

A Data Model for Effectively Computable Functions

Kevin Houzhi Xu
Department of Computer Science
Purdue University

Searching for a better data model would continue unless a satisfaction was reached. Improving the productivity and quality of developing traditional and emerging database applications is the motivation driving the research activities. This abstract is to introduce the approach of a data model - EP data model. Its data structure is able to store as a finite set of nodes arbitrary effectively computable functions. Its queries are arbitrary computable functions. Both the data structure and the queries of the EP data model are unified under an extended lambda calculus, which provides a uniform manner to access data (either finite or infinite).

Expressiveness and Descriptiveness 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.

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 latter cannot. 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 latter only recursive functions. 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. However, when one would like to view data 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 [4].

What is descriptiveness? There is not an absolutely definition of it. But the common sense was that the closer to machines a language references to, the less descriptive the language is; or that the easier in constructing functions, the more descriptive the language is. 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.

It is unfortunate that traditional data models are usually applied to some database applications beyond the scopes of what they are supposed to do. These database applications can be traditional ones and can be any computing application with the mixture of discrete finite data and infinite data. 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; constraints and triggers of relational database management systems for functional dependency; 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.

As a matter of fact, the most typical example of remedying the shortage of traditional data models is managing application data with ordering relations such as organization hierarchies, family trees, and network-oriented graphs in database application practices. 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]. To maintain the consistence of the “flattened” ordering data with the dynamic world, a remedy is required from imperative programming languages to support update operations. The notion of composite objects with hierarchical structures is useful in certain applications [5]. But it is indeed a remedy because it is the concept beyond the generic network-oriented data structures of object-oriented data models.

To eliminate the remedies of traditional data models, a fully descriptive data model must have to do with computability. It would always be handicapped if a data model can not fully manipulate the computer’s capacity as we mentioned above in data presentation, queries, and data interoperability. Many researches after relational and object-oriented models have been done, for example, semistructured data [2], graph-oriented object data model [3], and EP (enterprise-participant) data model [8]. However, they have too little to do with computability. A data model would not be fully expressive and descriptive unless it claims a semantics equivalent to the effectively computable functions.

The research in [6] and [7] has extended the EP data model by relating it with λ -models and the λ -calculus. The extended EP data model has its data structure and its query language which store and express arbitrary effectively computable functions. Both the data structure and the query language are unified under an extended λ -calculus. Therefore, the EP data model, unlike other data models or programming languages, possesses both the full expressiveness and the full descriptiveness.

This article outlines the approach of the EP data model. The detailed and rigorous formalism including the language definition, reduction, semantics, consistence, soundness, and Church-Rosser Theorem were provided in [6] and [7].

Functions and Data Structure A relation f is a function iff (i) the members of f are ordered pairs; and (ii) if $\langle x, y \rangle$ and $\langle x, z \rangle$ are members of f , then $y = z$. While the semantics of functions has been

precisely provided by mathematicians, the syntactical forms expressing functions can be various. Extensional definition enumerates all the argument/value pairs for functions; and the lambda calculus uses lambda terms to represent functions and reduction rules to calculate results (normal form). The data structure of the EP data model is a combination of extensional and intensional definitions of functions. With the combination, we can explicitly enumerate finite discrete objects if necessary, and store and manage infinite data in intensional definitions.

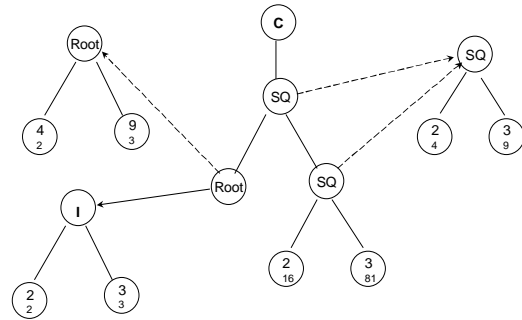


Fig. 1. Extensional Definition of Functions in EP data model

Figure 1 gives a function space among a few integers and four higher-order functions: a square function $\lambda n. n^2$, symbolized as SQ ; a square root function $\lambda n. \sqrt{n}$, symbolized as $Root$; an identity function $\lambda n. n$, symbolized as I ; and a composite function $\lambda f, g, n. f(g(n))$, symbolized 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 $Root$.

In this graphical presentation, a node represents a function. For example, the node SQ is the square function with the definition of $SQ = \{\langle 2, 4 \rangle, \langle 3, 9 \rangle\}$. The node ‘2’ (with larger font) under SQ , identified as $SQ\ 2$, represents the integer 4 itself, which is equal to the application $SQ\ 2$. Then a solid up-down link represents a function-value relationship between the up node as the function and the down node as the value, such as the relationship between SQ and $SQ\ 2$. 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 argument of C with the result of $C\ SQ = \{\langle Root, I \rangle, \langle SQ, \{\langle 2, 16 \rangle, \langle 3, 81 \rangle\} \rangle\}$. A solid arrow represents an application-value relationship between the tail node as the application and the head node as the

value. For example, $C SQ Root$ is defined to be equal to the value I .

Exactly in the same way as in Fig. 1, Figure 2 gives the functional view of a school administrative database application and the associated national social security department. 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 two occurrences of names “John” under $College$ are the aliases of ($SSD John$) and ($College Admin John$) respectively.

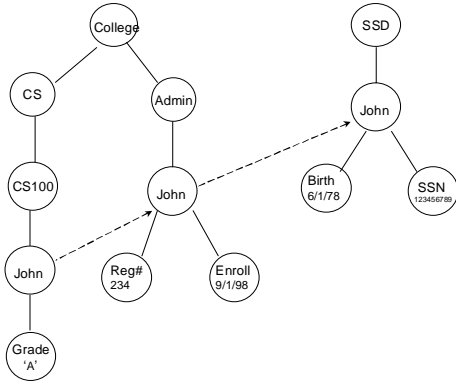


Fig. 2. A School Administrative Database in EP Data Model

We say that the EP data model is semantic or descriptive. It is not individual designers’ perspective to the EP data model, but the uniformly functional view to the application data without exception. This includes arbitrary functions with infinite data. As another example, the figure 3 gives the graphical view of the EP database for the factorial function

$$fac\ 0 = 1; fac\ n = n \times fac$$

Where n is a variable ranging from 1 to infinite.

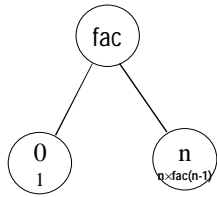


Figure 3. Factorial Function

An Extended λ -Calculus Storing arbitrary effectively computable functions in the data structure is not the only objective of the EP data model. Naming data, manipulating data, and constructing effectively computable queries against data in a uniform manner is another key to fully claim the expressiveness and the descriptiveness of the EP data

model. The pure λ -calculus is the language to be enhanced for this purpose.

Rather than the pure λ -calculus itself, its enhancement is the most interesting in this paper. To describe the discrete objects and their relationships in EP databases, we introduce two countable sets of symbols - proposition letters and constants in the enhanced λ -calculus. All the symbols such as SQ , $Grade$, $College$ in Fig. 1 and 2 are proposition letters. Different from variables in the poured λ -calculus, the proposition letters represent fixed meanings (functions) in a given EP database. And different from constants such as integers, they could dynamically change their meanings from a state to another of an EP database.

Each proposition letter or constant is a λ -term in the extended λ -calculus, and let’s call it a db -term in this paper. Like the applications in the pure λ -calculus or in the Combinatory logic, combining two db -terms M and N forms another db -term $M N$. Therefore, SQ , $SQ\ 2$ in Fig. 1, and $College$, $Grade$, and $College Admin$ ($SSD John$) in Fig. 2 are db -terms.

With the db -terms introduced, an EP database can be formally expressed as a finite set of db -terms, in which some of the db -terms have assignments. For example, the database in Fig. 1 then is specified textually as: $D = \{SQ, SQ\ 2\ (tag \equiv 4), SQ\ 3\ (tag \equiv 9), Root, Root\ 4\ (tag \equiv 2), Root\ 9\ (tag \equiv 3), I, I\ 2\ (tag \equiv 2), I\ 3\ (tag \equiv 3), C, C\ SQ, C\ SQ\ Root\ (tag \equiv I), C\ SQ\ SQ, C\ SQ\ SQ\ 2\ (tag \equiv 16), C\ SQ\ SQ\ 3\ (tag \equiv 81)\}$. From this, we can see that some db -terms serve as the names of the nodes in EP databases.

In addition to serving as a name, each db -term is a computational expression, from which a unique normal form (value, called db -nf) can be reduced. For example, all the db -terms naming non-leaf nodes in EP databases are in db -nf by themselves, such as SQ and $C SQ$ in Fig. 1 and $College$ and $College Admin$ in Fig. 2. All the db -terms naming leaf nodes are reduced to their tag values, such as $SQ\ 2 \rightarrow 4$ in Fig. 1; $College\ CS\ CS100\ John\ Grade \rightarrow 'A'$ in Fig. 2. And a proposition letter not appeared in databases is reduced to undefined automatically. Recursively, all the db -

terms, as long as they are computable, can be reduced to normal forms.

By incorporating the *db*-terms and their corresponding reduction rules into the pure λ -calculus, the extended λ -calculus unifies the finitely discrete data in *db*-terms with effectively computable functions in λ -terms. For example, *fac* and *fac 3* in Fig. 3 are *db*-terms; and *fac (SQ 2)* can be reduced to *24*.

Set-Oriented Operations Corresponding to the relational algebra of the relational data model, more powerful set oriented operations are alternatively available to the EP data model while the extended λ -calculus guarantees the possibility of constructing effectively computable queries. Since EP databases are sets by themselves, all the operations on sets like join and projection will be available to the operations against EP databases.

In addition to all the popular functions like plus +, and greater than >, a few more interesting built-in functions are uniquely available in the EP data model. They stem from the well-known function-argument-application relationships in function spaces. For example, *SQ* is the argument of the application *C SQ*, where *C* is the function in Fig. 1. Similarly, *SSD John* is the argument of *College Admin John* (recall that *John* here is the alias of *SSD John*) in Fig. 2. This relation can be recursively extended to the relationships of function-of-function; and argument-of-argument. For example, *C* is the function-of-function of *C SQ 2* in Fig. 1. By applying these relationships among the nodes in EP databases, the two types of tree structures, either along solid links or dashed arrows, are formed as illustrated in Fig. 1, 2, and 3. With these built-in operators, many fixed-point queries only expressible by Datalog in the relational data model will be expressible by a SQL-like query language in the EP data model. The examples are "print out all the information about the CS department", and "delete all the information about John from *College*" in Fig. 2.

In the function-argument-application relationships discussed above, we required exactly syntactical match of *db*-terms. For example, *Root (SQ 2)* is not an application of *Root* in Fig. 2 while *Root 4* does so. When we equalize the *db*-terms with the same normal form, another set of interesting built-in operators are developed for inducing the relationships among *db*-terms. Certain data and its queries with pre-ordering relations (reflexive and transitive, but may not antisymmetric) can be easily expressed by the data

structure and a SQL-like query language with the unique built-in operators in EP data model. An example of the pre-ordering data is directed graphs with cycles; and a query would be "Do vertexes A and Z reside on the same cycle?"

Distribution and Interoperability Flat structure over possible thousands of tables is the biggest obstacle for the relational model to retain its position in distributed database environments. The tree-structured relations existed in EP databases, however, would allow data physically distributed in hierarchies along the logic structure. This would significantly reduce the complexity of data distribution in terms of performance and development effort. Like NFS (Network File System), an EP database would be able to talk with other EP database by simply "attaching" it to the distributed database systems.

In addition, the EP data model, as a Turing-Machine equivalent language, is a sufficient language for interoperability across database applications. Further the EP data model, again due to its Turing-Machine Computability, could be a bridge for a traditional database application to talk with a non-database computing systems.

Reference:

- [1] S. Abiteboul, R. Hull, and V. Vianu. "Foundations of Databases". Addison-Wesley Publishing Company, 1995.
- [2] Peter Buneman. "Tutorial: Semistructured Data". In Proceedings of ACM Symposium on Principle of Database Systems, 1997.
- [3] 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.
- [4] Gerd G. Hillebrand, and Paris C. Kanellakis. "Functional Database Query Languages as Typed Lambda Calculi of Fixed Order". SIGMOD/PODS 1994, page 222 - 231.
- [5] 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.
- [6] K. H. Xu. "EP Data Model, a Language for Higher-Order Functions". Manuscript submitted to ACM Transactions on Database Systems, November 1999.

- [7] K. H. Xu. "A λ -calculus and its Database Applications". Manuscript submitted to LICS' 2000, December 1999.
- [8] 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.