the argument/value pairs together, where the values are not undefined, or meaningless, the union would be the complete, discrete properties of the given function. This union has the term "recursive enumeration" for its syntactical presentation, and the term "active domain" for the semantic presentation.

In the definition 5.1, we chose the λ–model $E \approx A + [E \to E]$ as the function space of the EP data model. The intention was to have a set of primary constants $A$ in the function space such that applying each primary constant to arbitrary element in the space would be equal to the least defined element $\perp_E$. This has been reflected for the λ–model $E$ and the corresponding lambda calculus $\Lambda(C_A)$ in Definition 5.1.3. Correspondingly, it is imposed on the EP data model.

**7.1 Definition** Let $\mathcal{C}_A \subseteq \mathcal{C}$. $\forall c \in \mathcal{C}_A, \forall M \in \mathcal{J}, c\, M = \perp$.

The notions of recursive enumeration or active domain practically make sense only under the special applicative behavior of the primary constants $\mathcal{C}_A$ defined above. Let's first introduce more abbreviations for certain syntactical forms of db-terms.

*Notation*

1. Let $\bar{N} \equiv \underline{N}N_1, ..., N_n \in \mathcal{J}$. Then $M\, N_1 ... N_n \equiv M\ \bar{N} \equiv (...((M\, N_1)\, N_2) \ ...\ N_n)$.
2. Let $\tilde{N} \equiv \underline{N}N_1, ..., N_n \in \mathcal{J}$. Then $\tilde{N}\, M \equiv (N_1\, (N_2, \ ..., \ (N_n\, M)...)$.
3. $\|M\|$ is the length of $M$ in symbols, here $M \in \mathcal{J}$.

**7.2 Definition** 1. Given $M \in \mathcal{D}$, the *RE* (Recursive Enumeration) of $M$, denoted as *RE* ($M$), is the set of all the *db*-terms:

$M\ \bar{N}$, where $\bar{N} \in DB\text{-}NF(\mathcal{D})$, $\|\ \bar{N}\ \| \geq 0$, such that $M\ \bar{N} \neq \perp$.

2. $\forall X \subseteq \mathcal{J}$, $RE(X) = \{RE(M) \mid M \in X\}$.

**7.3 Examples** 1. In Fig. 2, *RE(College Admin John)* = { *College Admin John, College Admin John Reg#, College Admin John Enroll*}.

2. In Fig.1, $RE(\mathcal{D})$ = {*Root, Root 4, Root 9, I, I 2, I 3, SQ 2, SQ 3, C, C SQ, C SQ Root, C SQ Root 2, C SQ Root 3, C SQ SQ, C SQ SQ 2, C SQ SQ 3*}

3. In Fig.3, $RE(\mathcal{D})$ = {*Plane, Plane Obj1, Plane Obj1 Color, Plane Obj1 Shape , Plane Obj1 Shape 0, ..., Plane Obj1 Shape 0 0, ..., Plane Obj2, Plane Obj2 Color, Plane Obj2 Shape , Plane Obj2 Shape 0, ..., Plane Obj2 Shape 0 0, ...,* }.

4. If $M = \perp$, then $RE(M) = \phi$.

**7.4 Definition** 1. Let $M \in \mathcal{J}$, $adom\ (M) = \{\mathbb{[}x\mathbb{]} \mid x \in RE(M)\}$
2. Let $X \subseteq \mathcal{J}$, $adom\ (X) = \{\mathbb{[}x\mathbb{]} \mid x \in RE(X)\}$.

Similar to the active domains in relational databases [1], the active domain of a set $X \subseteq \mathcal{J}$ gives the complete elements (except for $\perp_E$) in $E$ which are represented by the *db*-terms $M \in X$. Two *db*-terms in $RE(X)$ may collapsed into a single element in $adom(X)$ by the function *adom* if the two elements are convertible.

Theorem 6.8 indicated that all the *db*-terms in $\mathcal{J}$ have at most one convertible normal forms. A normal form represents a unique element in $E$. Therefore if all the constants $\mathcal{C}$ are derivable from $\mathcal{D}$, that is, $adom(\mathcal{C}) \subseteq adom(\mathcal{D})$, then $\mathcal{J}$ does not carry more meaning than what $\mathcal{D}$ does; and $\mathcal{D}$ defines the exactly same meaning as the set *DB-NF* does.

**7.5 Proposition** $adom(\mathcal{J}) = adom\ (\mathcal{D}) = adom(DB\text{-}NF)$ if $adom\ (\mathcal{C}) \subseteq adom\ (D)$

*Proof.* 1. $\forall M \in \mathcal{J}, \exists N \in DB\text{-}NF,\ M =_{EP} N$ by Corollary 6.12. Then $\mathbb{[}M\mathbb{]} = \mathbb{[}N\mathbb{]}$. Therefore $adom\ (\mathcal{J}) \subseteq adom\ (DB\text{-}NF)$.
2. Since $DB\text{-}NF \subseteq T$, then $adom\ (DB\text{-}NF) \subseteq adom\ (\mathcal{J})$. By 1 and2, $adom\ (\mathcal{J}) = adom\ (DB\text{-}NF)$.
3. Since $\mathcal{D} \subset \mathcal{J}$, then $adom\ (\mathcal{D}) \subseteq adom\ (DB\text{-}NF)$ by Proof 1.
4. $DB\text{-}NF = \mathcal{C} \cup$ the set of the non-leaf db-terms in $\mathcal{D} \subseteq \mathcal{C} \cup \mathcal{D}$. Then $adom\ (DB\text{-}NF) \subseteq adom\ (\mathcal{C}) \cup adom\ (\mathcal{D})$.
5. Since $adom\ (\mathcal{C}) \subseteq adom\ (\mathcal{D})$, then $adom\ (DB\text{-}NF) \subseteq adom\ (\mathcal{D})$. Then $adom\ (\mathcal{D}) = adom\ (DB\text{-}NF)$ by 3, 5.

Since the constants $\mathcal{C}$ are independent from EP databases $\mathcal{D}$, it is possible that some constants are not managed by the EP databases. This could be real in practice, but not interesting from the view point of the EP data model. The proposition 7.5 says that EP databases is the only concern of the EP data model if either all the pre-constructed finite functions are managed by the EP databases, or the EP data model are only interested in managing some pre-constructed functions. In other words, the constants $\mathcal{C}$ should be always derivable from EP databases $\mathcal{D}$ in practice.

Before moving on to ordering relations of the EP data model, we would like to explore more properties of recursive enumeration and active domains. The notion "recursive" is used in the common sense that a set $X$ is said to be recursive if "$x \in X$" is effectively decidable.

**7.6 Proposition**. Let $X \subseteq \mathcal{J}$, and $\mathcal{D}$ an EP database.
1. If $RE(\mathcal{C})$ is recursive, so is $RE\ (X)$.
2. If $\mathcal{C}$ is finite, so are *DB-NF* and $adom\ (\mathcal{J})$.

*Proof.* 1. Given a term $M \in \mathcal{J}$, and a $X \subseteq \mathcal{J}$, the question $M \in RE\ (X)$ is decidable. Search the elements in $X$ to see if there is a $L \in X$ such that $L\ \bar{N} \equiv M$ for a $\bar{N} \in \mathcal{J}$. If no, then $M \notin RE\ (X)$. Otherwise, reduce $M$ to a normal form. Since $RE(\mathcal{C})$ is recursive, then the relevant $\mathcal{M}$-redexes of $M$ are computable. By the theorem 6.8, reducing $M$ to a normal form is effective. If the normal form of $M$ is $\perp$, then $M \notin RE\ (X)$. Otherwise, $N \in RE\ (X)$.
2. By the definition 4.4.3, *DB-NF* is the union of $\mathcal{C}$ and the non-leaf *db*-terms in $\mathcal{D}$. Since the non-leaf db-terms are finite, and $\mathcal{C}$ is finite. So is *DB-NF*.

These propositions say that functions constructed by a $\mathcal{D}$ are always recursive or finite as long as the properties of the constants in $\mathcal{D}$ are recursive or finite respectively. Can we say that $RE\ (X)$ is finite if $RE\ (C)$ is finite? The answer is no in general. We will come back to this point in the next section once we defined more notions.

## 8 Ordering Relations

Different from the relational data model or other object-oriented models, the EP data model has its data organized with the interpretation of higher-order functions, and the relationships among the data obey the computing (applicative) behavior of the functions. In this section and the coming 2 sections, we are going to discover these relationships from the EP data model, and represent them in a few ordering relations. As built-in operators, the ordering relations are expected to be highly descriptive and expressive in the construction of queries, and updates against EP databases. The followings are some examples of the queries which can be supported by the ordering relations:

### 8.1 Query Example

1. "Print out all the students who registered in the College in Fig 2".
2. "Find all the coordinators <$x$, $y$>, where $x$, $y$ are integers, and they are both in Obj1 and Obj2"
3. "Delete all the data which is relevant to *John* from *College* in the database in Fig 2."
4. "Given a directed graph $G$, is there a path from vertex $A$ to vertex $B$? And is there a circle in the directed graph?"

In this section, we formalize the relationships among functions, arguments, and values. A value depends on its function and the corresponding argument. In other words, given a function, an argument, and the corresponding value, the value is partly dependent on the function, and partly on the argument.

**8.2 Definition**. 1. Given $M, N \in \mathcal{J}$, $M$ is called the function of $N$, and $N$ the value of $M$, denoted as $N <_{v\text{-}f} M$, if there is a *db*-term $L$ in $\mathcal{J}$ such that $M\,L = N$.
2. Given $M, N \in \mathcal{J}$, $M$ is called the argument of $N$, and $N$ the value of $M$, denoted as $N <_{v\text{-}a} M$, if there is a *db*-term $L$ in $\mathcal{J}$ such that $L\,M = N$.
3. Given $M$ and $N$, $M$ is called the ancestral function of $N$, denoted as $N \leq_{v\text{-}f} M$, if there is a $\bar{N}$, where $\| \bar{N} \| \geq 0$, such that $M\ \bar{N} = N$.
4. Given $M, N \in \mathcal{J}$, $M$ is called the ancestral argument of $N$, denoted as $N \leq_{v\text{-}a} M$, if there is $\tilde{N} \in \mathcal{J}$, where $\|\tilde{N}\| \geq 0$, such that $\tilde{N}\,M = N$.

**8.3 Example** By applying the defined ordering relations, the following boolean expressions can be given and each of them is true:

1. $C\ SQ\ Root <_{v\text{-}f} C\ SQ$
2. $I <_{v\text{-}f} C\ SQ$
3. $3 <_{v\text{-}f} Root$
4. $C\ SQ <_{v\text{-}a} SQ$
5. $College\ CS\ CS100\ John <_{v\text{-}a} SSD\ John$

**8.4 Example** When the relations $<_{v\text{-}f}$ and $<_{v\text{-}a}$ are used in an extended first-order language, some queries in Example 8.1 would have the following expressions.

1. For Query 8.1.1: $\{ x \mid x <_{v\text{-}f} College\ Admin \wedge x \neq \perp \}$.
2. For Query 8.1.2: $\{<x, y> \mid x, y \in Integers \wedge Plane\ Obj1\ Shape\ x\ y = true \wedge Plane\ Obj2\ Shape\ x\ y = true\}$

**8.5 Definition** 1. A relation $\rho$ in a set $X$ is reflexive iff $x \rho\ x$ for each $x$ in $X$. $\rho$ is symmetric if $x \rho\ y$ implies $y\ \rho\ x$, and it is transitive iff $x \rho\ y$ and $y\ \rho\ z$ imply $x \rho\ z$.
2. A relation $\rho$ in $X$ is an equivalence relation (=) iff $\rho$ is reflexive, symmetric, and transitive.
3. $\rho$ is antisymmetric iff whenever $x\ \rho\ y$ and $y\ \rho\ x$ imply $x = y$.

4. A relation $\rho$ is called partial ordering in $X$ iff $\rho$ is reflexive, antisymmetric, and transitive. And a relation $\rho$ is called pre-ordering in $X$ if $\rho$ is reflexive and transitive.

**8.6 Theorem** 1. $=_{EP}$ is a equivalent relation in $\mathcal{J}$.
2. the relation $\leq_{v\text{-}f}$, $\leq_{v\text{-}a}$ in $\mathcal{J}$ are pre-orderings (reflexive, and transitive).

*Proof.* 1. $=_{EP}$ is reflexive, symmetric, and transitive by its definition.
2. $\leq_{v\text{-}f}$, $\leq_{v\text{-}a}$ are reflexive in $\mathcal{J}$ by their definitions.
3. $\leq_{v\text{-}f}$ is transitive. If $M \leq_{v\text{-}f} L$ and $L \leq_{v\text{-}f} Q$, then there are $\bar{N}_1$ and $\bar{N}_2$ in $\mathcal{J}$, such that $M = L\ \bar{N}_1$, and $L = Q\ \bar{N}_2$. Then, $M = Q\ \bar{N}_2\ \bar{N}_1$. Therefore, $M \leq_{v\text{-}f} Q$.
4. $\leq_{v\text{-}a}$ is transitive. Similarly if $M \leq_{v\text{-}f} L$ and $L \leq_{v\text{-}f} Q$, then there are $\tilde{N}_1$ and $\tilde{N}_2$ in $\mathcal{J}$, such that $M = \tilde{N}_1 (L)$, and $L = \tilde{N}_2 Q$. Then $M = \tilde{N}_1 (\tilde{N}_2 (Q))$. Therefore, $M \leq_{v\text{-}f} Q$.

It can be shown that the pre-orderings of a set are preserved under sub sets. Therefore, $\leq_{v\text{-}f}$, $\leq_{v\text{-}a}$ are pre-orderings in $RE(\mathcal{J})$, $RE(\mathcal{D})$,and $\mathcal{D}$. In the rest of the paper, we will have our eyes on the set $RE(\mathcal{J})$ in analyzing ordering relations since analyzing the *db*-terms convertible to $\perp$ does not add values to our conclusions.

The relations $\leq_{v\text{-}f}$, $\leq_{v\text{-}a}$, say $\rho$, are not generally partial orderings, or antisymmetric in $RE(\mathcal{J})$, because it is not generally true that $M \rho N$ and $N \rho M$ implies that $M = N$ for $M, N \in RE(\mathcal{J})$. See the section 9 for a directed graph as an example. However, if additional restrictions are imposed on $\mathcal{D}$, the relations $\leq_{v\text{-}f}$, or $\leq_{v\text{-}a}$ could be partial ordering in $RE(\mathcal{J})$.

**8.7 Definition** 1. Let two distinct $M, N \in RE(\mathcal{J})$. If $M = N$ and $M \leq_{v\text{-}f} N$, then $M$ is a *fun-loop* of N, denoted $M \approx_{v\text{-}f} N$.
2. Let two distinct $M, N \in RE(\mathcal{J})$. If $M = N$ and $M \leq_{v\text{-}a} N$, then $M$ is a *arg-loop* of $N$, denoted $M \approx_{v\text{-}a} N$.

It is a fun-loop (function loop) if applying a function to an argument is equal to the function itself; and it is an arg-loop (argument loop) if applying a function to an argument is equal to the argument itself. The notions of fun-loops and arg-loops is used between *db*-terms. And then they are also implied between functions.

**8.8 Corollary** 1. In $RE(\mathcal{J})$, if there are not two distinct $M$, $N$, such that $M \approx_{v\text{-}f} N$, then $RE(\mathcal{J})$ is partially ordered by $\leq_{v\text{-}f}$.
2. In $RE(\mathcal{J})$, if there is no two distinct $M$, $N$, such that $M \approx_{v\text{-}a} N$, then $RE(\mathcal{J})$ is partially ordered by $\leq_{v\text{-}a}$.

*Proof.* It is obvious from the Theorem 8.6 and the definition of partial ordering.

The examples given in Section 3 don't have *fun-loops* or *arg-loops*. Therefore the *db*-terms offered by each $\mathcal{D}$ in Section 3 are partially ordered by $\leq_{v\text{-}f}$, and $\leq_{v\text{-}a}$.

Section 7 has left a question: is $RE(X)$ finite if $RE(C_X)$ is finite?

**8.9 Proposition** Let $\mathcal{D}$ be a database.
1. $RE(\mathcal{J})$ is infinite if there is a *fun-loop*, or a *arg-loop* in $RE(\mathcal{J})$.
2. $RE(\mathcal{J})$ is finite if $RE(C_{\mathcal{D}})$ is finite and there is neither *fun-loops*, nor *arg-loops* in $\mathcal{D}$.

*Proof.* 1. If there is a *fun-loop* in $RE(\mathcal{J})$, then there are two distinct $M, L \in RE(\mathcal{J})$, and others $\bar{N} \in RE(\mathcal{J})$, such that $M \equiv L\ \bar{N}$. Then $L = L\ \bar{N}$. Then the infinite *db*-terms $L$, $L\ \bar{N}$, $L\ \bar{N}\ \bar{N}$, ... are in $RE(\mathcal{J})$. Similarly if there is a *arg-loop* in $RE(\mathcal{J})$, then there are two distinct $M, L \in RE(\mathcal{J})$, and others $\tilde{N} \in RE(\mathcal{J})$, such that $M \equiv \tilde{N}(L)$. Then $L = \tilde{N}(L)$. Then the infinite *db*-terms $L$, $\tilde{N}(L)$, $\tilde{N}(\tilde{N}(L))$, ... are in $RE(\mathcal{J})$

2. Since $RE(C_{\mathcal{D}})$ is finite, and $\mathcal{D}$ is finite, then there are a finite number of constant symbols and finite proposition letters in $\mathcal{D}$ and $RE(C_{\mathcal{D}})$. And *DB-NF* is finite by Theorem 7.6.2. If $RE(\mathcal{J})$ was infinite, it must be true that there is a *db*-term $M \in RE(\mathcal{J})$, such that $M \neq \bot$, and $\|M\| = \infty$. In other words, there are at least two sub terms of M such that both are in $RE(\mathcal{J})$, and are equal. The $M$ could be either $M \equiv L\ \bar{N}$ or $M \equiv \tilde{N}\ L$, where $L$, $\bar{N}$, and $\tilde{N}$ are arbitrary *db*-terms. It must be true in one of the following cases.

Case 1. $M \equiv L\ \bar{N}$. It happens that there are two distinct $\bar{N}_1$, and $\bar{N}_2$, such that $L\ \bar{N}_1 = L\ \bar{N}_2$, and $L\ \bar{N} \leq_{v\text{-}f} L\ \bar{N}_2 \leq_{v\text{-}f} L\ \bar{N}_1$. Then $L\ \bar{N}_1 \approx_{v\text{-}f} L\ \bar{N}_2$, which is contradict with the condition that there is no *fun-loops* in $RE(\mathcal{J})$.

Case 2. $M \equiv \tilde{N}\ L$. It similarly happens that there are two distinct $\tilde{N}_1$, and $\tilde{N}_2$, such that $\tilde{N}_1 L = \tilde{N}_2 L$, and $L\tilde{N} \leq_{v\text{-}f} L\tilde{N}_2 \leq_{v\text{-}f} L\tilde{N}_1$. Then $L\ \bar{N}_1 \approx_{v\text{-}a} L\ \bar{N}_2$, which is contradict with the condition that there is no *arg-loops* in $RE(\mathcal{J})$.

The above proposition says that fun-loops and arg-loops add infinite meaningful *db*-terms even though the semantics of the db-terms may be finite. Normally in database application practice, we shall avoid this situation. However, arg-loops may be natural to represent some special application data in the real world. In the coming section, we will give an example with arg-loops.

## 9 Database Design II.

Propositions 7.6.2 and 8.9 indicate that a database with fun-loop and arg-loop doesn't offer more elements in the active domain than those in the recursive enumeration. They ought to be avoided in representing partially ordered data. However, not every thing in the world is simply partially ordered. In this section, let's give one example to demonstrate how the data not partially ordered in the world can be represented by arg-loops, and how the queries like 8.1.4 can be expressed by the ordering relations.

**9.1 Example**. EP database presentations for directed graphs.

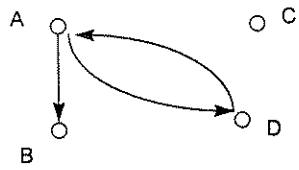


Fig.4 A Directed Graph

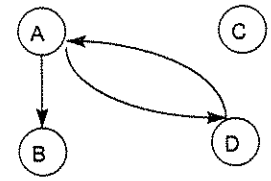


Fig.6 Althernative EP Database Presentation

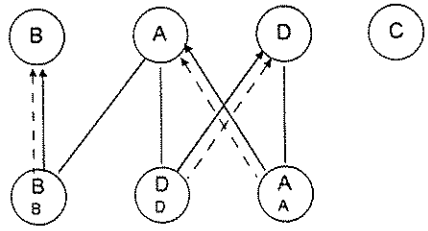


Fig.5 EP Database Presentation

A directed graph could be represented in the EP data model in many ways. The presentation in Fig. 5 for the directed graph in Fig. 4, however, is a "natural" one for directed graphs, and allows the queries in examples 8.1.4 to be expressed by ordering relations. In Fig. 5, each node represents a vertex in the graph of Fig. 4, and is named by the label given in Fig. 4. When a directed link connects a vertex (say $B$ ) from another vertex (say $A$), $A\ B$ in Fig. 5 will be an application of $A$ by applying to $B$, and $A\ B$ is assigned a tag identical to $B$. The textual presentation of the database in Fig. 5 is:

$$\mathcal{D} = \{A,\ A\ B\ (tag = B),\ A\ D\ (tag = D),\ D,\ D\ A\ (tag = A),\ B,\ C\}.$$

In this database, It is clear that $A\ B \approx_{v\text{-}a} B$; and $A\ D \approx_{v\text{-}a} D$; and $D\ A \approx_{v\text{-}a} A$. When the left hand side (say $AB$) of $\approx_{v\text{-}a}$ is collapsed with the right hand side (say $B$), and the corresponding up-down solid link is replaced by an solid arrow from the down node $AB$ to the up node $A$, the presentation in Fig. 6 can be reached, which looks like the same as the original graph in Fig. 4. Another sense of the "natural" is that no matter which node one starts with and which arrow one traces, the sequence of the node names along a tracing path in Fig. 6 forms a valid (meaningful) *db*-term in $RE(\mathcal{D})$, for example, $DAB$, $DADADADAD$ ....

Now, no matter which presentation (Fig. 5 via Fig. 6) one references, the queries in 8.1.4 can be easily expressed in a first order logic with the ordering relations as logic functions.

Query "*Can D be reached from B*?" is represented as

$$B \leq_{v\text{-}f} D$$

Since $D\ A\ B \leq_{v\text{-}f} D$ is true by Definition 8.2.2, and $D\ A = A$ and $A\ B = B$ by the definition of the database $\mathcal{D}$ in this example, then $D\ A\ B = B$. Therefore the answer to the query "$B \leq_{v\text{-}f} D$" is true.

Query "*Are nodes A and D along a circle in the graph*?" is represented as

$$A \leq_{v\text{-}f} D \wedge D \leq_{v\text{-}f} A$$

Similar to the previous query, it can be shown that $D$ has a path to $A$, that is, $A \leq_{v\text{-}f} D = true$; and $A$ has a path to $D$, that is, $D \leq_{v\text{-}f} A = true$. Then the answer to the current query is true.

It is coincident that each *db*-term in Fig. 6 is corresponding to a retraction: an identity function whose domain is identical to its range [33]. That is, $\forall M \in \mathcal{D}$ in Fig. 5, $M \circ M = M$, where $g \circ f \equiv \lambda x.\ g(f(x))$. More generally, the notion of arg-loop exactly reflects the concept of fix-point of functions [34]. For example, the infinite $RE(\mathcal{D})$ of the directed graph in Fig. 4 has the finite presentation of the EP database. The analogy would be that the infinite list x = (a, (a, (a, ...))) has the finite presentation x = (a, x) [27].

This example used arg-loop functions to represent directed graphs. Are there data applications that can be "naturally" represented by fun-loop functions? There is not much the author could thing of far up to now. However, fun-loops will have applications in defining recursive data types like lists and trees, and thereby in defining data schemata for EP databases. Since defining data schema is beyond the scope of this paper, it will not further discussed.

## 10 Data Dynamics.

So far the ordering relations developed in Section 8 were based on the equality ($=_{EP}$) of the EP data model. These relations would be too loose in dealing with data dynamics in the real world. For example, the equality *Plane Obj1 Color* = "*black*" induces

1. *Plane Obj1 Color* is the application of *Plane Obj1* to *Color*; or it is equivalent to itself.
2. "*black*" is the application of *Plane Obj1* to *Color*; or it is equivalent to *Plane Obj1 Color*.

The statement 1 is expected to be true in any circumstance. But the statement 2 may not be true in database application practice. The color of the object 1 may change to gray after a while; or it has never been black even it was wrongly assigned "*black*" in the database. Similarly, if *Obj1* needs to be removed from the database, then *Plane Obj1 Color*, but not "*black*", is expected to be removed from the database. As one of the primary objectives of the EP data model, more restricted ordering relations under EP databases is developed to deal with data dynamics in this section. Simply, we give another version of the definition 8.2 by turning off the inference rules of $=_{EP}$ except for the identical equality $\equiv$.

**10.1 Definition**. 1. Given $M, N \in \mathcal{J}$, $M$ is called the *rator* of $N$, denoted as $N <_r M$, if there is a *db*-term $L \in \mathcal{J}$, such that $M L \equiv N$.
2. Given $M, N \in \mathcal{J}$, $M$ is called the *rand* of $N$, denoted as $N <_d M$, if there is a *db*-term $L \in \mathcal{J}$, such that $L M \equiv N$.
3. Given $M, N \in \mathcal{J}$, $M$ is called the *ancestral rator* of $N$, denoted as $N \leq_r M$, if there is $\bar{N} \in \mathcal{J}$, where $\| \bar{N}\| \geq 0$, such that $M \bar{N} \equiv N$.
4. Given $M, N \in \mathcal{J}$, $M$ is called the *ancestral rand* of $N$, denoted as $N \leq_d M$, if there is $\tilde{N} \in \mathcal{J}$, where $\|\tilde{N}\| \geq 0$, such that $\tilde{N} M \equiv N$.

The domain of the relations introduced above was $\mathcal{J}$ legitimately, but it is not interesting in database application practice. Readers can simply consider the relations are applied in the domains of EP databases $\mathcal{D}$, subsets of $\mathcal{J}$.

By replacing $<_{v\text{-}f}$ with $<_r$, and $<_{v\text{-}a}$ with $<_d$ in Example 8.3, the statements 1, 4 and 5 are still true, but the statements 2 and 3 are false. Like the relations defined in Definition 8.2, the relations defined in 10.1 also can be used for query expressions. For example, the expression $\{x \mid x <_r \textit{College Admin} \wedge x \in \mathcal{D}\}$ would have the same effect as 8.4.1.

It is obvious from the reduction rule 4.1.2 that each relation defined in 10.1 will be a sub set of the corresponding relation defined in 8.2

A set $X \subseteq \mathcal{J}$ under the relations defined in 10.1 is not only partially ordered, but also tree-structured.

**10.2 Definition**. A set $X$ is tree-structured under a relation $\rho$ iff
1. there is one and only one root $r \in X$ such that $\forall e \in X, \langle r, e\rangle \notin \rho$, and
2. $\forall e \in X$, and $e$ is not the root $r$, there is one and only one $e' \in X$, such that $\langle e, e'\rangle \in \rho$.

**10.3 Proposition**. 1. Let $\mathcal{D}$ is an EP database. $\mathcal{D}$ is tree-structured under the relation $<_r$, $<_d$

*Proof* 1. Under the relation $<_r$. First of all, each propositional letter $p \in \mathcal{D}$ is a root under the relation $<_r$ according to definition of the relation $<_r$. For the rest of the db-terms $MN \in \mathcal{D}$, $MN$ is not a root because $MN <_r M$. Therefore the propositional letters are the only roots in the database $\mathcal{D}$ under $<_r$. Secondly, let $M \in \mathcal{D}$. If there were two *db*-terms $N_1, N_2 \in \mathcal{J}$ such that M N $<_r N_1$, and M N$<_r N_2$. Then, there are two other *db*-terms $L_1, L_2 \in \mathcal{D}$, such that $M \equiv N_1 L_1 \equiv N_2 L_2$. Then it must be true that $N_1 \equiv N_2$.
2. Under the relation $<_d$. the proof is similar to Proof 1.

**10.4 Example** 1. In order to identify the majors of the students studied in the *College* in Fig.2, the database needs to insert an additional function *Major* for each student under *College Admin*. To make this update in the database, The query 8.1.1 shall be given to retrieve all the students registered in the College. The query could be expressed by 8.4.1, or an expression with $<_r$. Then a *Major* function would be inserted under each student of *College Admin*. *College Admin John Major*, for example, would be a meaningful *db*-term in $\mathcal{D}$.

2. To efficiently assign the values of the majors of the students in the example above, the system administrator would initialize "*CS*" as the values of the majors of the students who take the class "*CS100*". A pseudo expression could be,

$x$ *Major* := "*CS*", where $x <_r$ *College Admin*, and *College CS CS100* $x \neq \perp$.

We wouldn't worry about the integrity of EP databases, and thereby the correctness of the entire theory of the EP data model after databases had updating operations if the updating is not breaking the rules[2] imposed by Definitions 2.4 and 4.3. Adding functions or updating values of functions such as the examples given above don't break the rules of EP database as it can be shown easily. Deleting a db-terms from database, however, is tougher. For example, What is the implication of removing *John* from the *College*? Here, another relation, called *biography* in [38], is introduced. A query expression with this relations can retrieve the complete *db*-terms relevant to a given *db*-term in a *EP* database.

**10.5 Definition**. Let $\mathcal{D}$ be a EP database, and $M \in \mathcal{D}$, $x \leq_B M$ if $x \in \boldsymbol{B}(M)$, where $\boldsymbol{B}(M)$ is defined as

$$\boldsymbol{B}(M) = \{M\} \cup (\bigcup_{\substack{x \leq_r M \\ x \in \mathcal{D}}} \boldsymbol{B}(x)) \cup (\bigcup_{\substack{x \leq_d M \\ x \in \mathcal{D}}} \boldsymbol{B}(x)).$$

**10.6 Example** 1. Now the resulting EP database with removing *John* from the *College* in Fig. 2 would be:

$\mathcal{D} - \boldsymbol{B}$(*College Admin John*).

That is, the remaining nodes under *College* in Fig. 2 are *CS*, *CS100*, and *Admin*. It can be shown that the EP databases after removing *db*-terms by relation $\boldsymbol{B}$ will remain to be EP databases.

2. In Example 3.2, we have shown the two alternatives of defining the values of a *db*-term in an EP database $\mathcal{D}$. The *db*-term *C SQ Root* was assigned a tag *I* in Fig. 2. The alternative way was to spell out the argument/value pairs of *I* under the *C SQ Root*, as if it was done for *C SQ SQ*. These alternatives for the same function are exactly distinguished by the relation $<_r$. In Fig. 2, the expression *C SQ SQ 2* $<_r$ *C SQ SQ* has the value of *true*, but the expression *C SQ Root 2* $<_r$ *C SQ Root* has the value of *false*. When the composite function *C* is to be removed from the database in Fig. 2, the function *C SQ SQ*, or $\lambda n.\ n^4$, will be removed, but *I* will not be removed although *C SQ Root* will be removed too. One application of these alternatives is to avoid data redundancy. For example, *I* could be tagged by other *db*-terms while *C* is tagging on *I* in Fig. 1. The second application of these alternatives may allow database designers to identify the data "ownership" among *db*-terms in a database. In other words, an object may own another in one database design; but the former may not own the latter in a different database design according to the different interests of managing objects. For example, a car owns its parts in the database of a car dealer because selling a car implies selling the parts inside the car too. However, a car doesn't own its parts in the database of a auto part recycling shop because destroying a car doesn't mean destroying its parts.

3. Suppose that the *I* is removed from Fig. 1, what does the tag *I* of the *db*-term *C SQ Root* mean? The notion of the EP databases has been carefully defined in 2.4 such that no special treatment is needed for this case. Since *I* is no longer in $\mathcal{D}$, then $I \rightarrow_{EP} \perp$. While *C SQ Root* is still a valid non-leaf *db*-term in $\mathcal{D}$, *C SQ Root 2* is no longer equal to 2, but $\perp$.

In relational database management systems, the notion of data dependence was used to reflect certain dependent relationships among the data in one or multiple tables. For example, the value of a data is dependent on another one; and the removal of the later from the tables implies the mandatory removal of the former to maintain the data integrity of the databases. The data dependence was not a part of the

[2] Here, the rules from data schemata are not considered.

relational data model. The EP data model, however, have the partial ordering relations defined in this section to completely take care of the functional dependency. In other words, these relations know how to "trim" functions, instead of only atomic values.

## 11. Functional Programming

The EP data model incorporates the expressiveness of Turing-machine equivalent languages. The Turing-machine equivalent expressiveness is not what traditional data models possess. By closely looking at the proof of Theorem 6.15, we can see that the Turing-machine equivalent expressiveness of the EP data model is not from the finite number of the *db*-terms in EP databases, but the constants tagged by those *db*-terms. The constants and their applicative behavior are not governed, but accepted by the EP data model as its axioms (Axiom 6.4). In other words, the EP data model is just a framework of organizing and maintaining dynamic data (or functions), the real engine of driving the EP data model to the height of Turing-machine equivalent is a programming language. An analogy is CPUs driving machine-dependent languages, or the machine-dependent languages driving higher-level programming languages.

The style of the EP data model is similar to that of the type-free lambda calculus, or its extended functional programming languages. To be more descriptive, the EP data model would choose a functional programming language for constructing its constants and governing the computing behavior of the constants. This functional programming language may have millions of different syntactical forms. But it must have the pure lambda calculus as its foundation. The EP data model can be smoothly fitted in the functional programming language just as another "teaspoon of sugar".

Be cautious! "The whole field of information systems" is "largely untouched by functional programming" [25]. "The methodological benefits of functional languages are well known" [16], [3], "but still the vast majority of programs are written in imperative languages such as C" [23]. What do we expect the functional programming in database computing applications? And what do we expect the EP data model in energizing functional programming? Well, we are betting on the EP Data model. We know that using assignment statements is the most controversial feature of both imperative and functional programming languages. Allowing assignment statements gives imperative programmers flexibility in developing application programs; but it is a major source of bugs, and discourages parallel computing. On the other hand, prohibiting assignment statement frees functional programmers from prescribing the flow of control in program, and permits lazy evaluation and parallel computing; but it gives functional programmers "no help in exploiting the power of functional language" [16]. More specifically, "destructive updates (i.e. assignments)" are restricted with the "inferior or inappropriate data structures" [23]. And a relevant criticism is the problematic Input/Output issues [9], [25].

The EP data model could reduce the degree of the controversy of assignment statements. First of all, an extended functional language with the flavor of the EP data model will not have update operations against any data except EP databases. Comparing with assignment statements everywhere in imperative languages, the extended functional language minimizes update operations to only those reflecting the dynamics of the world. Secondly, the update operations in the extended functional language can be controlled by using transaction control technologies.

With the discussion above, the author would like to propose a more "measurable" definition of the term descriptiveness. That is, the sole purpose of a language is 1) to construct functions representing application data; 2) to alter the constructed functions to reflect the dynamic world. Then the

descriptiveness is to categorize the easiness of both constructing and altering functions. From this proposal, we can see that the traditional functional programming languages are not as descriptive as we expected. As it happens, the EP data model offers its data structures for storing higher-order functions and the corresponding ordering relations for queries and updates against higher-order functions. This is another key of the EP data model. If readers accept this measurement of languages' descriptiveness, then it is not hard to see that the EP data model, or an extended functional programming language with the flavor of the EP data model, would be more descriptive than traditional functional, imperative programming languages, and traditional data models. In the more "measurable" definition of descriptiveness above, function update operations are singled out from function construction operations although the former can be viewed as the latter. But this separation indeed reflects the database practice, and "destructive" updates are quite different from accumulative constructions.

The "ad-hoc" relational calculus in relational databases was a fact that contributed the success of the relational data model in database application practice. Computing an expression with logic variables results an arbitrarily large set of objects, instead of a single one. This distinguishes logic-oriented programming languages from functional-oriented programming languages. Then how does a logic-oriented programming language fit in the paradigm of the extended functional programming with the flavor of the EP data model? Since the well-formed formulas of the first-order language are nothing, but functions with truth values as the range, adding a first-order language with popular logic symbols into the extended lambda calculus is expected. Many research works have been done in the related area [13]. And the author has informally given some examples with the mixture of logic symbols with λ–calculus. Data management has been isolated from programming languages in both research and practices [20]. From the EP data model, we can see some commonality of data management with programming languages.

## 12 Conclusion

The EP data model is expected to be a highly descriptive and expressive language which can "naturally" express "arbitrary" data in database application practice. The descriptiveness stems from the simple notions of the EP data model, and the embedding ordering relations in EP databases. Readers could react on the flavors of the EP data model from the examples used in this paper. The expressiveness, on the other hand, is guaranteed by computable functions EP databases manage. The EP data model is not only a data type capturing application data, but also a language. Its *db*-terms are both the names of managed data and query expressions against managed data. Certain queries and updates on data and the relationships among the data can be expressed by built-in ordering relations existed between *db*-terms in a first-order logic language. From the examples in the paper, It is obvious that certain traditional fix-point queries, such as school organization hierarchies, paths and circles in graphs, can be straightforwardly expressed by the ordering relations defined in this paper. It is the ordering relations again that free developers from considering data constraints or functional dependency in database update operations.

Thought of as a higher layer abstraction, the EP data model offers an interface interconnecting an EP database with others and with heterogeneous computing applications. The EP data model doesn't see the physical presentation of data such as software code or data locations, but data itself, as if NFS (Network File System) was for distributed file systems. Thought of as a higher layer abstraction, the EP data model could become "system administrators" in software engineering "who" can update software source code, query on software source code, and even execute the corresponding binary code. Do we need lower-layer file systems? We don't have to, because the directory of file systems is a special case of the EP data

model. In addition, the EP data model may have influences on other computing applications if the EP data model is used as a data type in programming languages.

**Reference:**


[1] S. Abiteboul, R. Hull, and V. Vianu. "Foundations of Databases". Addison-Wesley Publishing Company, 1995.

[2] A. V. Aho and Jeffrey D. Ullman. "Universality of Data Retrieval Languages". Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Taxes, January, 1979, page 110 - 120.

[3] John Backus. "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algegra of Programs". Communication of the ACM, Vol. 21, Number 8, 1978, page 613 - 641.

[4] H. P. Barendregt. "The Lambda Calculus - its Syntax and Semantics". North-Holland, 1984.

[5] A. K. Chandra, and D. Harel. "Computable Queries for Relational Data Bases". Journal of Computer and System Sciences 21, 1980, Page 156 - 178.

[6] A. J. T. Davie. "An Introduction to Functional Programming Systems using Haskell". Cambridge University Press, 1992.

[7] R. E. Davis. "Truth, Deduction, and Computation - Logic and Semantics for Computer Science". Computer Science Press, 1989.

[8] R. Elmasri and S. B. Navathe. "Fundamentals of Database Systems, Second Edition". The Benjamin/Cummings Publishing Company, Inc., 1994.

[9] Andrew D. Gordon. "Functional Programming and Input/Output". Cambridge University Press, 1994.

[10] P. M. D. Gray, K. Kulkarni, and N. W. Paton. "Object-Oriented Databases, A Semantic Data Model Approach". Prentice Hall, 1992.

[11] Stephane Grumbach, and Jianwen Su. "Finitely Representable Databases". SIGMOD/PODS 94, page 289 - 300.

[12] M. Gyssens, J. Paredaens, J. V. Bussche, and D. V. Gucht. "A Graph-Oriented Object Database Model". IEEE Transactions on Knowledge and Data Engineering. Vol. 6, No. 4. August 1994, page 572 - 586.

[13] M. Hanus, H. Kuchen. "Integration of Functional and Logic Programming". ACM Computing Surveys, Vol. 28, No. 2, June 1996, page 306 - 308.

[14] Gerd G. Hillebrand, and Paris C. Kanellakis. "Functional Database Query Languages as Typed Lambda Calculi of Fixed Order". SIGMOD/PODS 1994, page 222 - 231.

[15] J. R. Hindley. "Basic Simple Type Theory". Cambridge University Press, 1997.

[16] John Hughes. "Why Functional Programming Matters". The Computer Journal, Vol. 32, No. 2, 1989, page 98 - 107.

[17] W. Kim. "Introduction to Object-Oriented Databases". The MIT Press, 1990.

[18] W. Kim, Hong-Tai Chou, and Jay Banerjee. "Operations and Implementation of Complex Objects". IEEE Transactions on Software Engineering, Vol. 14, No. 7, July 1988.

[19] C. P. J. Koymans. "Models of the Lambda Calculus". Information and Control, 52, 1982, page 306 – 332.

[20] D. Maier. "Why isn't There an Object-Oriented Data Model?". Information Processing 89, G.X. Ritter (ed), Elsevier Science Publishers B. V. (North-Holland), page 793 - 797.

[21] Albert R. Meyer. "What is a Model of the Lambda Calculus?". Information and Control 52, 1982, page 87 - 122.

[22] B. Meyer. "Introduction to the Theory of Programming Languages". Prentice Hall, 1990.

[23] Chris Okasaki. "Purely Functional Data Structures". Cambridge University Press, 1998.

[24] G. E.. Revesz. "Lambda-Caculus, Combinators, and Functional Programming". Cambridge University Press, 1988.

[25] Colin Runciman. "Applications of Functional Programming". UCL Press, 1995.

[26] Peter Schauble. "On the Expressive Power of Query Languages". ACM Transactions on Information Systems, Vol. 12, No. 1, January 1994, Page 69 - 91.

[27] D. Scott. "Outline of a Mathematical Theory of Computation". Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems, Princeton University. 1970, page 169 - 176.

[28] D. Scott. "Models for Various Type-Free Calculi". Suppes et al. 1973, page 157 - 187.

[29] David W. Shipman. "The Functional Data Model and the Data Language DAPLEX". ACM Transactions on Database Systems, Vol. 6, No. 1, March 1981, Pages 140 - 173.

[30] R. M. Soley, and C. M. Stone. "Object Management Architecture Guide, Revision 3.0, Third Edition". John Wiley & Sons Inc., 1995.

[31] D. Steedman. "X.500 - The Directory Standard and its Application". Technology Appraisals, 1993.

[32] R. R. Stoll. "Set Theory and Logic". W. H. Freeman and Company, 1963.

[33] J. E. Stoy. "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory". The MIT Press, 1977.

[34] Alfred Tarski. "A lattice-Theoretical Fixpoint Theorem and Its applications". Pacific J. Math. 5 1955, page 285 - 309.

[35] J. D. Ullman. "Principles of Database and Knowledge-base Systems, volume I". Computer Science Press, 1988.

[36] J. D. Ullman. "A Comparison between Deductive and Object-Oriented Database Systems". 1993, page 263 - 277.

[37] C. P. Wadsworth. "The Relation between Computational and Denotational Properties for Scott's $D_\infty$-Models of the Lambda-calculus". SIAM J. Comput. Vol 5, No. 3, September 1976, page 488 - 521.

[38] K. H. Xu and B. Bhargava. "An Introduction to Enterprise-Participant Data Model". Seventh International Workshop on Database and Expert Systems Applications, September, 1996, Zurich, Switzerland, page 410 - 417.