Preliminary Ph.D. Thesis Proposal:

EP Data Type from Function Space

Kevin Houzhi Xu
Purdue University
West Lafayette, IN 47907
Email: xu16@cs.purdue.edu

Abstract

Currently a typical database application has three forms of data that need to be translated back and forth. The first form is relation (table) in which the application data is stored in a relational DBMS. The second form is user-defined data type in host programming language spaces. The third form is a common data exchange protocol like XML, in which the data is exchanged with other applications across distributed computing environments. The data translations have to be manually done to make the data application work and the data is copied from one computing environment to another. As a result, there are overhead in system performance, system development, and system maintenance. Another motivation of this research is that the advanced database applications in data streams are looking for DBMSs that can treat triggers and data equally. This research is to propose a new data model or a language that supports a data type shared by DBMSs, programming languages, and data communications. The sharable data type can be used for arbitrary application functions including data and triggers repository.

1. Problem Statement

Currently a typical database application has three forms of data that need to be translated back and forth. The first form is relation (table) in which the application data is stored in a relational DBMS. The second form is user-defined data type in host programming language spaces. The third form is a common data exchange protocol like XML, in which the data is exchanged with other applications across distributed computing environments. The data translations have to be manually done to make the data application work and the data is copied from one computing environment to another. As a result, there are overhead in system performance, system development, and system maintenance. Another motivation of this research is that the advanced

database applications in data streams are looking for DBMSs that can treat triggers and data equally.

This section analyzes the existing DBMSs, programming languages, and relevant previous research work in support of this research statement.

1.1 Data Types in Databases

There are hundreds of data models having been proposed in the past decades. But all of them can be classified as the relational data model; graph oriented models; and hierarchical data models.

Relational Data Model

The relational data model has been dominant in the industry. It is simple in database design and sound in theory [1], [18]. The drawback is its flat structure of relations [1]. As far as this research's concern, we argue that the flat structure causes the necessity of a secondary data exchange model for data communication in distributed computing environments. The reason is that the communication expects messages to carry application-oriented data, like a purchase order or an organization. However, the application-oriented data may have to be spread across multiple relations in a relational DBMS. This would imply multiple messages for a single transaction at application level if the relational data model was the data exchange model. See more discussion on data exchange models in Section 1.2.

The relational algebra provides DBAs and programmers a declarative query language. It is the major contributor towards the success of the relational data model. On the other hand however, Relations limit the relational algebra (and Datalog) only to be the functions from relations to relations. This causes the disconnection between programming languages and relational DBMSs, and was called impedance mismatch in [32], [35]. It is clear that not all the functions of a database application are from relations to relations. For example, they could be simply from relations to integers, which cannot be expressed in the relational algebra or Datalog. Therefore, different data types in a host programming language have to be used to transform relations to other data values.

Of course, the most popular criticism against the relational data model is its lack of power in expressing rich semantics and queries of database applications, again due to the flat structure of relations. This forces a host programming language on the top of DBMSs to make up whatever is missed from relational databases.

Graph Oriented Data Models

There are many graph oriented data models having been proposed over many decades. Although they were proposed with different emphasizes, the basic structure is a directed and labeled graph [11], [22]. The earliest graph model was called the network data model (CODASYL) [56], [18] in parallel to the relational data model and hierarchical data models for traditional database applications. The Entity-Relational (ER) Data model is the most popular tool in database design, but it offers no query language. DAPLEX [45] was called a functional data model, it is actually a graph oriented model [22]. Object Oriented data models were categorized into graph oriented [22] though the focus was on class inheritance and method embodiment with classes. Complex Objects model is also graph oriented [1]. The latest data exchange models like OEM [38] and XML [11] are claimed to be graph oriented, where semistructured data is the focus for web-based database applications. The earlier graph oriented models, Complex Objects typically, offers a mix set of data structures like record, list, set, nested set, etc.. The latest graph oriented models, XML typically, are refined to simpler structures, e.g., single rooted, labeled and directed edges, and atomic values at leaves. The graph oriented data models offer more powerful query languages like path expressions than the relational algebra and Datalog [11].

The good thing about the graph oriented data models is its flexibility in representing application data as complex as it is needed. The drawback is its difficulty in offering a sound query language. The big assumption of the graph oriented data models is that they allow cyclic data (or they could be categorized into the hierarchical data models). It follows an immediate controversial however. For example in Figure 4 in Appendix D, the query of "find the values of all the names that reachable starting from the path `college.Dept` could be expressed like: `select college.Dept.*.name,` here the symbol '*' is the wildcard matching any path. Since there is a cycle in the database, this query could lead to an infinite loop in evaluating. We may have artificial ways to re-define the query behavior and to improve system performance, but we would lose the legitimate of the graph oriented data models. In addition, certain data dependencies like inverse dependency is missed from a graph base data model unless additional edge was added into databases [12], [13].

Hierarchical Data Models

Tree structure is efficient in capturing database application semantics (data dependency) and in system performance. Human beings think of and organize the real world in a hierarchical fashion, and file systems in computers are organized in hierarchy. Primary keys and foreign keys are

utilized to form hierarchical structures (functional and inclusion dependencies) in Relational DBMSs. The exchange models X.500 [57] and CORBA/IDL [47] are hierarchical data models. Although XML/Semi-structured data has been generically claimed as graph oriented data model [11], representing hierarchical data in XML/Semi-structured data is the most appropriate and efficient [11], [16], [26]. Even the relational DBMS use primary keys and foreign keys for more semantics. The foreign keys for inclusion dependency actually reflect the hierarchical structures among the relations.

The key concept in a hierarchical data model is the parent-child relationship [18]. A single parent dominates its children; and a child can have only one parent. A removal of a parent must imply the removal of its children. The parent-child relationship is built-in (no user-defined relationship label is necessary). Therefore, the transitive-closure queries like printing all the children in the tree under a parent are uniquely expressible in hierarchical data models [29], [57].

Obviously the world is not always that simple. An object may have to be dominated by multiple objects; and more, multiple objects could cyclically depend on each other. For example, the success of a person not only depends on himself or herself, but also environment. A directed cyclic graph is another example hardly making the hierarchical data models acceptable.

There are normally two ways of dealing with the data having multiple parents in hierarchical data models. The first is to duplicate all the parents under the child except one parent that is taken as the parent of the child. This would cause data redundancy and increase the complexity of maintaining data integrity. The second way of dealing with multiple parents is to use virtual parent-child relationship, e.g., using a pointer from the child record to the second parent. However, a virtual parent doesn't act as a dominating parent, but a dependent attribute. When the virtual parent is to be removed from database, the child would not be removed automatically from database. This ruins the principle of the hierarchical data models.

1.2 Data Types in Data Communications

As discussed earlier, many data exchange models like X.500, CORBA/IDL, OEM, and XML have been proposed. XML is the most accepted one in Web-based database applications, where unstructured data or semi-structured data is assumed. There are two ways of storing XML data. One approach is to store XML data into relational DBMSs, which needs data translation between relations and XML. SilkRoute [20] is an effort to automatically translate data based on the relational data schemas and XML DTD. However, a generic optimization on data translation in web-based distributed environments is NP-complete [36], [20].

The second approach is to store data in the graph oriented data models, e.g., Lore [21], StruDel [19], and IRA-OSS [17]. This approach, however, cannot get rid of the problems existed in the traditional graph or hierarchical oriented data models.

1.3 Data Types in Programming Languages

The primary data types are atomic types like integers, strings, characters; arrays; list; variants; trees; and records; and combinations of all in programming languages. They provide programmers flexible ways of constructing objects. But constructing objects itself and building operations against the objects on data structures are expensive; and further maintaining and changing the data structures are difficult in imperative languages. Declarative languages like logic and functional programming languages offer built-in data structures like list in functional programming language and relation in Prolog. For example, the expression: `1: [2, 3, 4] = [1, 2, 3, 4]` is a common practice for inserting an element into a list in functional programming languages. Built-in data structures easy software development effort.

The data types in programming languages are run-time objects. Explicit code has to be written in programming language that stores or retrieve data from data storage. Keeping data persistent, e.g., storing data in storage in the form of its data type, became an issue as soon as database management systems did so in 1970s such as Pascal/R [4]. The researchers recognized the economy of unifying the paradigms of programming languages and database management systems. In 1980s and 1990s, there are many proposals made toward this direction, e.g., Galileo (surveyed in [4]), Functional Object Language [30], Machiavelli [35], PFL [46], BULK [42],Orthogonal Persistent Java [28], [3]. These proposals can be classified into two categories.

The first approach is to make data types in programming languages persistent. The latest development on this direction is called orthogonal data persistent Java [28], [3], [27]. It means that all the data types from atomic types to complex types are equally subject to be persistent. In addition, all the source code is also subject to be persistent. Eliminating the overhead of connecting database and programming languages is an obvious obje ctive. The second objective is to provide a continuation of computing after a computing execution is accidentally terminated [3]. This approach focuses user-defined data types and makes the life of data transparent to programmers by analyzing data reachability [27]. On the other hand however, this approach doesn't address useful declarative query languages like relational algebra.

The second approach is to construct higher-level functions on the top of relations in a functional programming language. To reuse higher-level functions over multiple DBMSs, constructing

polymorphic functions was proposed in [33]. These functions, however, are from relations to relations, and then were called relation transformers. To overcome the limitation of the relation transformers, the work in [8], [9], [10] explicitly proposed structural recursion on sets, which could output values that are not relations. But the framework was still based on the structure recursion on sets. For example, the following expression from [8]

```
Let f(Nil) = N
| f(Cons(x, l)) = C(x, f(l))
in … f … f …
```

is to define a structural recursion that walk through a list while arbitrary operations can be done against element on the list. However, Not all the practical experiences want the expensive linear travel against lists or sets.

The work in [23], [24], [25] elegantly and comprehensively encodes the relations, relational algebra, transitive closure, complex objects, and object-oriented methods into a typed lambda calculus; and classifies the complexity classes of query languages in the orders of lambda terms.

1.4 Data Types for Arbitrary Functions

So far the data types we discussed earlier are normally representing finite data. To represent continuous time and geographical space, the temporal and spatial, or called constraint, database management systems were introduced based on the relational data model [40], [41]. However, there has been no data type that offers a data structure and operations against the data structure for arbitrary infinite data (or functions). As a result, a data exchange model, e.g., XML, has to leave infinite data unspecified. When an arbitrary function needs to be communicated across different systems, it has to be specified in a different language. For example, Java Scripts are normally sent from web servers to web clients along with HTML or XML messages [34].

On the other hand, a new class of advanced database applications, data streams, is emerging. The researchers in this area observed that the traditional database management systems are "Human Active and DBMS Passive (HADP)". In other words, they treat triggers and alerters as second-class citizens [39]; and their implementation don't scale to a large number of triggers. The data stream applications, however, are trigger-oriented. Their role is to alert humans when abnormal activity is detected. This is a "DBMS-active, Human-Passive (DAHP)", or it is desired to treat data, triggers, and alerters equally [5], [6], [15], [49].

2. Proposal Detail

This section describes the philosophy of the EP approach, gives a formal definition for the EP data model or language, and briefly discuss the ordering relations that provide advanced set-oriented primitives.

The detailed grammar for the latest EP language system is provided in Appendix A. Many sample expressions or application examples are given in Appendix B. From the example, one may get more sense about the EP language that connects database management and programming language together. Many examples about database design, e.g., school administration database, a factorial function, and a directed cyclic graph, and their graphical presentations were provided in [51], [54]. Appendix D reiterated the school admin database in comparison with other data models.

2.1 Approach and Objectives

As we know, a *function* is a binary relation so that no two distinct members have the same first coordinator. In other words, a relation *f* is a function iff it meets the following requirements:

1. The members of *f* are ordered pairs.
2. If <*x*, *y*> and <*x*, *z*> are members of *f*, then y = z.

x is called an argument of the function *f*; and *y* the result of *f* by applying to *x*, and written as *f x* = *y*. In a different way of describing the relationships among a function and its argument/result pairs, *f x* is an application of *f* ; *y* is an image of *x*; and *f* is called a higher-order function where *x* and *y* are functions. The collection of all the (higher-order) functions is called the function space, or a λ-model [43], [7]. The functions in the space are interconnected with the relationships of function, argument, and image.

This research is to propose a data type – EP Data Model, or EP Language. The EP data model has a data structure and operations against the data structure. An instance of the data structure represents a finite set of functions from the function space; and the functions are interconnected with the relationships of function, argument, and image. By extending (typed) lambda calculus, the operations against the data structure can be from arbitrary types to arbitrary types. In turn, the operations are expressed as data in the data structure, which treats data and arbitrary functions equally. The objectives of the proposal are the followings:

1. The EP data structure is rich in database design, and sound in database update and database integrity. It is a practically effective data structure for data repository.

2. A few built-in query primitives against the EP data structure offer powerful set-oriented query expressions in the place of the relational algebra and Datalog. The query primitives offer optimized or the best system performance for the equivalent query operations.

3. The EP data structure offers a language, the EP language that can be used to construct functions from arbitrary types to arbitrary types. This language is based on the lambda calculus having its λ–terms and β–reduction. This would allow any programming language to adopt the EP data structure. In other words, the EP data structure is the bridge merging the paradigms of database management and programming languages together.

4. The EP data structure is rich for data communication in the place of XML. In a distributed computing environment where all the components support the EP data model, the data exchange protocol is the EP data model itself that offers application-oriented messages, and no data translation is needed.

5. The EP data structure is extended to incorporate infinite data (or arbitrary functions). This would eliminate the necessity of a secondary language like JavaScript to carry functions in data communication. Further it provides a mechanism to treat data and triggers equally in data stream applications.

3. Formal Definition

This section gives the syntactical definition of the EP language. The definition hasn't included the type system yet.

Definition The following symbols are allowed in the EP language:

A countable set of constant symbols C;

A countable set of identifiers (or called prepositional symbols) P;

A countable set of variables V, each of which will have a range; and

Parentheses: (,)

A variable *$x* has a range in the implemented EP system. For example:

```
create fac $n:[$n > 0] = $n * fac ($n - 1);
```

If there is no range declared for a variable, it means that the variable ranges all possible values.

The range is not specified syntactically in this formal definition; but it exists semantically and is denoted as *range* ($*x*) when it is needed in the rest of this section.

Definition Let Γ be the set of terms expressible in EP language:

1. $v \in \mathsf{V} \rightarrow v \in \Gamma$;
2. $p \in \mathsf{P} \rightarrow p \in \Gamma$;
3. $c \in \mathsf{C} \rightarrow c \in \Gamma$; or
4. if $M \in \Gamma$, and $N \in \Gamma$, then $M\ N \in \Gamma$.

Γ is the complete set of terms can appear in the EP language. *M N* is called an application term as usual in lambda calculus.

Definition Let $\Im$ be the set of terms that can appear in a database:

5. $v \in \mathsf{V} \rightarrow v \in \Im$;
6. $p \in \mathsf{P} \rightarrow p \in \Im$;
7. $c \in \mathsf{C} \rightarrow c \in \Im$; or
8. Inductively, if *M, N* $\in \Im$, and if *M*'s inner-most subterm is not a constant and *N* is not an application term including a variable as a subterm, then $M\ N \in \Im$.

It is certain that $\Im \subset \Gamma$. As examples, *4* (constant), p_1 (identifier), p_2 (identifier), and *$x* (variable) are in $\Im$. So do the terms p_1 *4*, p_1 *$x*; but not *4* p_1 and p_1 *(p_2 $x)*. The last constraint says that a database cannot extensively re-define a constant by constructing subordinators under the constant as a root. Further it also constrains that a node (or term) with variable(s) as subterm(s) cannot be applied as an argument to another node in a database.

Definition An EP database is a finite set $\mathsf{D} \subset \Im$:

1. if $M\ N \in \mathsf{D}$, then $M \in \mathsf{D}$, $N \in \mathsf{D}$.
2. if $M \in \mathsf{D}$, and there is no $N \in \mathsf{D}$ such that $M\ N \in \mathsf{D}$, then *M* may (or may not) be assigned a term $Q \in \Gamma$ (denoted *M* (*tag* ≡ *Q*), and it is required that FV (*Q*) ⊆ FV (*M*).

For example, the textual presentation of the database in Fig. 1 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*)}. The factorial function is defined in a database: {*fac, fac 0 (tag* ≡ *1), fac $n:[$n>0] (tag* ≡ *$n * fac ($n – 1)*}.

When *M, M* N_1, *M* N_2, ..., *M* N_n are in a database, then *M* is viewed as having been extensively defined, and semantically equivalent to an abstraction in the lambda calculus. As an additional

constraint as to be defined later, here N_1, N_2, ..., and N_n are not overlapped both syntactically and semantically.

The restriction FV (Q) $\subseteq$ FV (M), here FV stands for the set of Free Variables, says that all the terms defined in a database must be closed. For example, *fac $n (tag ≡ $n * $m)* is not allowed in a database. This requirement bundles all the variables in Q with those in M.

Note that in a term M ($tag \equiv Q$) of a database, here M is called a leaf node, all the variables in M will have their ranges no matter they are explicitly defined or not. Further, all the terms in Γ have either given or derived types under a given database D. Therefore, the EP language is a typed system though the type notion has not been formally introduced yet so far. Further work on types and polymorphism needs to be done with a guideline from [13].

Definition Given a database D, a term $M \in \Gamma$ is a *db-redex* if

1. $M \in$ P, but $M \notin$ D.
2. M is an application term, e.g., $M \equiv P\ Q$, where $P \in$ D, and $Q \notin$ D.
3. M is a leaf node, e.g., M *(tag ≡ Q)*.

Definition A term $M \in \Gamma$ is a normal form, denoted as *nf* (M), if there is not a *db-redex* in M as a subterm.

Since variables are allowed, the beta-reduction has its correspondence in the EP language. For a lambda term ($\lambda\ x.\ M$) N in the lambda calculus, the beta-reduction is denoted as $M\ [x := N]$. Similarly a term $M \in \Gamma$ in the EP language, M may have an environment [*env*] during EP evaluation. [*env*] is a set of variable-value assignments, e.g., [*env*] = {$\$x_0 := V_1$; ...; $\$x_n := V_n$}, where n >= 0.

Definition Let D be a database, $M \in \Gamma$ is *db*-reduced to N, denoted as $M \rightarrow_{db} N$, if

1. $M \in$ D, M is a leaf node, e.g., M *(tag ≡ Q),* then $N \equiv Q$ *[env]*. Semantically equivalent to the β–reduction, here Q [*env*] is the term whose variables are substituted with their values provided in the [*env*]. The environment [*env*] is set to empty after the substitution.
2. $M \equiv a$, here $a \in$ P, but $a \notin$ D, Then $N \equiv \perp$.
3. $M \equiv \perp.\ Q$, where $Q \in \Gamma$. Then $N \equiv \perp$.
4. $M \equiv Q\ L$, where $Q \in$ D; $L \in \Gamma$.
   4.1. $N \equiv M\ N_i$ if there is a N_i such that $M\ N_i \in$D, and there exists a term N_i' such that $L \rightarrow_{db} N_i$' and $N_i \rightarrow_{db} N_i$',

4.2. $N := (M\ \$x)$, and $[evn] := [evn] \cup [\$x := nf(L)]$ if there is a term $M\ \$x \in \mathsf{D}$, and $nf(L) \in range(\$x)$; or

4.3. $N \equiv \perp$ if neither the condition 4.1 nor condition 4.2 is satisfied.

Since an EP database defines functions, an additional constraint is needed to keep the integrity of databases.

Definition Given a $M \in \mathsf{D}$, for all the $N_1, N_2, ..., N_n$ such that $M\ N_1, M\ N_2, ..., M\ N_n \in \mathsf{D}$, then the database D must further satisfy the condition: $range(N_1) \cap range(N_2) \cap ... \cap range(N_n) \equiv \varnothing$. Here $range(N_i) \equiv N_i$ if $N_i \notin \mathsf{V}$ (variables).

The semantics has been given in [54], [53], where the Church-Rosser, Consistency, and Soundness, and Turing-Computing completeness were provided. The idea is to encode all the prepositional symbols into the corresponding lambda terms in the pure lambda calculus; and all the reduction rules also are encoded to the beta reduction. Note that the syntax in [54] and [53] was not exactly following the grammar defined in this proposal. For example, all the lambda abstractions (then variables) were not explicitly presented in the data structure of the EP data model, but they were treated as constants. Then the reduction behavior (the beta reduction) of the constants then was presented as axioms by using applicative structures.

2.3 Ordering Relations

The set of lambda expressions in a database embed pre-ordered (reflexive, transitive, but not anti-symmetric) semantically; and partially ordered (reflexive, transitive, and anti-symmetric) syntactically. The pre-ordering and the partial ordering relationships are carried by the function-argument-value relationships. The partial orderings provide the theoretical guidelines for maintaining data integrity in data update and delete; and for improving system performance. Both partial orderings and pre-orderings provide a set of rich, built-in, and deductive operators for set-oriented queries. The queries like:

1. "tell me all the information about the student John Smith", and
2. "is there a cycle between two vertices in a directed graph?"

can be simply expressed by some of operators. See more discussion in [51], [54].

The introduction of the partial orderings brings a phenomenon not usual in the lambda calculus, e.g., the db-reduction rules by adding the partial orderings look not Church-Rosser. Appendix C provides a detail discussion, stating that the partial orderings are our intention for database

management; and that the db-reduction rules are still Church-Rosser by forbidding the db-reduction process against the operands of the partial orderings.

3. Related Work

The EP data model can be viewed as a different approach to hierarchical data models. When a child has multiple parents. The EP data model takes one of the parents as the function, and the other parents as curried arguments. For example in figure 4 in Appendix D, the course CS101 takes the argument `College Admin (SSD John)` and forms the new node `College CS CS101 (College Admin (SSD John))`. In the EP data model, both function and argument dominate the application (child) which preserves the semantics of hierarchical relationship (inclusion dependency). The removal of either a function or an argument implies the removal of the child. In [55], functions were called enterprises; and arguments as "participants". This is how the terminology EP (Enterprise-Participant) came from.

In short, the EP data model uses one type of links (solid arrows) for data sharing and the other type of links (dash arrows) for multiple parent-child relationship, while the virtual parent-child relationship approach of the traditional hierarchical data models has only one type of links (pointer) for both different circumstances.

In comparing with traditional database management systems and programming languages, the EP data structure is the key that makes it different from the others. Here is a brief summary of the main areas the EP language finds its uniqueness.

1. Built-in data structure for database programming languages. Like lists in LISP and ML and relations in Prolog, Pascal/R [4], and Machiavelli [35], the EP data structure is built-in for coding application programs. This would save the effort of constructing user-defined data structures or object classes in imperative languages.

2. Data structure matching both database management and programming language. The infamous "impedance mismatch" said that the type systems in programming languages don't match the data structure in data models [29], [33]. Relations offer functions from relations to relations via the relational algebra and relation transformers, but there is no efficient way of constructing functions from arbitrary types to arbitrary types [10]. Graph oriented data structures offer regular path expressions from sets to set, but the path expressions don't provide a sound semantics against cyclic data as we discussed earlier. Hierarchical data structures offer advanced primitives for transitive closures, but they are not generic enough for arbitrary data, as we discussed earlier. The EP data structure generalizes the hierarchical data structures and adopts the

structures of the function-argument-image relationship in the function space. Based on the lambda calculus, the extended λ–terms are the form that identifies the nodes in a database; and expresses queries from λ–terms to λ–terms; and carries messages in data communications. It is sufficient to make the EP language unique in approaching the issues of database programming languages – unifying the paradigms of database management and programming language.

3. Data exchange model for data communication in distributed database environments. Semi-Structured Database is the most accepted data exchange model so far. It is either viewed as a hierarchical data model or a graph data model [11]. If it is a hierarchical model, then the multiple parents relationship with children is the major obstacle in data presentation, as we discussed earlier. If it is a graph model where cyclic data is allowed, then expressing path expressions against cyclic data is a controversial. Using the same EP data structure and the extended λ–terms, however, all the issues including infinite data (arbitrary functions) around data communication have their resolutions under the EP data model.

4. Data structure that supports a higher degree of polymorphism in querying data. Appendix D gave a few examples, where the query expressions against relational databases and graph oriented databases are more tedious and complicated than those against EP databases. The reason is the built-in primitives (pre-ordering and partial ordering relations, such as <=a, <=A, etc.) that are uniquely available in the EP language. Polymorphism allows a function to take operands with different types [14]. If we view a database application as a type, then different database applications are different types – different schemas. However, when we express queries against data with pre-orderings or partial orderings, we have to know and reference the data schemas in detail in the relational data model or a graph data model. Then a query against a database application cannot be reused against another. But the ordering relations in the EP language are the polymorphic functions that takes data from different database applications as argument and outputs ordering data.

Viewed differently, the EP ordering relations (then transitive [54]) are actually transitive closures. In the relational database application, some hierarchical data like college – department – course – student – grade is a transitive closure, but the length of the ordering is known in data schema. Then the query expressions for the ordering must reference the exactly attribute names in a graph oriented data model or the relational data model. But this is not necessary in the EP language and in the composite objects in [29]. On the other hand, some hierarchical data like `assembly = {parts, assembly}` has unknown length of ordering during data schema design time. Then the query is given in Datalog in the relational data model; and in regular path expression in semi-

structured data. The EP ordering relations have the similarity in this case, but go further for more expressiveness by incorporating the *db*-reductions.

5. The data structure with the partial orderings (<=A, <B, etc.) that precisely defines the behaviors of database update (see more discussion in Appendix C and [54]): Removal of a node `M` implies the recursive removal of all the nodes `N`, where `N <A M` or `N <B M`. Updates in the relational model, hierarchical models, and graph models are problematic in some ways as we discussed earlier. The problem with the relational model is discussed in [18], [1]. The addition of primary keys and foreign keys reduces the update anomaly. But the problems from hierarchical models, that have been discussed earlier, are inherited. With graph data models, update is done by adding/deleting individual edges or values; the nodes related by the EP partial orderings obviously cannot be removed at one shot according to the application semantics.

6. Data structure allowing arbitrary functions including triggers. Triggers are the additional mechanism on the top of databases; and the triggers in relational DBMSs are not sufficient for data stream applications [15], [5]. Triggers as a part of functions in the EP language, however, are equally treated as data in the EP data structure.

7. Data structure with embedded trees for better performance. One of the main reasons for the popularity of hierarchical data structures in programming language systems and data presentations is their support of better performance. Since the EP data structures are tree-structured with multiple dimensions, it is expected that the EP language improves the system performance. For example, the recursive queries in EP language are optimized while it is hard in Datalog.

4. Future Work

The EP data model or language has been proposed in [51], [54], [55]. The future work is to implement the EP language systems with optimization, revisit the previous semantics for the EP language, and have a few case studies in traditional database applications and data streams.

1. Implementation. A language system for the EP data model will be implemented that has the EP data structure, the *db*-reduction rules against the data structure, the ordering relations, set-oriented operations, and data update operations. The implemented system further can express triggers and arbitrary functions by expressing the functions as data in the EP data structure.

2. Optimization. All the operations and the reductions will be optimized in the sense that the tree structures embedded in the EP data structure are the guiding structure in implementing the

EP language. The implemented EP language system shall be demonstrated with the best performance in comparison with the traditional database applications. The performance is reached under the generic optimization strategy.

3. Case Study. This research shall show a few traditional database and data stream applications. The case study shall demonstrate that that the EP data model is not only a toy language, but also can be a real system for database design, software development, and data communication. The case study shall further demonstrate that the effort with the EP data model is minimized in software development and maintenance. The examples on data streams will also demonstrate that the triggers and arbitrary functions can be handled efficiently in the EP language.

4. Revisit of the EP semantics. The EP semantics has been studied in [53], [54]. But it was based on the previous version of the EP language, where variables, types, and sequence of statements were not explicitly introduced. Although the presentation of the EP semantics has fully covered all the mathematical material, a revisit would simplify the presentation and refine the work precisely along the latest syntax of the EP language.

References:

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

[2] A. Asperti, H. G. Mairson. "Parallel Beta Reduction is not Elementary Recursive", Information and Computation 170, page 49-80, 2001.

[3] M. P. Atkinson, L. Daynes, J. Jordan, T. Printezis, and S. Spence. "An Orthogonally Persistent Java™", In ACM SIGMOD Record, Dec, 1996.

[4] M. P. Atkinson, P. Buneman. "Types and Persistence in Database Programming Languages". ACM Computing Surveys. Vol. 19, NO. 2. June 1987.

[5] S. Babu, J. Widom. "Continuous Queris over Data Streams", ACM SIGMOD 2001.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom. "Model and Issues in Data Stream Systems". PODS 2002: 1-16.

[7] H. Barendregt. "The Lambda Calculus: Its Syntex and Semantics. North Holland, 1984.

[8] V. Breazu-Taneman, Ramesh Subrahmanyam. "Logical and Computational Aspects of Programming with Sets/Bags/Lists". LNCS 510: Proceedings of 18th International Colloquium on Automata, Languages, and Programming *(1991).*

[9] V. Breazu-Taneman, P. Buneman, L.Wong. "Naturally Embedded Query Languages". In Proceedings 4th ICDT. Lecture Notes in Computer Science, Springer Verlag, 1992.

[10] V Breazu-Tannen, P. Buneman, S. Naqvi. "Structural Recursion as a Query Lnaguage". In Proceedings DBPL2, page. 9 – 19. Morgan-Kaufmann, 1992.

[11] P. Buneman, M Fernandez, D. Suciu. "UnQL, A Query Language and Algebra for Semistgructured Data Base on Structural Recursion", VLDB Journal: Very Large Database, Volume 9, Number 1, page 76 – 110, 2000.

[12] P. Buneman, Wenfei Fan, Scott Weinstein. "Path Constraints in Semistructure Database.

[13] P. Buneman, W. Fan, S. Weinstein. "Interaction Between Path and Type Constraints". To appear in ACM Transactions on Computational Logic(TOCL).

[14] L. Cardelli, P. Wegner. "On Understanding Types, Data Abstraction, and Polymorphism", Computing Surveys, Vol. 17, No. 4, December 1986.

[15] D. Carney, etc., Monitoring Streams – A New Class of Data Management Applications. Proceedings of the 28th VLDB Conference, Hong Kong, China 2002.

[16] S. Davison, Y. Chen, Y. Zheng. "Indexing Keys in Hierarchical Data". Technical Report 2001, University of Pennsylvania.

[17] G. Dobbie, W. Xiaoying, T. W. Ling, M. L. Lee. "ORA-SS: An Object-Relationship-Attribute Model for Semi-Structured Data". IIWAS2001.

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

[19] M. Fernandez, D. Florescu, A. Levy, D. Suciu. "Declarative Specification of Web Sites with Strudel". VLDB Journal 9 (1): 38-55 (2000).

[20] M. Fernandez,W. C.Tan, D. Suciu. "SilkRoute: Trading between Relations and XML. November 22, 1999.

[21] R. Goldman, J. McHugh, J. Widom. "From Semistrucured Data to XML: Migrating the Lore Data Model and Query Language". In Proceedings of the 2nd International Workshop on the Web and database (WebDB' 99), Page 25 – 30, 1999.

[22] 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.

[23] G. Hillebrand and P. Kanellakis. On the Expressive Power of Simply Typed and Let-Polymorphic Lambda Calculi. Eleventh IEEE Symposium on Logic in Computer Science, New Brunswick 1996.

[24] G. Hillebrand, P. Kanellakis and H. Mairson. Database Query Languages Embedded in the Typed Lambda Calculus. Information and Computation 127(2):117-144, 1996.

[25] G. Hillebrand, P. C. Kanellakis, "Functional Database Query Languages as Typed Lambda Calcluli of Fixed Order", ACM SIGMOD/PODS 94.

[26] H. V. Jagadish, Laks V. S. Lakshmanan, D. Srivastava, K. Thompson. "TAX: A Tree Algebra for XML".

[27] A. L. Hosking, J. Chen. "PM3: An Orthogonally Persistent Systems Programming Language – Design, Implementation, Performance". Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

[28] M. Jordan and M. Atkinson. "Orthogonal Persistence for the Java™ Platform: Specification and Rational". SMLI TR-2000-94, Sun Microsystems, December 2000.

[29] 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.

[30] C. Laasch, M. H. Scholl. "A Functional Object Database Language", Proceedings of DBPL4, 1993.

[31] T. W. Ling, M. L. Lee, G. Dobbie, "Applications of ORA-SS: An Object-Relationship-Attribute Data Model for Semistructured Data", Third International Conference on Information Integration and Web-based Applications & Services (IIWAS 2001), September 2001, Linz, Austria.

[32] D. Maier. "Why database languages are a bad idea". In F. Bancilhon and O. P. Buneman., editors. Workshop on database programming languages, Addison – Wesley, 1989.

[33] T. H. Merrett, H. Shang. "Unifying Programming Languages and Databases: Scoping, Metadata, and Process Communication", Proceedings of DBPL2, 1992, page 139-148.

[34] Nigel McFarland. "Instant JavaScript". Wrox Press Ltd. July 2000.

[35] A. Ohori, P. Buneman, V. Breazu-Tannen. "Database Programming in Machiavelli – a polymorphic language with static type inference. In ACM SIGMOD, 1989, page 46 – 57.

[36] M.T. Ozsu. "Principles of Distributed Database Systems, second edition", Prentice Hall, 1999.

[37] Jens Palsberg, Tian Zhao, "Efficient and Flexible Matching of Recursive Types", Information and Computation 171, 1 – 24, 2001.

[38] Y. Papakonstaintinou, H. Garcia-Molina, and J. Widom. Object Exchange across Heterogeneous Information Sources. In Proceedings of the Eleventh International Conference on Data Engineering, page 251 –260, Taipei, Taiwan, March 1995.

[39] Paton, N.W. Paton, O. Diaz. "Active Database Systems". ACM Computing Surveys, Vol 31, No 1, 63-103, 1999.

[40] P.Revesz. "Introduction to Constraint Databases". Springer-Verlag, 2002.

[41] P. Revesz, "Constraint Databases: A Survey", In Semantics in Databases, L. Libkin and B.Thalheim, Eds., Springer-Verlay, LNCS 1358, Page 209 – 246, 1998.

[42] S. Rozen, D. Shasha. "Rationale and Design of BULK", Proceedings of DBPL3, 1992, page 71-85.

[43] 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.

[44] J. Shanmugasundaram, K. Tufte, G. He, C.Zhang, D. DeWitt, J. Naughton, " Relational Databases for Quering XML Documents: Limitations and Opportunities", Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

[45] 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.

[46] C. Small, A. Poulovassilis. "An Overview of PFL", Proceeding of DBPL3, 1992, page 96-110.

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

[48] D. Stemple, T. Sheard. "A Recursive Base for Database Programming Primitives", Proceedings of First International East/West Database Workshop, Kiev, USSR, October 1990, page 311-332.

[49] M. Stonebroker, "Position Paper on Monitoring Applications", NSF Workshop on Context-Aware Mobile Database Management (CAMM), January 2002.

[50] D. 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.

[51] K. H. Xu, B. Bhargava. "A Functional Approach for Advanced Database Applications". Third International Conference on Information Integration and Web-based Applications & Services (IIWAS 2001), September 2001, Linz, Austria.

[52] K. H. Xu. "A Data Model for Effectively Computable Functions". PhD Workshop in 7th International Conference on extending Database Technology, March 27-31, 2000, Konstanz, Germany.

[53] K. H. Xu. "A λ-Calculus and its Database Applications". Manuscript unpublished, December 1999.

[54] K. H. Xu. “EP Data Model, a Language for Higher-Order Functions”. Manuscript unpublished, March 1999.

[55] K. H. Xu and B. Bhargava, “An Introductioin to Enterprise-Participant Data Model”, Seventh International Worshop on Database and Expert Systems Applications, September, 1996, Zurich, Switzerland, page 410 – 417.

[56] Report of the CODASYL Data Base Task Group, ACM, April 1971.

[57] ITU-T Recommendation X.500 (1993). Information Technology – Open Systems Interconnection – The Directory: Overviews of Concepts, Models, and Services.

Appendix A. Latest Version of EP Grammar

```
input:
      | input line
;

line:   ';'
      | term ';'
      | bin_exp ';'
      | ‘typeof’ term ':' term ';'
;

term: '(' term ')'
      | atom_term
      | application
      | abstract
      | error
;

atom_term: IDENT
      | INTEGER
      | variable
;

variable: '$' IDENT range
;
```

```
range:
      | ':' '[' bool_exp ']'
;

application: term atom_term
      | term '(' term ')'
      | term  '(' bin_exp ')'
      | '(' bin_exp ')'
      | uni_optr term
      | 'select' term_list where_clause
;

abstract: 'fun' variable '.' term
      | '{' set_terms '}'
      | update_optr term_constraint assign_value where_clause
;

uni_optr: next | prev | Anext | Bnext | Aprev | Bprev | Aptr | Bptr
      | Afirst | Bfirst
            /* navigational operators, equivalent to the ordering operators <=a, <=A, etc.. */
;

update_optr: 'create' |'update' | 'delete'
;

bin_exp: bool_exp
      | num_exp
      | bin_assign
;

term_list: term
      | term_list ',' term
;

bin_assign: term ':=' term

;
```

```
assign_value :
      | ‘:=’ term_list
;

where_clause :
      | WHERE bool_exp
;



term_constraint : term
      | term ':' term
;


num_exp: term
      | num_exp num_optr num_exp
;

num_optr: ‘+’ | ‘*’ | ‘/’ | ‘%’
;

set_terms :
      | set_element
      | set_terms ',' set_element
;

set_element: term
      | ‘{‘ term_constraint ',' term ‘}’ option
;

option :    | ‘+’ /* only used in schema, 1 or more instance is
allowed*/
            |’*’  /* only used in schema,0 or more is allowed */
                  /* more operators are needed in the future */
;

bool_exp:  atom_bool
```

```
      |  '(' bool_exp  ')'
      |  bool_exp  'and' bool_exp
      |  bool_exp  'or'  bool_exp
      |  'not' bool_exp
;


atom_bool: term  bool_optr term
;

bool_optr: 'amem' |
            'aSubetOf' |
            '<' | '>' | '==' | '<>' | '<=" | '>=' |
            '<A'| '>A' | '<=A' | '>=A' |
            '<B'| '>B' | '<=B' | '>=B' |
            '<a'| '>a' | '<=a' | '>=a' |
            '<b'| '>b' | '<=b' | '>=b' |
;
```

Appendix B. Examples in EP language

The examples are based on the latest version of the EP language system.

```
create a b = 4; /* create nodes a and a b; assign a b = 4 */
create  c = a;  /* create node c */

a b ;
=> 4 /* result */

c b ;
=> 4 /* a and c are equal */

create A B = 10; /* create A and A B, and assign A B = 10 */
create V $n $m = $m $n; /* create node V, V $n, V $n $m, assign
                           V $n $m = $m $n, since there is no
ranges
                           specified for $n and $m, the both
                           variables are un-typed */
```

```
V B A; /* B matches $n; and A matches $m */
=> 10

create AA $n $m = certain;
create AA BB = uncertain;
create AA CC DD = fuzzy;

AA BB ; /* EP matches an extensive definition first */
=> uncertain;
AA CC DD; /* EP matches the extensive definition first */
=> fuzzy;
AA 8 9; /* variables are considered last after there is
            no match with extensive definitions */
=> certain;

typedef complex = {<c, {<d, integer>}>}; /* define a type "complex" */
create A:complex ={<c, {<d, 123>}>}; /* define an instance of type
complex */
create C:complex ={<c, {<d, 321>}>};  /* define an instance of type
complex */

create TT $n : [$n amem complex] = (($n c d) * 3);
                              /*TT $n has the type complex */
create UU $n : [$n amem complex c d] = ($n * 2);
                              /* UU $n has the type (complex c d) */

TT A;
=> 369

UU (A c d);
=> 246
```

The next example is to simulate a loop in imperative languages. When the variable $j < 10, print the $j, increate the count value by 1, and recursively call loop ($j + 1) until the variable $j (or named $k, as EP is implemented) == 10. Note that a leaf node can be assigned as the value a sequence of statements. Allowing a sequence of statements in EP language fully simulates the

function/procedural definitions in imperative languages. Though it causes side-effects that is against the functional programming language principal, say losing the lazy evaluation possibility and reducing the degree of parallel computing, but it does add the welcomed, efficient flexibility the imperative languages have.

```
create count = 0;
create loop $j : [$j < 10] = $j,
                              update count = count + 1,
                              loop ($j + 1);
create loop $k : [$k >= 10] = END;

loop 4;

4, 5, 6, 7, 8, 9, END

count;

=> 6;
```

The following example simulates a trigger: when the temperature is exceeding 90 degree, then trigger air conditioner.

```
create action $n: [$n < 90] = doing nothing;
create action $m: [$m >= 90] = TriggerAirConditioner $n;
                        /* TriggerAirConditioner is a pre-defined */

create temperature = 0;
create monitorTemperature $input = (update temperature = $input),
                                    action temperature;

monitorTemperature 60;
=> doing nothing

temperature;
=> 60

monitorTemperature 99;
=> TriggerAirConditioner 99  /*the pre-defined function is called */

temperature;
```

```
=> 99
```

In this following example, the schema (`typedef`) is given for the school administration in the Figure 5 of Appendix D. Both schema definition and instance data are stored together. The schemas (types) are viewed as (identity) functions.

```
create SSD;
typedef SSD human = {<name, string>, < birth, integer>};
create SSD John : SSD human = {< name, John Smith>, < birth, 060476>};
create SSD Mike : SSD human = {< name, Mike Lee>, < birth, 060466>};
create SSD Dave : SSD human = {< name, Dave Feng>, < birth, 120570>};
typedef classGrade = {A, B, C, D, E, F};
typedef college = {< principle, SSD human> ,
               < admin, {< SSD human, { < Major, college dept>,
                               < Enroll, integer>,
                               < RegNum, integer>
                              }
                        > *,
                       < chair, string>
                       }
               >,
               < dept, {< Head, SSD human>,
                     < class, {< college admin (SSD human),
                              {< grade , classGrade>}
                             > *
                            }
                     >*
                    }
               > *};

create Purdue: college =
            {< principle, SSD John>,
             < admin, {< SSD John, {< Major, Purdue CS>,
                              < Enroll, 090300>,
                              < RegNum, 123455>
                             }
                     >,
```

```
                < SSD Dave, {< Major, Purdue Math>,
                            < Enroll, 0008>,
                            < RegNum, 8988>
                           }
                >
               }
        >,
        < CS: dept, {< Head, SSD Mike>,
                  < CS101:college dept class,
                    {< Purdue admin (SSD John), A >,
                     < Purdue admin (SSD Dave), B>}>
                 }
        >,
        < MATH:dept, {< Head, SSD Mike>,
                  < MATH1: class, {<Purdue admin (SSD John), B >}>
                 }
        >
       };
```

A few SQL-like query expressions.

Find all the information about the Department of Computer Science at Purdue University:

```
select $x where $x <=A Purdue CS;
```

The output is: `Purdue, Purdue CS, Purdue CS Head, Purdue CS101, Purdue CS101 (Purdue Admin (SDD John)), Purdue CS CS101 (Purdue Admin (SDD John)) Grade.`

Find all the students who registered at SSD (Social Security Department), and obtained the grade A in CS101; and print their names, SSNs, and Majors.

```
select $student name, $student SSN, Purdue Admin $student Major
where $student <=A SSD and Purdue CS CS101 (Purdue Admin $student)
Grade == A;
```

Appendix C. Partial Orderings

In the expression:

```
SQ 2 <A SQ.
```

It will be true if `SQ 2` is not reduced by the reduction rules. But it would be false if `SQ 2` was evaluated to `4`. Are the db-reduction rules Church-Rosser by adding the partial orderings like <A and <B as constant functions?

As intended in [54], we are not expecting any reduction on the operands of the partial orderings. The reason is that the equation $=_{EP}$ with the db-reductions including the β-reduction is no longer valid under the syntactical oriented operators (partial orderings). When we evaluate `SQ 2 <A SQ`, we are not allowed to reduce the expression `SQ 2`.

Nevertheless, the syntactical oriented orderings are the functions in the function space. To express them in a lambda calculus where the CR is a dominant principal, we can view that their operands are restricted to the range of the Godel numbers of the lambda terms (Definition 6.5.6 in [7]). A Godel number uniquely identifies a syntactical λ–term.

To be more meaningful in database practice, we should further limit ranges of the operands of the partial orderings to the set of terms defined in a given EP database. For example, the expression for the mathematical functions: `C SQ SQ 2 <=A C` is true, but `C SQ Root 2 <=A C` is false as shown in Figure 1. Note that the partial orderings presented in [54] took the entire infinite set of the db terms as the range, which was not different from what we said here.

The significance of the syntactical oriented orderings in the EP database is further explored with the math functions in Fig 1. If we write the definition of the node `C SQ` textually, it is

```
C SQ = {<Root, I>, <SQ, {<2, 16>, <3, 81>}>}.
```

But the EP data model allows two alternative presentations that semantically (in the sense of db-reductions) are equivalent to the previous one. The first alternative is:

`C SQ = {<Root, I>, < SQ, SQSQ>}`, where `SQSQ` is defined in the database as:

```
SQSQ = {<2, 16>, <3, 81>}.
```

The graphical representation is in the Fig 2.

The second alternative as graphically represented in Fig 3 is:

```
C SQ = {<Root, {<2, 2>, <3, 3>}>, <SQ, <2, 16>, <3, 81>}>}.
```

We said that the three representations are semantically equivalent under the db-reductions because the following expression is reduced to the same result along with any of the 3 representations:

```
C SQ SQ 2 -> 16.
```

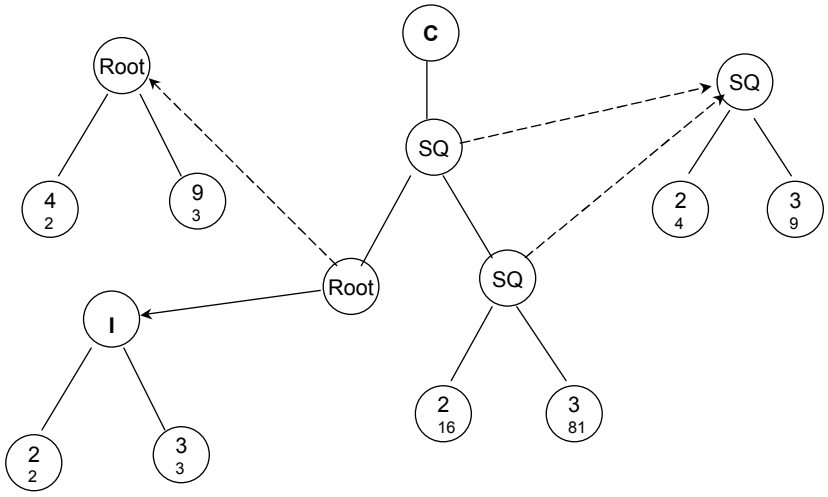

Fig 1. Original Presentation of Math Functions in [ ]

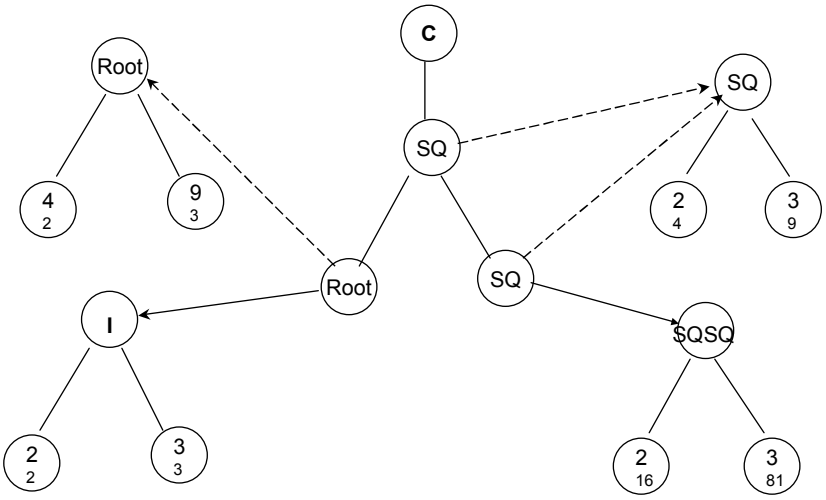

Fig 2. First Alternative Presentation of Math Functions in EP data model

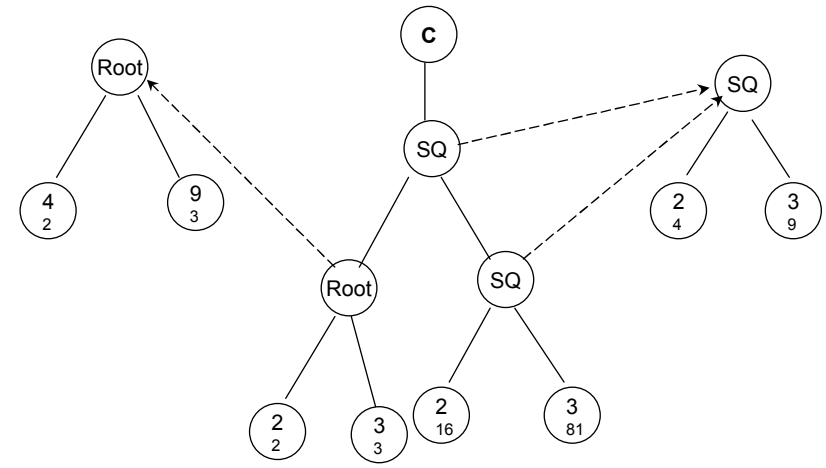

Fig 3. Second Alternative Presentation of Math Functions in EP data model

Allowing the different presentations for the same functions happens to have its usages in database applications (see more discussion in [54]):

1. Data sharing or avoidance of data redundancy. In many circumstances, an object needs to be shared by other objects. The shared object is an attribute of the others; and it has a solid arrow pointing to it. For example, `John` and `Joe` are both CS students, then their majors shall point to the `(College CS)` via solid arrows in Figure 4. If `John` is to be removed from database, then `College CS` will still retain in the database. In this case, it is not true that `College Admin (SSD John) Major <=A College CS`.

2. Data Ownership. In other circumstances, an object needs to be owned by other object. The owner is a function; and the given object is an application of the function. For example, the CS department is a part of College. When a college is to be dispended, then CS department will obviously be dispended too. This is called inclusion dependency in the relational database; and parent-child relationship in hierarchical data models. In this case, it is true that `College CS <=A College`. The EP data model has another unique way of representing data ownership. That is the argument – application relationship. For example, the argument `SSD John` has an application `College Admin (SSD John)`. Then `College Admin (SSD John) <B SSD John`.