# A FUNCTIONAL APPROACH FOR ADVANCED DATABASE APPLICATIONS

Kevin Xu
Bigravity Technologies
Bridgewater, New Jersey
kevin.xu@bigravity.com

Bharat Bhargava
Purdue University
West Lafayette, Indiana
bb@cs.purdue.edu

***Abstract:*** *Searching for better data models would continue unless a satisfaction was reached. Semi-structured data/XML, beyond the relational data model and object-oriented data models, was proposed for better productivity and quality of developing advanced database applications like Web-based applications. This paper is to introduce a functional approach - the EP data model. The EP data model is to map arbitrary finite and infinite data into functions; and to relate them with the well-know function-argument-value relationships. Function naturally incorporates programming language capability into the EP data model. The EP data model is also a comprehensive solution for data management in data manipulation, data distribution, and data interoperability.*

## 1. Introduction

The current technology used for developing database applications has been evolving and become more sophisticated in industry. Its database systems have become a great wealth in our modern life. However, this technology is still not the best we can. We feel the pain in developing and maintaining large-scaled database applications. Meanwhile advanced database applications such as in Web-based applications, biological information, and intelligent systems are looking for a better data model [1], [2], [6], and [9]. Semi-structured data/XML is the most recognized data model for Web-based database applications. The graph-oriented data models, however, can not avoid many unresolved arguments we had in earlier 1970s' about hierarchical data models and network data models.

Relation and the first order logic are the commonly accepted theoretical foundations for database management. Function that belongs to programming languages, however, has been disregarded as a guidance of database management after quite a few research activities in this area [3], [5], and [8].

The EP (Enterprise-Participant) data model takes a fresh look at data and map it to a finite set of (high-order) functions that are related by function-argument-value relationships. A set of functions

To be presented in "Third International Conference on Information Integration and Web-based Applications & Services", Linz, Austria, September 10 – 12 2001.

that are inter-related by function-argument-value relations are called a function space in [7]. The following properties can be seen from the EP data model:

1. The EP data model incorporates arbitrary (finite and infinite) data into function spaces.

2. While functions are at the center of both programming languages and the EP data model, we have the opportunity to unify the paradigms of programming languages and database management. Both data and application functions are equally treated in the EP data model. SQL-like select operations become a special case of the merged paradigm.

3. An EP database can be viewed as a restricted graph, in which tree structure is embedded. With the embedded tree structure, we can be much effective in data manipulation and data distribution.

4. The lambda calculus, the base of the EP language, provides the interoperability and allows rule reductions across the arbitrary data represented in EP databases.

The paper [13] presented the embryo of the EP data model. The papers [12], [11], and [10] found the connection of the EP data model with function spaces; and fully expanded the EP data model under the guidelines of the lambda calculus. In this paper, we guide readers through the EP data model by showing an implemented version of the EP language in conjunction with intuitive examples. This may not give readers a precise and complete picture of the EP data model, but shows the general idea behind it.

We first show EP database – logical and physical data structure for arbitrary data in Section 2. Data here and through the rest of document is traditional data or application functions. Then we walk readers through the EP language by its grammar in Section 3. The EP language is to construct data, to maintain data, to name data, to query, and to induce data. The EP language is based on the lambda calculus. But it also incorporates many useful features in imperative and logical programming languages. Section 4 maps function-argument-value relationships to a set of ordering relations. The ordering relations act as binary operators in EP language. Among the ordering relations, a set of tree-structured relations will play the crucial role in data manipulation and distribution.

## 2. EP Database

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. Given a collection of (higher-order) functions, a *function space* (structure) are formed among the functions by ordering them with the relationships of function, argument, and image.

An EP *database* (or simply database in the rest of the paper) is a restricted graph representing a collection of functions (a function space). To show the restricted graphs, we give an example in Figure 1 that demonstrates the function space of a few arithmetic functions. *SQ* symbolizes a

square function $\lambda n$:[$n \in \{2, 3\}$]. $n^2$; *Root* a square root function $\lambda n$:[$n \in \{4, 9\}$].$\sqrt{n}$; *I* an identity function $\lambda n$:[$n \in \{2, 3\}$].$n$; and *C* a composite function $\lambda f$:[$f \in \{SQ\}$],$g$:[$g \in \{SQ, Root\}$],$n$:[$n \in \{2, 3\}$],$f(g(n))$ .

In this graphical presentation, each *node* represents a function. For example, the node *SQ* is the square function with the definition of *SQ* = {<*2*, *4*>, <*3*, *9*>}. The node '*2*' (with larger font) under *SQ*, named as *SQ 2*, represents the integer *4* itself (with smaller fond), which is equal to the application *SQ 2*. Then a solid *up-down link* represents a function-application relationship between the up node as the function and the down node as the application.

A *dashed arrow* represents an argument-value relationship between the head node as the argument and the tail node as the result. For example, *SQ* is the only argument of *C* with the result of *C SQ* = {<*Root, I*>, <*SQ*, {<*2, 16*>, <*3, 81*>}>}. In this case, "SQ" is redundantly spelled out in the node under the node C while the dashed arrow clearly indicates the relationship.

A *solid arrow* represents an application-value relationship between the tail node as the application and the head node as the value. In this case, the value is not spelled out in the node. For example, *C SQ Root* is defined to be equal to the value *I*.

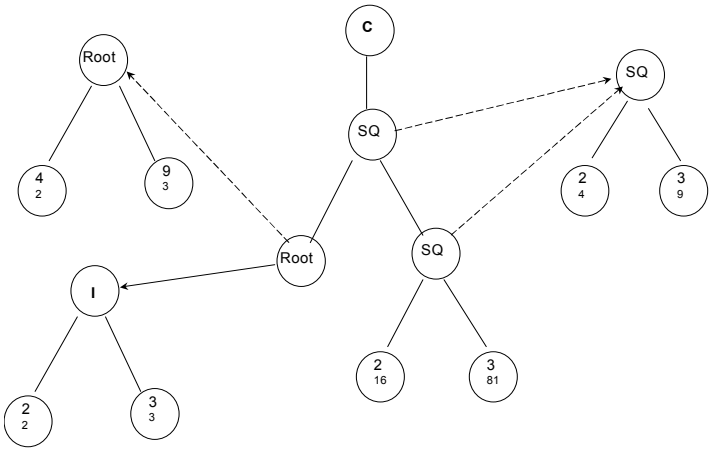


**Fig.1. EP Database for Arithmetic Functions**

Sticking with the structure of function space, we try to map application data into EP databases. Figure 2 gives a school administrative data in the EP database. Before the school 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 *CS100*, in which John studies with grade '*F'*. Like nodes in Figure 1, each node in this example represents a function. And the functions have relationships among them indicated by lines and arrows. John is a resident as defined under *SSD John*. John is a student in a College as given under *College Admin*, and he takes a class *CS100*. Note that the two occurrences of *John* under *College Admin* and *College CS CS100* are redundant just for ease to read. The relationships are clear with dashed arrows.

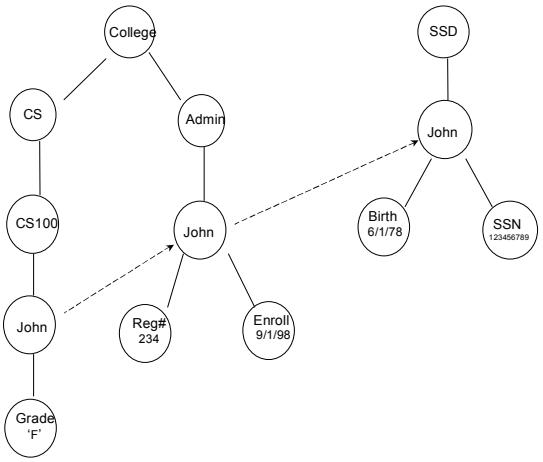


**Fig. 2.EP Database for a School Administration**

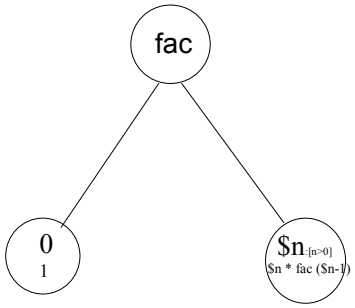


**Fig. 3. EP Database for a Factorial Fuction**

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 like integers.

To show how the EP data model represent infinite data, the figure 3 gives the EP database for the factorial function

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

In this presentation, the node *fac $n* represents a range of argument/value pairs. Given any integer $i > 0$, *fac i* will have the value $i$ * *fac* ($i - 1$) and it will eventually be reduced to an integer.

When there are two variables under the same node, the names of the variables (prefixed with '*$*') shall be distinguished to represent un-overlapped ranges.

Without exception, the EP data model incorporates cyclic data. Figures 4 and 5 show how a directed graph with a cycle is mapped to an EP database. In this presentation, for each directed link, a function-argument-value relationship is established. For the directed link *A* to *D,* as an example, the database defines: *A D* = *D*. This says that *D* is an argument of *A* and the application *A D* is equal to *D*.

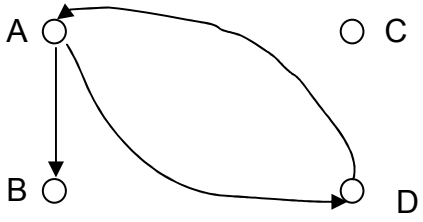


**Fig. 4. A Directed Graph**

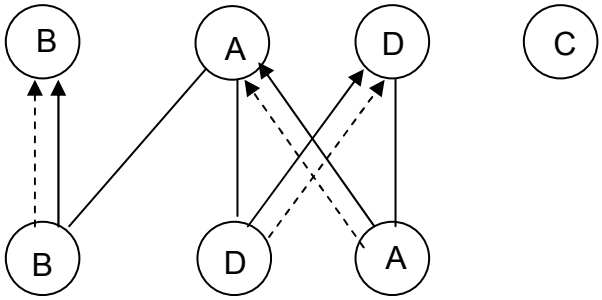


**Fig. 5. EP Database for the Directed Graph**

Similarly for the directed links *D* to *A* and *A* to *B*, the database defines: *D A* = *A* and *A B* = *B* respectively. From the presentation, we can see that the cyclic relationships among data are decomposed and represented by non-cyclic function-argument-value relationships in the EP database. We will see some interesting coincidences of functional properties with the natures of cyclic data in the later sections.

In an EP database, all the solid links form a set of *trees*; and so do dashed arrows and solid arrows. The embedded tree structures are the key to effectively manipulate and distribute data, as we will discuss more in Section 4.

The EP data model is structurally open. For example, John, as a "participant", involves with multiple activities under different "enterprises". This allows that the attributes associated with John are semantically distributed into individual enterprises. When it is time to construct a database for a company where John is working, the data portion for the company can be naturally added into the database in Figure 2. The open structure of the EP data model not only absorbs new enterprises, it also refines existing entities. For example, we may need to have more information about Computer Science in general; and the information may be shared by other universities. In this case, a new root node, named as *CS* having multiple subordinate nodes, would be drawn in the Figure 2, which acts as the argument of the node *College CS* as usual. The terminologies "Enterprise" and "Participant", from which the name "EP" was abbreviated in [13], have the correspondences of function and argument.

## 3. EP Language

A data model is just the logical (and possibly physical) view of data stored in a database. To manipulate (construct, maintain, and query) data in a database, a language must be used. The EP language is developed on the basis of (typed) lambda calculus. We introduce the EP language by showing a grammar. Through examples, we intuitively give the semantics of the language.

Much like the lambda calculus, we define a *term* (expression) as an atomic term, an application, or an abstraction. A pair of parentheses around a term syntactically groups the term together.

```
term ::= atom_term    | application    | abstraction    | '(' term ')'
```

**Atomic Term**

An atomic term is a constant, an identifier, or a variable.

```
atom_term ::= constant | identifier | variable
```

A constant is either an integer, a real number, a string, the truth value '`true`' or '`false`', or a special value '`undefined`'. An integer, a real number, and the truth values have their normal and non-dividable meanings. A string is a binary string that has its own non-dividable meaning. A string could be a multimedia file, an executable file in another language, etc. The constant '`undefined`' will be used to represent the meaningless result from a calculation. A constant is always treated as a function.

```
constant ::= integer | real | string | 'undefined' | 'true' |'false'
```

An identifier is an alphabetic-numerical string that could appear to be the name of a node in a database. The examples of identifiers are:

```
SQ, College, fac, anyId, Garbage.
```

If an identifier appears in a database, it has meaning other than itself. For example, `SQ` in Figure 1 identifies a square function. If an identifier doesn't appear in a database, it would mean no more than itself as we will see in the reduction section.

A variable is an identifier prefixed with '`$`'. For example `$n`, and `$var` are the variables.

```
variable ::= '$' identifier
```

A variable semantically represents a range of functions. It is defined when the variable is created in its database as we will see in Section 3. For example, `$n` has the range integers greater than `0` in Figure 3. A variable here is similar to a variable in the (typed) lambda calculus, $\lambda x$:[*range*].*M*, but not a variable in imperative languages. It is also like an argument in a function declaration in imperative languages. Note that the identifiers and the combinatory terms in the EP language have their counterpart of the variables in imperative languages as we will set in the coming sections.

**Application**

Similar to an application term in the lambda calculus, an application term here is a combination of two terms, or a binary expression.

```
application ::= term atom_term | term '(' term ')'
      | term  '(' bin_exp ')' | '(' bin_exp ')'
```

The examples of combinatory terms are:

```
C SQ, fac 3, fac $n, $var others,
This is a combination,
college CS CS100 (college Admin (SSD John)) Grade.
```

Semantically, an expression $E_1\ E_2\ ...E_n$ is equivalent to $(((E_1)\ E_2)\ \ldots E_n)$, but not another way around such as $E_1\ (E_2 ...\ E_n)$. Further, we treat the two terms identical syntactically, denoted as

$E_1\ E_2 \ldots E_n == (((E_1)\ E_2)\ \ldots\ E_n)$.

This grammar discourages parentheses that is not necessary and makes expressions easier to read.

With the forms of atomic terms and combinatory terms, all the nodes in a database can be uniformly *named*. For example, `SQ` represents the root node marked with `SQ` in Fig. 1. `C SQ` represents the node marked with `2` under the node `SQ`. `College CS CS101 (college Admin (SSD John)) grade` will identify the node marked with "`Grade`" at the bottom left corner with value of `F` in Figure 2. `fac $n` identifies the node `$n:[$n>0]` in Figure 3 (here the variable `$n` uniquely identifies an non-overlapped range under `fac`).

Binary expressions, e.g., (`2 + 3`), are a sugared form of combinations, e.g., `+ 2 3`, in lambda calculus. The nature form of binary expressions is preserved in the grammar. Their meanings are obvious so we don't discuss further.

```
bin_exp ::= bool_exp | num_exp
num_exp ::= term | num_exp num_optr num_exp
num_optr ::= '+' | '-' | '*' | '/'
bool_exp ::=  atom_bool  |  '(' bool_exp  ')' |  'not' bool_exp
      |  bool_exp  'and' bool_exp  |  bool_exp  'or'  bool_exp
```

```
atom_bool ::= term  bool_optr term
```

The examples of binary expressions are:

```
((SQ 2 + 1) > 3);
(SQ 2 > 3 and fac (1 + 2) < 7);
```

The discussion of the non-terminal symbol `bool_optr` will be completed in Section 4.

**Abstraction**

Abstraction terms, e.g., $\lambda x.M$ in lambda calculus, are the expressions representing higher-order functions. Correspondingly, the EP data model has nodes in databases representing higher-order functions. The nodes in databases have to be constructed and maintained by the EP language.

```
abstract ::= db_op typed_term assign_value where_clause
      | 'select' term_list where_clause
db_op ::= 'create'| 'update' | 'delete' | 'typedef'
typed_term ::= term | term ':' term | term ':' '[' bool_exp ']'
assign_value ::=  | ':=' term
where_clause ::=  | 'where' bool_exp
term_list ::= term | term_list ',' term
```

The examples of constructing and maintaining database are

```
create SSD John Birth := 6/21/1970;
create College Admin (SSD John);
create fac $n:[$n>0]:=($n*fac ($n-1));
update $x := A where
      $x <A College CS CS100 and
      $x Grade == F;
delete College Admin (SSD John);
```

The meanings of the creation operations above are straightforward. The update operation is to change the grades to `A` for all the students whose grades are `F` in `CS100`. We will discuss the binary operator `'<A'` in Section 4. The deletion operation above will delete all the nodes under `College Admin` and `College CS CS100`. The delete operations are done according to the embedded trees formed by solid links and dashed arrows as it will be seen in Section 4.

We also encourage "assignment" operations in imperative languages. For example, we can have the following operation:

```
update SQ 2 := (1 + SQ 2);
```

Since `SQ 2` in Figure 1 is `4`, then the resulting value of `SQ 2` will be `5`. The result itself is incorrect arithmetically, but it reflects the need of programming and the nature of database management. Further, the implemented version of the EP language also allows side-effect operations that are not discussed here due to the limited space. This kind of operations is discouraged in functional programming languages; but it is indeed straightforward and easy to use in practice [4].

The EP data model also supports types (schemata). For example, we could have the type definition and the corresponding instance creation:

```
typedef  UNIVERSITY;
create College: UNIVERSITY;
```

A type is treated as an identity function with a range, and can have its structure as complex as it is needed. Here we don't discuss it further.

A database is conceptually the collection of all the possible computable functions. But it is impossible and unnecessary. Instead, a database collects a finite set of interesting functions in the limited computer storage. When a database changes from a state to another, the database acts like a sliding window that drops off some functions and picks up new functions. In this process, the same name of a node may represent different functions.

**Application Functions**

Construction operations not only construct data, they also construct application functions that are normally built up in imperative language like Java or C/C++ on the top of DBMSs. Application functions are data; and data is application functions in the EP data model. We have seen it as discussed earlier. Here we give another example to emphasize this point. To build a Web application for the school administration in Figure 2, for example, we would have to create a new node in Figure 2:

```
create get_grade $person $course := College CS $course (College
     Admin (SSD $person)) Grade;
```

This function would take a person name and a CS course number entered via a web page, and calculate the grade for a student. The next sub-section shows how the calculation is done by rule reductions.

The ability of unifying the constructions for both data and functions under the same mechanism is an opportunity to dramatically cut cost and improve quality for database application developments by reducing and/or eliminating application software on the top of traditional database management systems.

In the EP data model, the SQL-like select expresses are a special set of application functions. We use select expressions when we need to retrieve a set of elements from database. As an example, the following expression is to retrieve all the pairs of vertices in Figure 5, between each of which there is a path from the second vertex to the first vertex. The boolean operator `<=a` will be discussed in Section 4.

```
select $x, $y where $x <=a $y;
```

The EP data model is able to define infinite data in a database. However, the select expressions are best applied to finite data in databases. More examples of selection expressions are given along the discussion in the rest of paper.

**Reduction**

Reduction is the piece in the EP language that reduces or eliminates manual procedural process of programming; and makes programmers to specify more on what, less on how.

Like the reductions in lambda calculus, any expression in the EP language that are supposed to terminates will be reduced to either a node in the database, a constant value or undefined. The result (*normal form*) of the derivation is unique.

First of all, we want the name of each node in a database to be reduced to its value if it has a value. For the expressions in Figure 1,

```
SQ 2 => 4;
C SQ SQ 3 => 81.
```

A non-leaf node in a database is its own normal form. For example, the expressions in Figure 1,

```
SQ => SQ; C SQ => C SQ.
```

A node without value acts like a non-leaf node. For example in Figures 5 and 2,

```
C => C; CS => CS; Grade => Grade;
```

Further, the rest of the expressions can be derived into their values even they are not the names of nodes in database. The examples in Figure 1 and 5 are:

```
Root(SQ 2) => Root (4) => 2;
C SQ Root 3 == (C SQ Root) 3 => I 3 => 3;
A D A B == (((A D) A) B) => ((D A) B) => A B => B.
```

The β-reduction rule in the lambda calculus, which expands reductions to infinite data, is simulated into the EP reduction rules.

```
fac 4  => ($n * fac ($n -1)) [$n:=4] => 4 * (fac 3)  => … => 24.
```

In the previous section, we created a new node `get_grade` in Figure 2 for a hypothetical Web-based application. Here are the reduction steps if a user enters `John` and `CS100` as student name and class number respectively via the web page:

```
get_grade John CS100
  => (College CS $course (College Admin (SSD $person)))
     [$person:=John][$course:=CS100]
  => College CS CS101 (College Admin(SSD John))  => F.
```

Applying any term to a constant is reduced to `undefined`. For example,

```
3 any => undefined;
```

Reductions in the EP language are dependent on the state of the databases. Many expressions may have to be reduced to `undefined` according to a given database. For example in Figure 1 and Figure 2,

```
SQ 5 => undefined;
College Math => undefined.
```

If the data in Figure 1 and Figure 3 is in a single database, then we will have

```
fac (SQ 2) => fac 4 => … => 24.
```

Otherwise,

```
fac (SQ 2) => undefined.
```

An identifier not defined in databases has no more meanings than itself.

```
garbage any => undefined;
any garbage => undefined.
```

## 4. Ordering Relations

Since the EP databases are based on function spaces, we can fully take advantages of the function-argument-value relationships among functions. The focus in this section is the new binary operators. The operator '>', '>=', and many other operators not shown here have their normal meanings, such as `3 > 2; SQ 2 >= 4.`

```
bool_optr ::= '>' | '>='
      | '>A' | '>=A' | '>B'| '>=B'
      | '>a' | '>=a' |'>b' | '>=b'
```

**Tree-Structured Ordering**

In section 2, we used an up-down link to represent a function-application relationship between two nodes in a database. Alternatively speaking, this relationship is a binary relation, denoted as `<A`. For example:

```
SQ 2 <A SQ; College CS CS100 <A College CS; fac 0 <A fac; A B <A A.
```

We also stated in Section 2 that all the up-down links in a database form a set of trees. In other words, a node in a database has a unique path to its root node along a sequence of links. The relationship between two nodes along the path then can be said as the function of the function of … of application. For example, `College` is the function of the function of the application `College CS CS100`. Precisely speaking, this relationship is a binary relation, denoted as `<=A`. For example:

```
C SQ SQ <=A C;
College Admin (SSD John) <=A College.
```

Analog to up-down solid links, dashed arrows introduce the other two binary operators `<B` and `<=B`. For example:

```
C SQ <B SQ;
College CS CS100 (College Admin (SSD John)) <=B SSD John.
```

The first expression says that `SQ` is the argument of the application `C SQ`; and the second says that `SSD John` is the argument of the argument of the application `College CS CS100 (College Admin (SSD John))`.

With these operators, we can give a query like "print out all the information about the CS department of College in Figure 2":

```
select $x where $x <=A College CS.
```

This query will retrieve all the down-stream nodes having solid upper-down links up to `College CS`. The query "retrieve the information about John in `College` in Figure 2",

```
select $y where $y <=B College Admin (SSD John).
```

This query will retrieve all the down-stream nodes having dashed arrows eventually terminating at the node `College Admin (SSD John)`.

The embedded tree-structured relations `<=A` and `<=B` will be crucial to effectively distribute data. The flat data structure over tables is the obstacle for the relational model to retain its position in distributed database environments. Tree, on the other hand, is the well-known structure for data distribution. For example, the files in a computer are organized in tree. With no alternative in a

distributed database system for large volume of data, a tree-oriented data model like XML on the top of relational databases is popularly utilized as the common data model in the current industry. Without exception, the EP data model by itself has the opportunity to resolve the issues of data distribution by utilizing the embedded tree structures. When the EP data model acts both as the data model for storing data and as the common data model in distributed database environments, the cost for data conversion between different data models is no longer needed.

When it is said that tree structures benefit data distribution, it implies that the system performance of data manipulation is a beneficiary. In the implementation of the EP language, the tree structures are physically built in storing data. The system performance of manipulating data is optimized. For example, the complexity is at O($lg(n)$) for searching any node in a EP database; and at O($n\ lg(n)$) for finding a path in a directed graph as discussed later.

**Pre-Ordering**

In the example `SQ 2 <A SQ`, `SQ 2` is an application of `SQ`; and `SQ 2 => 4`. Then what is the relationship of `4` with `SQ`? We say that `4` is a value of an application of the function `SQ`. In corresponding to `<A`, we denote this relationship by `<a`. Similarly, since `College CS CS100 (College Admin (SSD John)) Grade <=A College,` and `College CS CS100 (College Admin (SSD John)) Grade => F,` then we say that F is a value of value of… of the function `College`, denoted as

```
F <=a College.
```

By applying the binary operator `<=a` and the reduction rules to the directed graph in Figure 5, we see a few interesting characteristic coincidences of cyclic data with functions. By the reduction rules, first of all, we have already shown in Section 3: `A D A B => B.` Similarly, a term representing a path traveling along a directed cycle as many time as needed will be reduced to the last vertex of the path:

```
A D A D … A D == ((… (((A D) A) D)… A) D)
=> ((…(((D) A) D)…A) D)
=> ((…(A D)…A) D)
=> …=> D.
```

Secondly, the binary operator `<=a` can be used to express whether or not two vertices are connected by a directed path. For example, "is there a path from vertex `D` to vertex `B` in Figure 4?" can be expressed as

```
B <=a D.
```

Since `D A B <=A D,` and `D A B == (D A) B => A B => B`; then `B <=a D` will be reduced to true in the implementation of the EP language. Similarly, the query "is there a circle passing the vertices `A` and `D` in figure 4?" is expressed as

```
A <=a D and D <=a A.
```

## 5. Summary

An EP database is to store arbitrary finite and/or infinite data in a uniform data structure. When a data schema, a data instance, and an application function are to be built, they are written in the EP language, and stored physically together.

An EP database is a finite set of nodes. Nodes represent functions; and inter-connected by function-argument-value relationships.

The EP language is based on the lambda calculus. A node is named by a term in the EP language. An arbitrary term that terminates has a unique normal form. The EP language applies its reduction rules consistently against databases storing schemas, instances, and application functions uniformly. The EP language also adopts useful features from imperative and logical programming languages.

Like database management systems, the EP language allows explicit data updates. A data update in the EP language is understood as a function of changing a database from a state to another. A database is viewed as a sliding window in the front of the entire function space. With a data update, a database drops off some functions and adds other functions.

The ease of programming is not only the benefit of the EP language. The EP data model is also able to deals with data distribution and interoperability. The tree structure embedded in the EP databases is the gift for data distribution and better system performance. Since the EP language is based on the general and powerful form of lambda-expressions, arbitrary data or application functions can be referenced and exchanged in the uniform way across distributed computing environments.

**Acknowledgement:** The authors thank Gerald Baumgartner, Alex Borgida, Jens Palsberg, Jari Veijalainen, Aidong Zhang and Guo-Qiang Zhang for their comments and encouragement; and anonymous reviewers for their criticisms.

**Reference:**

[1] P. Buneman, M. Fernandez, D. Suciu. "UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion". The VLDB Journal, 2000.

[2] 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.

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

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

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

[6] A. Sanfilippo. "Lexicons for Constraint-based Grammars". In Ronald Cole, editor, "Survey of the State of the Art in Human Language Technology": page 102-105, Cambridge University Press, 1997.

[7] 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.

[8] 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.

[9] D. Suciu. "Web Data and the Resurrection of Database Theory". 7th International Conference on Database Systems for Advanced Applications, April 2001, Hong Kong.

[10] 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.

[11] K. H. Xu. “A λ-Calculus and its Database Applications”. Manuscript unpublished, December 1999.

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

[13] 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.