Data Model and Total Recursive Functions

Kevin H. Xu, Jingsong Zhang, Shelby Gao, Roger R. McKeown
Bigravity Business Software LLC
2306 Johnson Circle Bridgewater, New Jersey 08807, U.S.A.
{Kevin, Jingsong, Shelby, Roger}@froglingo.com

ABSTRACT
Data models square off programming languages by preventing software applications from having exceptions and from not terminating. The more expressive a data model is, the easier the process of developing and maintaining software. When a data model is semantically equivalent to a class of total recursive functions, it, along with a programming language, achieves the greatest possible ease in software development and maintenance. The purpose of this paper is to demonstrate that the EP is such a data model.

Categories and Subject Descriptors
D.3.1 [**Formal Definition and Theory**]: Semantics, Syntax.

General Terms
Languages, Theory, Management, Design, Security.

Keywords
Data model, programming language, ordering relations, normal form, reduction, total recursive functions, applicative structure.

1. INTRODUCTION
Many database applications were written in programming languages in the 1960s and 1970s, and they are still in operation. The use of Database Management System (DBMS) came to database application software in and around the 1970s. Although a significantly improved productivity in the development and maintenance of database applications, its limited expressive power forced it (DBMS) to be used with a programming language.

Froglingo is a system consolidating the multi-component system architecture of the traditional technologies into a single component. It is a unified solution for information management, and an alternative to having to combine a programming language, DBMS, a file system, and a web server. It is a "database management system" (DBMS) that stores and queries business data; a "programming language" that supports business logic; a "file system" that stores and shares files; and a "web server" that interacts with users across networks. It does more than combine existing technologies; it is a single language that uniformly expresses both data and application logic. It is a system supporting integrated applications without using application-based data exchange components and data access control mechanism [24, 26].

Assessing a language's ease-of-use is generally considered subjective. Froglingo, however, suggests how one might assess ease-of-use more objectively. The authors in [25] made a case for this view. It started with two assumptions:

(1). A data model is easier to use than a programming language in the development and maintenance of those applications expressible in the data model.
(2). If one data model is more expressive than another data model, the former is easier than the latter in the development and maintenance of those applications where a programming language is involved.

The authors in the article concluded that ease-of-use reached the limit mathematically when the data model was semantically equivalent to a class of total recursive functions.

This article formalizes the notions presented in the paper [25]. Specifically, they are as follows:

1. There exists a language, called the EP data model, Froglingo without variables, that is semantically equivalent to a class of total recursive functions.
2. The concept of "data model" is formally defined to quantitatively tell differences between a programming language and a data model.
3. From the definition above and the two assumptions earlier, EP data model is concluded to be the easiest.

The notion that EP data model is equivalent to a class of total recursive functions stands by itself and is the main course of this article. We devote the entire article to demonstrate it except for Section 2 and the theorem 4.3. Independent from the discussion of ease-of-use of languages, the article [24] concluded that consolidating the multi-component system architecture of the current technologies into a single component in Froglingo starts from this notion.

To relate the work of Froglingo with the rest in the fields programming language and database management, we reiterate in Section 2 the ease-of-use discussed in the article [25]. Further, we carefully choose a mathematical definition for the concept of "data model" so that any (set-oriented) queries expressible by a data model are decidable. This decidability justifies the assumption that the data model is easier to use than a programming language for the software applications expressible by data model. It follows from this that the more expressive a data model is, the easier it will be in the development and maintenance of arbitrary software applications that involve a programming language. By assuming this ease-of-use, we are able to determine the position of a language system in the coordinate with the following two dimensions: expressive power and ease-of-use. Also, to stress the notion that the EP data model is the easiest, we point out that the upper limit of the ease-of-use is reached when a data model is equivalent to a class of total recursive functions.

EP data model is at the center of Froglingo. Froglingo is not the main focus of this article. For its language specification, one can reference the article [26]. For its formal theory aspects of

semantics, consistency, soundness, and completeness, one can reference the article [28].

While facilitated as sample business applications through the article, the following examples are also intended to demonstrate that the data discussed here is no longer limited to relational table and hierarchical structure, but extended to a more generic semantics: (total recursive) high-order functions.

Example 1.1: 1. The unary function $s(n) = n^2$, where $n \in \mathbb{N}$. It can be equivalently expressed by its properties: s = {<0, 0>, <1, 1>, <2, 4>, …}.

2. The 2-ary function f(x, y) = x + y, here x, y are non-zero positive integers. It can be equivalently expressed by its properties in ordered triples: f = {

<1, 1, 2>, <1, 2, 3>, …,

<2, 1, 3>, <2, 2, 4>, …,

…}

3. A database application for the Social Security department in the United States; and a central administration office in a college or university. In the social security department (SSD), each resident has his/her social security number (SSN), name, and birth date, etc. For the college or university, a student registers. The college has departments, each department offers classes, and each class has students who attend.

2. EASY OF USE

Expressive power is a well-established dimension on which to measure the quality of a computer language. Ease of use is another. This has been the main stream of development in programming language and database management.

2.1 Language

A language has its syntax and its semantics. Semantics characterizes a quality of language, i.e., the effect or outcome of the programs written in language. It has been well studied and quantitatively measurable by expressive power.

Syntax characterizes another quality of language, i.e., the effort of writing the programs in language. In this article, we call it ease-of-use. When two languages have the same expressive power, one may prefer to use one rather than another. There hasn't been a precise measurement on easiness. We raise the issue here because it has been a factor in the evolution of computer languages.

Two languages can be comparable in ease of use only if they are equivalent in terms of expressive power. It is meaningless to compare two languages in ease of use if they contain two completely different sets of semantics; however, if the semantics of one language are a superset of the semantics of a second language, the ease of use of the first language can be compared with the ease of use of the second when that first language is used to express the semantics of the second.

2.2 Programming Language

A programming language is a language with the semantics equivalent to a class of computable (or partial recursive) functions that determine the upper limit of what a computer can handle.

Programming languages have evolved from low-level machine/assembly languages to higher-level languages. One aspect of programming languages remains unchanged, though—their expressive power, the Turing equivalent.

What, then, has changed in the process of programming language evolution? What has changed is the *ease of use*. Driven by the need for programmers to be ever more productive in software development and maintenance, we have worked continuously to produce easier programming languages.

Now we note here the key difference between a programming language and a DBMS: A programming language can represent computable functions, and therefore infinite data, in expressions that are themselves finite.

The semantics of programming languages inevitably falls into a class of *partial* recursive functions. We strive hard to design and to use programming languages so that the programs written in the programming languages can eventually fall into a more narrow class—a class of *total* recursive functions. The class of total recursive functions is all those useful in representing software applications within the given limitations of a computer.

2.3 Data Model

A data model is a language. As far as the existing data models and the objective of this article are concerned, we contend that the semantics of a data model is a subset of a class of total recursive functions. A data model is the mathematical abstraction of a database management system (DBMS) for database applications.

A data model normally refers to a data structure that stores a set of data and that has a set of built-in operators under which that set is closed. (By closed, we mean that an operation always terminates and returns members from the set.) To emphasize the dominant role of data structure, we will define a data model as a data structure that stores a set of objects and that offers decidable dependencies among those objects. The second definition is intended to be equivalent to the first one, while the dependency attributes attend the built-in operators.

We define dependency as follows: If one object depends on another in a set, and that first object is in the set, then the second must be in the set as well. This dependency is bi-conditional, so conversely, if the second object is *not* in the set, then the first object cannot be there either. (E.g., an attribute in a relational table depends on its row; a child object depends on its parent object in hierarchy, and the birth of an infant depends on both its mother and father.) This dependency restriction avoids exceptions in a programming language.

Dependency must also be decidable. A data model must always be able to tell if two arbitrary objects in a set are dependent or not. This restriction disallows an object whose dependence is on itself (i.e., data that is organized to have a cyclical loop) in the managed sets. Such a situation disqualifies a computer language as a data model. There can be a program in the language that doesn't terminate on an input.

This definition should preserve the essence of the concept of a data model that DBMSs started with in the 1970s; at the same time, the definition distinguishes a data model from a programming language. The essence of a data model is that it is set-oriented and always halting in queries and update operations on finite data. In expressing a finite set of objects (usually referred

to as business data), using a data model is easier than using a programming language. The dependency is the supporting factor.

Below, we provide a formal definition for a data model. For convenience, we will specify that a data model is itself a set, rather than a language to manipulate a set. This does not mean that a data model is a set alone, but that it is a language system that maintains the set and its integrity.

The concept of relation is critical to our argument, and we start with it and its familiar notational symbols:
A (binary) relation ρ in a set A is a subset of the Cartesian product $\rho \subseteq A \times A$, i.e., a set of ordered pairs. If ρ is a relation in A, we write $<x, y> \in \rho$ and $x \rho y$ exchangeably, here $x, y \in A$. We also say that x is ρ-related to y. Given a set $A = \{1, 2, 3, a\}$, for example, we can have the following notations: the relation "less than": $< = \{<1, 2>, <1, 3>, <2, 3>\}$; $<1, 2> \in <$; $1 < 2$; 1 is less than (<-related to) 2. The characteristics of a specific relation can be named, e.g., we say that < is transitive in A, i.e., $<1, 2> \in < \cap <2, 3> \in < \Rightarrow <1, 3> \in <$.

Relations can be dependent, decidable, and tree-structured. We explain below:

Definition 2.1: Given a non-empty set A, a strict subset $B \subset A$, and $x, y \in A$, x is dependent on y in B, denoted as $x \rho y$, if and only if $x \in B$ implies that $y \in B$, i.e., $\rho = \{<x, y> \mid x, y \in A$; if $x \in B$, then $y \in B\}$.

This definition specifies a dependent relationship between two entities. It does not forbid one entity from depending upon two entities. Given a function f(x) = x + 1 and an argument 4, as another example, we say that the process of applying f to 4 and ending up with the value 5 is dependent on both the function f and the argument 4.

A dependent relation has to be transitive, but this is not biconditional: not all transitive relations are dependent.

Definition 2.2: A relation ρ in A is decidable if and only if $\forall x, y \in A$, there is an algorithm that determines if $x \rho y$ holds in finite steps.

Definition 2.3: A data model is a set containing at least one dependent and decidable relation.

This definition does not restrict the data model to only one dependent relation; instead, the more dependent relations or other relations exist, (as long as they are decidable), the easier the process of software development and maintenance will be. But one dependent relation *does* distinguish a data model from a programming language in ease of use, as the relational data model and the hierarchical data model did in the software industry.

A decidable relation will prevent an operation from not terminating. A computer language cannot qualify itself as a data model if a program in that language allows non-termination on an input. With the definition above, we say that stack and queue in a programming language are examples of data models. This is also true of the relational data model and the hierarchical data model with the containment relationship. The "network data model," as it is traditionally called, allows cyclical data, and because it does not clearly define the dependencies among this cyclical data, it is excluded from being a data model in this article. An operation on cyclical data may not terminate. Obviously, those languages with variables and while-loops (or self-reference procedures) can cause infinite loops and they too are not data models. One example is Datalog [1], but this applies to any programming language.

Our definition does not specify whether or not a data model is finite or infinite. Mathematically, we allow a data model to be an infinite and countable, even though this is not possible in practical terms. This allows us to show that there is a data model that can "store" the entire class of total recursive functions.

Before continuing, we briefly note the research effort that has been put into developing more expressive data models, the models that go beyond the relational data model, the hierarchical data model, and the network structure (called network data model at the time). Starting in the 1970s, there were many proposals toward more semantic data models, such as CODASY [17], DAPLEX [22], Entity-Relationship [10], X.500 [13], XML [12], semi-structured database [8], ORA-SS [15], and EP data model [29, 28, and 27]. While a large number of them appeared to be variations of the three well-recognized data models [7], these efforts lacked a mathematical definition of data model concept, which made an objective measurement of the proposals impossible.

2.4 Hybrid

Programming languages define functions by coding algorithms; data models define functions by enumerating properties. One might say that although a data model is preferable, a programming language is inevitable.

There are several reasons for this. First of all, a lot of business data falling into a class of total recursive functions may be desired, but not expressible in a traditional data model. (By not expressive, we mean that some dependencies would be lost even if they were placed, that is to say decomposed, into the data structure of a data model.) Hierarchical data, as a typical example, can be folded into a table, but its containment relationships cannot be captured by the relational data model. Another example would be the relationships among the vertices in a directed graph (e.g., is there a path from A to B), which cannot be captured in both relational data model and hierarchical data model.

Secondly, constructing arbitrary queries on the top of a managed data set requires a programming language. Although built-in operators can be used to construct a class of useful queries, they don't exhaust all the queries that practicality requires, and they fall into a class of total recursive functions. For example, a query in the relational data model cannot simply return a single attribute or a sequence of attributes out of a relational database. There is no exception to this. This holds true even for a unification, which will be discussed in the next section.

A system having both a programming language and a data model is called a hybrid. Hybrids started with the research efforts into "database programming language" in 1970s to the early 1990s. They offered programming languages on the top of relational data models, hierarchically data models, and network structures (called network data models at that time). Some proposals were Galileo (surveyed in [4]), Functional Object Language [14], Machiavelli [16], PFL [23], BULK [18], and XML/XQUERY [6]. This, the combined relational data model and programming language, is the most popular hybrid today. This approach, however, due to the

lower expressive power of the underlying data models [7], didn't start from a well-established foundation.

In database applications, a hybrid is easier than a stand-alone programming language. A hybrid is easier because a data model is used for a part of database application. A hybrid is easier than another hybrid if the data model of the first hybrid is semantically is a superset of the data model of the second hybrid. It is not meaningful to compare the ease of use of hybrids based on relational and hierarchical data models because their semantics overlap, and are not inclusive.

2.5 Unification

A hybrid becomes a unification when its data model is semantically equivalent to a class of total recursive functions. Mathematically, this means that the data model could represent arbitrary software applications as long as they are totally recursive, without a programming language, assuming space was unlimited.

We conclude, given the understanding of ease of use that we have provided in this article, that the easiest hybrid to use is a unification.

This brings us back to Froglingo. Froglingo is just such a unification. First, Froglingo, without a variable, is an EP data model, which we prove in Section 4. Second, an EP data model is semantically equivalent to a class of total recursive functions, which we prove in Sections 6 and 7. Finally, Froglingo is a programming language. It has variables to express infinite data in finite algorithms.

Practically, the easiest implies a consistent and terminating method for as much finite data as a business application needs. In addition, it implies the consolidations of the multi-component architecture of traditional technologies into a single component. The two components that are completely eliminated are (1) data exchange agent in data communication and (2) data access control mechanism. Froglingo handles data communication and data access control as if NFS (Network File System) handled file communication and file access control. For more discussion about the second implication, readers can reference [24].

3 EP DATABASE

In traditional data models, an entity is either dependent on one and only one other entity, or independent from the rest of the world. The functional dependency in relational data model and the child-parent relationships in Hierarchical data model are typical examples. This restriction, however, doesn't reflect the complexities of the real world that can be managed by using a computer. The logic of the EP data model is that if one entity is dependent on entities, then those entities are precisely two in number. Drawing terminology from the structure of an organization or a party in article [29], one dependent entity was called enterprise (such as organization and party), the other was called participant (such as employee and party participant), and the dependent entity was called participation. An enterprise consists of multiple participations. Determined by its enterprise and its participant, a participation yields a value, and this value is in turn another enterprise.

Definition 3.1: Let P be a set of identifiers, and C a set of constants where `null` is a special constant. The set of terms T is formed by the following rules:

1. A constant is a term, i.e., $c \in C \Rightarrow c \in T$
2. An identifier is a term, i.e., $a \in P \Rightarrow a \in T$
3. The application of a term to another is a term, i.e., $m \in T,\ n \in T \Rightarrow (m\ n) \in T$

For example, the expressions `3.14`, "a string", a_id, (f 1), ((country state) county), ((a b) (c d)) are terms.

For convenience in the discussion, we use the following notations:

Notation 3.2 1: Given an application $(m\ n) \in T$; m is called the left sub-term, and n the right sub-term.

2. The parentheses surrounding an application can be omitted when the right sub-term is not another application. For example, (f 3), ((country state) county), and ((a b) (c d) can be re-written the following way: f 3, country state and county, and a b (c d) correspondingly.
3. A term $t \in T$ has an iteration size, denoted as $\|t\|$, and the iteration size is calculated with the following formulas: if $a \in C$, or $a \in P$, then $\|a\| = 1$; otherwise $\|m\ n\| = \|m\| + \|n\|$.

Unlike Lambda Calculus, the EP database doesn't have variables. Note that Froglingo does have variables (this is not discussed here). EP carries high-level functions by using identifiers; Lambda Calculus doesn't have identifiers. Assignment is the starting point in the EP database that carries high-level functions.

Definition 3.3: Given $m, n \in T$, the form $m =: n$ is an assignment. Here, m is called the assignee; and n the assigner. All the assignments in a given T make up a set: $A = \{ m =: n \mid m \in T,\ n \in T \}$.

Notation 3.4 1. Given an expression $a \equiv b$, the symbol $\equiv$ indicates that the two symbols a and b are identical.

2. Given an expression $a \not\equiv b$, the symbol $\not\equiv$ indicates that the two symbols a and b are not identical.
3. Let $\mathbb{N}$ be the set of natural numbers $\{0, 1, 2, \ldots\}$.
4. Let $m, n_0, \ldots, n_i \in T$, here $i \in \mathbb{N}$. We write:

$$m\ n_0 \ldots n_i \equiv m\ \bar{n} \equiv (\ldots((m\ n_0)\ n_2) \ldots n_i).$$

We further write: $\bar{n} \cong n_0, \ldots, n_i \in T$; and $\|\bar{n}\| = i$.

5. Given a term $m\ \bar{n}$, m is called the left-most term of $m\ \bar{n}$, here $\|\bar{n}\| \geq 0$.

Now we can introduce the formal, logical definition of an EP database:

Definition 3.5 An EP database D is the union of a set of terms $T \subset T$ and a set of assignments $A \subset A$, i.e., $D = T \cup A$, such that the following are true:

1. If an application $m\ n$ is in D, the left sub-term m must not be a constant and the right sub-term n must not have an assigner, i.e.,

$$m\ n \in D \Rightarrow m \in (T - C) \cap \forall k \in T, (n =: k) \notin D$$

2. If an assignment $(m =: n)$ is in D , m can not be the left most term of another term in D, i.e.,

$$(m =: n) \in D \Rightarrow \forall t \in T \text{ and } \|\bar{t}\| \geq 1, m\ \bar{t} \notin D$$

3. The database D must have no circular set of assignments, i.e., $m_0 =: m_1, m_1 =: m_2, \ldots, m_{n-1} =: m_n \in D$, here $n \geq 1 \Rightarrow m_n =: m_0 \notin D$.

The above restrictions force users to enter those and only those business applications that are semantically equivalent to a class of total recursive functions. This will be discussed formally in Sections 5 and 6.

We adduce a few EP database examples below. They will be used later in the article:

Example 3.6 1. The function $s(x) = x^2$ in Example 1.1.1 can be expressed in EP database $S = \{s, s\ 1 =: 1, s\ 2 =: 4, \ldots, 1, 2, 3, \ldots\}$.
2. The function f(x, y) = x + y in Example 1.1.2 can be expressed in EP database $F =: \{f, f\ 1, f\ 1\ 1 =: 2, f\ 1\ 2 =: 3, \ldots f\ 2, f\ 2\ 1 =: 3, f\ 2\ 2 =: 4, \ldots 1, 2, 3, \ldots\}$
3. A sample EP database for the database application in Example 1.1.3 can be expressed as: D = {*SSD*, *SSD John*, *SSD John birth* =: '*6/1/90*', *SSD John SSN* =:*123456789*, *College*, *College admin*, *College admin* (*SSD John*), *College admin* (*SSD John*) *enroll* =: '*9/1/08*', *College admin* (*SSD John*) *Major* =: *College CS*, *College*, *College CS*, *College CS100*, *College CS100* (*College admin* (*SSD John*)), *College CS100* (*College admin* (*SSD John*)) *grade* =: "*F*", '*6/1/90*', *123456789*, '*9/1/08*', "*F*"}
4. $G = \{a, a\ b, (a\ b\ c =: 3), a\ b\ d, b, c, 3\}$.
5. $H = \{p\ s\ =: 1, q\ s =: 1, p, q, s, 1\}$.
6. The union of the sets above $S \cup F \cup D \cup G \cup H$ form an EP database.

There are two obvious propositions for a database D:

Proposition 3.7 1. If an application is in a database, so are its left sub-term and its right sub-term, i.e.

$m\ n \in D \Rightarrow m \in D\ \cap n \in D$

2. If an assignment is in a database, so are its assignee and assigner, i.e.

$m =: n \in D \Rightarrow m \in D\ \cap n \in D$

Proof: Q. E. D. from the EP database definition itself.

4 ORDERING RELATIONS

Among terms, sub-terms, and assignments, there are rich relations (and therefore built-in operators) developed for the EP data model [26] and [28]. For the purposes of this article, we limit our discussion to a few relevant ones.

Definition 4.1 1. Given terms $m\ n \in \mathsf{T}$, we define the relation "has the left sub-term", denoted as $m\ n\ \{+\ m$; and the relation "has the right sub-term", denoted as $m\ n\ \{-\ n$, i.e.,

$\{+ = \{<m\ n, m> \mid m, n \in \mathsf{T}\}$;

$\{- = \{<m\ n, n> \mid m, n \in \mathsf{T}\}$.

The relations {+ and {- are among those built-in operators in Froglingo.

Proposition 4.2
1. {+ and {- are dependent in a database D.
2. {+ and {- are decidable in a database D.

Proof: 1.1. {+ is dependent in D. Because:

$l\ \{+\ m \Rightarrow \exists n \in \mathsf{T}, l \equiv m\ n$. (by 4.1.1)

$l \in D \Rightarrow\ m\ n \in D \Rightarrow n \in D$ (by 3.7.1).

Then {+ is dependent in D by Definition 2.1.

1.2. {-. The proof is similar to {+.

2.1. {+ is decidable in a database D.

Given $m, n \in D$, here $\|m\|, \|n\| \in \mathbb{N}$, find an algorithm to determine if $m\ \{+\ n$ holds in finite steps:

i). We see if the iteration size $\|m\|$ is equal to 1. If it is, then $m\ \{+\ n$ must be false because $m\ \{+\ n$ would imply that m was an application by 4.1.1 and therefore $\|m\|$ should be greater than 1.

ii). If $\|m\| > 1$, then compare to see if m's left sub-term is identical to n. If yes, then $m\ \{+\ n$ is true. Otherwise $m\ \{+\ n$ is false.

All the decisions take finite steps because $\|m\|$ and $\|n\|$ are finite. Therefore {+ is decidable.

In fact, {+ is decidable in the whole term set T.

2.2 {-. The proof is similar to {+.

Theorem 4.3 An EP database is a data model.
Proof. By 3.5 and 4.2.

To support our arguments in subsequent sections, we provide here some additional properties of an EP database.

Definition 4.4 A set X is tree-structured under a relation ρ if there is no circular set of relation elements: $e_0\ \rho\ e_1, e_1\ \rho\ e_2, \ldots, e_{n-1}\ \rho\ e_n$, here $n \in \mathbb{N}$ and $e_0 \equiv e_n$.

Proposition 4.5 An EP database is tree-structured under the relations {+ and {-.

Proof 1 {+. Given a database D, if there was a set of elements in {+ that form a circle: $e_0\ \{+\ e_1, e_1\ \{+\ e_2, \ldots, e_n\ \{+\ e_0$, where $e_i \in D$ for all $i \le n \in \mathbb{N}$, then there were a set of elements $l_0, l_1, \ldots, l_{n-1} \in D$ such that $e_1\ l_0 \equiv e_0, e_2\ l_1 \equiv e_1, \ldots, e_0\ l_{n-1} \equiv e_{n-1}$. It implied that $e_0\ l_{n-1}\ l_{n-2} \ldots l_0 \equiv e_0$. This is not possible.

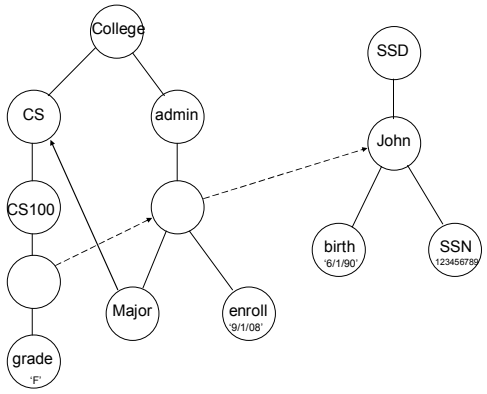
2 {-. The proof is similar to the proof for {+.

Notation 4.6 1. The assignment =: in a database D is a relation: $=: = \{<m, n> \mid m, n \in D, m =: n\}$.

Proposition 4.7 D is tree-structured under the relation =:.
Proof. It is true according to Definition 3.5.3

To show that each of the relations {+, {-, and =: is tree-structured in a database, we provide a graphic of the sample database. (This is Example 3.6.3).

Each circle represents an assignee in the database. A root node represents an identifier, where the identifier is spelled out in the circle center. A non-root node represents an application, where the left sub-term is spelled out in the circle center. The leaf nodes are the assignments normally having explicit assigners. A solid up-down link connects an application to its left sub-term. A dashed arrow connects an application to its right sub-term. A solid arrow connects an assignee to its assigner. For those assignees whose values are constants or other non-assignees, their values are spelled out in the cycles



The up-down links, dash arrows, and solid arrows represent the relations {+, {-, and =:. By ignoring two of the three types of the links, the remaining graph is a tree structure.

5 NORMAL FORM AND REDUCTION

The normal form and reduction rules are adduced below to further prove that the EP data model is equivalent to a class of total recursive functions.

Definition 5.1 Given a database D, the set of normal forms NF is defined as follows:

1. All the constants are normal forms, i.e., $c \in C \Rightarrow c \in NF$
2. All the terms in D that don't have assigners are normal forms by themselves, i.e.,

$t \in D - A \Rightarrow t \in NF$

For example, the database F in Example 3.6.2 has the normal forms: $f, f\,1, f\,2, f\,3, \ldots, 1, 2, 3, \ldots$.

Definition 5.2 Given a database D, we have the one-step evaluation rules, denoted as $\rightarrow$:

1. An identifier not in D is reduced to `null`, i.e.,

$p \in P \cap p \notin D \Rightarrow p \rightarrow$ `null`

2. A term having its assignment in D is reduced to its assigner, i.e.,

$(m =: n) \in D \Rightarrow m \rightarrow n$

3. If $m, n \in NF$, and $m\,n \notin D$, then $m\,n$ is reduced to `null`, i.e.,

$m, n \in NF, m\,n \notin D \Rightarrow m\,n \rightarrow$ `null`

4. The application of two terms are reduced to the application of their normal forms, i.e.,

$m, n \in T, m \rightarrow m', n \rightarrow n' \Rightarrow m\,n \rightarrow m'\,n'$.

Definition 5.3 Let $m, n \in D$. If there is a finite sequence $l_0, \ldots, l_n \in D$, where $n \geq 0$, such that $m \equiv l_0, l_0 \rightarrow l_1, \ldots, l_{n-1} \rightarrow l_n, l_n \equiv n$, then

1. m is effectively, i.e., in finite steps, reduced to n, written as $m \rightarrow_{EP} n$.
2. if n is a normal form and $m \rightarrow_{EP} n$, then we write: $nf(m) = n$.
3. If $m_1 \rightarrow_{EP} n$ and $m_2 \rightarrow_{EP} n$, then we write: $m_1 == m_2$.

Lemma 5.4 A term having an assignment in a database can be effectively reduced to its normal form, i.e.,

$\forall m \in D \cap A \Rightarrow m \rightarrow_{EP} n$, here $n \in D \cap NF$.

Proof. When $m \in D \cap A$, we are always able to find one and only one chain: $m =: l_0, l_0 =: l_1, \ldots, l_{n-1} =: l_n$, where $n \in \mathbb{N}$ because one assignee can only have one assigner (Definition 3.4). The reduction process can be done in finite steps because the relation =: forms tree structures in D (Proposition 4.7). Since $l_n \in D$ has no more assigner, itself is a normal form (Definition 5.1.1). Therefore $l_n \equiv n \in NF$.

Theorem 5.5 An arbitrary term under a database can be effectively reduced to one and only one normal form, i.e., given D, $\forall m \in T$, $m \rightarrow_{EP} n_1$, $m \rightarrow_{EP} n_2$, and $n_1, n_2 \in NF \Rightarrow n_1 \equiv n_2$

Proof. 1. Case $m \in D$,

a). If $m \in D \cap A$, then m is a term with an assignment in D and it can be effectively reduced to one and only one normal form (Lemma 5.4).

b). If $m \in D - A$, then m is a term without assignment, and then it is the normal form by itself (Definition 5.1.2)

2. Case $m \notin D$,

a). if $m \in P$, then $m \rightarrow_{EP}$ `null` (Definition 5.2.1).

b). if $m \in C$, then m itself is the unique normal form (Definition 5.1.1).

c). if $m \equiv n_1\,n_2$. By induction, n_1, n_2 are effectively reduced to two normal forms n_1', n_2' correspondingly, then $m \rightarrow_{EP} n_1'\,n_2'$ (Definition 5.2.4) :

i). If $n_1'\,n_2' \notin D$, then $n_1'\,n_2' \rightarrow_{EP}$ `null` (Definition 5.2.3).

ii). If $n_1'\,n_2' \in D$, then m will be effectively reduced to the normal form of $(n_1'\,n_2')$ according to the step 1 of the proof.

All of the processes are effective (in finite steps).

EP data model, as a formal theory, is consistent. A formal system is said to be consistent if it lacks contradiction, i.e. the ability to derive both a statement and its negation from the system's axioms. For a formal theory that has a decidable reduction process, i.e., that any term it contains can be reduced to its normal form in a finite number of steps, we can redefine the consistency as the following:

Definition 5.6 A formal theory is consistent if a term doesn't have two distinguishable normal forms.

If a term were to have two normal forms, this is equivalent to saying that two normal forms were equal. This is because they would be derived from the same term. At the same time, it would also be saying that two normal forms were not equal, because they were not identically defined in the theory's axioms.

This definition is more straightforward and stronger than the one given in [5] for Turing-equivalent formal systems such as Lambda Calculus, where a reduction process may not terminate with a normal form.

Corollary 5.7 The EP data model is consistent.

Proof. It is clear from Theorem 5.5.

6 APPLICATIVE STRUCTURE

We will show here that an EP database is interpreted as a high-order function.

An applicative structure is commonly used to interpret a Turing-equivalent language [5]. A class of total recursive functions is a strict subset of a class of partial recursive functions, and therefore it can be fitted well into an applicative structure [3]. We will develop an applicative structure in this section such that each term of the EP data model, under a database, is interpreted as an element in the applicative structure. Conversely, an element from the applicative structure can be expressed as an EP database. It is

done by proving that the entire applicative structure can be expressed as an EP database in the next section.

To construct this applicative structure, we first divide the normal forms in a database into three different categories:

Notation 6.1 Given a database D, and therefore its NF,

1. All the constants belong to a category, i.e., $NF^0 \equiv C$
2. All the terms in D that are not constants, don't have assigners, and are not functionally depended on by other terms, i.e.,
   $NF^1 = \{m \mid m \in D - A - C \cap (\forall x \in T, m\ x \notin D)\}$
3. The remaining normal forms belong to the third category, i.e.,
   $NF^+ = NF - NF^0 - NF^1$

As it will become clear soon, a member in the last category NF^+ has a derived semantics; a member in the second category NF^1 is interpreted as nothing else but its syntactical information; and a constant is mapped to a constant (0-ary) function.

Example 6.2 For the database $G = \{a, a\ b, (a\ b\ c =: 3), a\ b\ d, b, c, 3\}$ defined in Example 3.6.4:

1. $NF^1 \equiv \{b, c, a\ b\ d\}$
2. $NF^+ \equiv \{a, a\ b\}$

The applicative structure to be developed will be a class of total recursive functions. Within the applicative structure, applying an element (as a function) to another element (as an argument) always effectively (in finite steps) yields a third element (as the value), and all three elements belong to the collection. Application is the only operation in the applicative structure.

The applicative structure is built on the top of a set of constant functions, or called 0-ary functions. We map the syntactical form of each term from T, except for the special term `null`, to an element of the set of the 0-ary functions. It is done by the Gödel numbering # as it was done for lambda expressions (6.5.6 of [5]):

Definition 6.3 # is an effective one-to-one map:

$$\#: T - \{\texttt{null}\} \to \mathbb{N}.$$

Note that this mapping is purely syntactical, i.e., the syntactical form of a term becomes a 0-ary function in the applicative structure. Note that applying a 0-ary function to any element in the applicative structure yields to a least element, denoted as $\perp$.

The map # is applied not only to the constants C, but also to the non-constant terms in T. Mapping the non-constant terms to $\mathbb{N}$ is to carry the syntactical information to the applicative structure, i.e., to provide an index for the EP terms in the applicative structure. To make it happen, we further introduce an extra 0-ary function, denoted as $\mathbf{i}$.

Now we denote the entire set of the 0-ary functions as $\mathbf{R}^0$.

Definition 6.4 $\mathbf{R}^0 = \mathbb{N} \cup \{\mathbf{i}, \perp\}$, where $\mathbf{i}$ and $\perp$ are two unique 0-ary functions beyond $\mathbb{N}$.

We will not address the issue of how to generate the whole class of total recursive functions over the 0-ary functions $\mathbf{R}^0$. Previous work on this establishes that such a class exists, and that it can be enumerated and represented in an effective applicative structure [the Corollary on page 169 of the text book 11 and the article 2]. We simply denote such a class as $\mathbf{R}$, and represent it in a form of effective applicative structures.

Definition 6.5 The applicative structure ($\mathbf{R}^0$, $\mathbf{R}$, *) satisfies:

1. The set of the 0-ary functions $\mathbf{R}^0$.
2. The complete set of total recursive functions $\mathbf{R}$ over $\mathbf{R}^0$, therefore $\mathbf{R}^0 \subset \mathbf{R}$.
3. $\forall \mathbf{a}, \mathbf{b} \in \mathbf{R}$, there is a operator *, such that $\mathbf{a} * \mathbf{b}$ is effectively (in finite steps) reduced to $\mathbf{c} \in \mathbf{R}$, denoted as $\mathbf{a} * \mathbf{b} = \mathbf{c}$.

Here, we are not interested in how $\mathbf{a} * \mathbf{b}$ is reduced to $\mathbf{c}$, the result, but what the result is.

To help the discussion later, and to understand better the applicative behavior of a total recursive function, we give an alternative notion of an element $\mathbf{f} \in \mathbf{R}$.

Notation 6.6 1. let $\mathbf{f} \in \mathbf{R}$, $\mathbf{f}$ is alternatively expressed as:

$$\mathbf{f} = \{\langle \mathbf{e}, \mathbf{j} \rangle \mid \mathbf{e} \in \mathbf{R} \cap \mathbf{f} * \mathbf{e} = \mathbf{j}\}$$

2. The set is also called the properties of $\mathbf{f}$.

Below are the rules for mapping EP terms under a database to $\mathbf{R}$.

Definition 6.7 Given a database D, and an arbitrary $m \in T$, the semantics $[m]$ is derived according to the following rules:

1. The term `null` in T is interpreted as the least element in $\mathbf{R}$, i.e., $[\texttt{null}] = \perp$,
2. A constant in $C - \{\texttt{null}\}$ is mapped to the corresponding Gödel number, i.e.,
   $\forall c \in C, [c] = \# c$,
3. A normal form in NF^1 is mapped to the function that only contains its syntactical information, i.e.,
   $\forall m \in NF^1, [m] = \{\langle \mathbf{i}, \#m \rangle\} \cup \{\langle [o], \perp \rangle \mid o \in T\}$,
4. A normal form in NF^+ is mapped to the function containing its syntactical information and mainly its derived information, i.e.,
   $\forall m \in NF^+, [m] = \{\langle \mathbf{i}, \#m \rangle\}$
   $\cup \{\langle [n_i], [m\ n_i] \rangle \mid$ for all $n_i \in T$, such that $m\ n_i \in D\}$
   $\cup \{\langle [o_i], \perp \rangle \mid$ for all $o_i \in T$, such that $m\ o_i \notin D\}$,
5. The semantics of an arbitrary term is the semantics of its normal form, i.e.,
   $\forall m \in T, [m] = [\mathsf{nf}(m)]$.

We will prove that $[m]$ is an element in $\mathbf{R}$. In Definition 6.7.4 above, we attempted to find all $m\ n_i$, where $i = 0, \ldots, n$ for an integer $n \geq 0$, such that $m\ n_i \in D$. By induction, we assumed that m and n_i have their semantics $[n_i]$ and $[m\ n_i]$. The collection of all the elements $\langle [n_i], [m\ n_i] \rangle$ is the derived information.

For each normal form $m \in NF^1 \cup NF^+$, we added its syntactical information $\{\langle \mathbf{i}, \#m \rangle\}$. This ensures that the normal form has a unique interpretation. There are cases in which an element $\langle [n_i], [m\ n_i] \rangle$ might not be interpreted uniquely among the derived information if the syntactical information was not a part of the semantics $[m]$. In the database $H = \{p\ s =: 1, q\ s =: 1, p, q, s, 1\}$ from Example 3.6.5, for example, both $[p]$ and $[q]$ would end up with the same interpretation: $\{\langle [\#s], [\#1] \rangle\}$ if $\langle \mathbf{i}, \#p \rangle$ and $\langle \mathbf{i}, \#q \rangle$ were not a part of their corresponding semantics.

Carrying the syntactical information to the interpretation makes sense in the practice of database application. It is not unusual for two database entities to represent two distinct objects in the real world while temporarily having the same set of attributes.

Before demonstrating that a term in a database is interpreted as a high-level function, we present two intermediate results: If two terms are equal, their interpretations are equal, and if two terms cannot be reduced to the same normal form, their interpretations

are not equal. Note that the interpretation of a term is not formally claimed to be a high-level function (a member in $\mathbf{R}$) yet until Theorem 6.11.

Lemma 6.8 $m \rightarrow_{EP} n \Rightarrow [m] \equiv [n]$
Proof. It is clear from Definition 6.7.

Notation 6.9 If a term m is reduced to its normal form n and n is not identical to another normal form k, then we write $m\ !\rightarrow_{EP} k$.

Lemma 6.10 $m\ !\rightarrow_{EP} n \Rightarrow [m]\ !\equiv [n]$.
Proof.
1. If m, $n \in NF^0 \cup NF^1$, then $[m] \neq [n]$ by Definitions 6.7.1, 6.7.2, and 6.7.3.
2. If $m, n \in NF^+$,
   If $m\ !\rightarrow_{EP} n$, then
   $\Rightarrow m\ !\equiv n$ (by 5.1, the normal form definition)
   $\Rightarrow \#m\ !\equiv \#n$ (By 6.3, the Gödel numbering)
   $\Rightarrow [m]\ !\equiv [n]$ (By 6.7.4, distinguished syntactical information)
3. If $m \in NF^+$ and $n \in NF^0 \cup NF^1$, then $[m]\ !\equiv [n]$ because $[m]$ has derived information while $[n]$ doesn't (by 6.7, the interpretation definition). Also they have different syntactical information.
4. If $m, n \in T$, then
   If $m\ !\rightarrow_{EP} n$, then
   $\Rightarrow m \rightarrow_{EP} m'$, $n \rightarrow_{EP} n'$ (by 5.5, the reduction theorem)
   $\Rightarrow m'\ !\rightarrow_{EP} n'$ (by the given condition of this lemma).
   $\Rightarrow [m']\ !\equiv [n']$ (by the proof 2 above)
   $\Rightarrow [m]\ !\equiv [n]$ (by 6.7.5)

Theorem 6.11 (soundness) An arbitrary term under a database has an interpretation of function, i.e., given D, $\forall m \in T \Rightarrow [m] \in \mathbf{R}$.

Proof. 1. $m \in NF^0$. $[m]$ is a 0-ary function, and therefore $[m] \in \mathbf{R}$ (by Definition 6.7.1 and 6.7.2).

2. $m \in NF^1$. $[m] \in \mathbf{R}$ and is a unary function by 6.7.3.

3. $m \in NF^+$.
   $\Rightarrow [m] = \{<\mathbf{i}, \#m>\} \cup \{<[n_i], [m\ n_i]> \mid m\ n_i \in D, i \text{ is } 0, 1, \ldots k \text{ for a } k > 0\}$ (by 6.7.4, here the formula is rewitten)
   $\Rightarrow n_i\ !\rightarrow_{EP} n_j$, here $n_i, n_j \in NF$, for any i and j between 0 and k and $i \neq j$. (by 3.5.1, the right sub-term of an application in D has no assignment)
   $\Rightarrow [n_i]\ !\equiv [n_j]$ (By lemma 6.10), and
   $\mathbf{i}\ !\equiv [n_i]$, for all $i < k$ (By Definition 6.5)
   $\Rightarrow [m] \in \mathbf{R}$ (by 6.6, where the first coordinates of all the elements in a function are distinguishable)

The soundness becomes stronger with the proof that the reduction rules of the EP data are consistent with the applicative behavior of functions.

Corollary 6.12 $[m\ n] = [m] * [n]$
Proof.
1. $m \in NF^0$.
   $\Rightarrow [m] \in \mathbf{R}^0$ (by 6.7.2)
   $\Rightarrow [m] * \mathbf{a} = \perp$, here $\mathbf{a} \in \mathbf{R}$ (by 6.5.1)
   Because m is a constant, then for any term n, we have
   $m\ n \rightarrow_{EP} \texttt{null}$ (by 5.2.3)
   $\Rightarrow [m\ n] = [m] * [n] = \perp$ (by 6.7.1)
2. $m \in NF^1$.
   $\Rightarrow [m] = \{<\mathbf{i}, \#m>\} \cup \{<[o], \perp> \mid o \in T\}$ (by 6.7.3)
   $\Rightarrow [m] * [n] = \perp$, here $[n] \in \mathbf{R} - \{\mathbf{i}\}$ (by 6.7.3, 6.6)
   $m\ n \rightarrow_{EP} \texttt{null}$ (by 5.2.3)
   $\Rightarrow [m\ n] = [m] * [n] = \perp$ (by 6.7.1)
3. $m \in NF^+$.
$\Rightarrow [m] = \{<\mathbf{i}, \#m>\}$
   $\cup \{<[n_i], [m\ n_i]> \mid m\ n_i \in D, i \text{ is } 0, 1, \ldots k \text{ for a } k > 0\}$
   $\cup \{<[o_i], \perp> \mid \text{for all } o_i \in T, \text{ such that } m\ o_i \notin D\}$
   (Definition 6.7.4, the formula in the middle was written)
   If $n \rightarrow_{EP} n'$ and n' is the normal form (Theorem 5.5),
   a). If $n' \equiv n_i$, where $i \leq k$ such that $m\ n_i \in D$, then
      i). $[n] \equiv [n_i]$ (by 6.7.5)
      ii). $[m] * [n_i] = [m\ n_i]$ (by 6.7.4 and 6.6. Since the function $[m]$ has an element $<[n_i], [m\ n_i]>$, then $[m] * [n_i] = [m\ n_i]$ is true)
      $\Rightarrow [m] * [n] = [m\ n_i]$
      iii). $m\ n \rightarrow_{EP} m\ n_i$ (by 5.2.4)
      $\Rightarrow [m\ n] = [\mathsf{nf}(m\ n)]$ (by 6.7.5)
      $\Rightarrow [m\ n_i] = [\mathsf{nf}(m\ n_i)]$ (by 6.7.5)
      $\Rightarrow \mathsf{nf}(m\ n) \equiv \mathsf{nf}(m\ n_i)$ (by 5.5)
      $\Rightarrow [m\ n_i] \equiv [m\ n]$ (by 6.8)
      $\Rightarrow [m\ n] = [m] * [n]$ (based on the results earlier)
   b). If $m\ n' \notin D$, then
      $\Rightarrow [m] * [n_i] = \perp$ (by 5.2.3)
      $\Rightarrow m\ n_i \rightarrow_{EP} \texttt{null}$ (by 6.7.4)
      $\Rightarrow [m\ n] = [m] * [n]$ (By 6.7.1)
4. $m \in T$.
   Assume that $m \rightarrow_{EP} m'$, $n \rightarrow_{EP} n'$, and $m', n' \in NF$. Then $[m'\ n'] = [m'] * [n']$ (by proof step 3 of this corollary)
   i). $[m] = [m']$, $[n] = [n']$ (by 6.7.5)
      $\Rightarrow [m] * [n] = [m'] * [n']$ (by 6.6)
   ii). $m'\ n' \rightarrow_{EP} \mathsf{nf}(m'\ n')$, $m\ n \rightarrow_{EP} \mathsf{nf}(m\ n)$
      $\Rightarrow \mathsf{nf}(m'\ n') \equiv \mathsf{nf}(m\ n)$ (by 5.5)
   iii). $[m'\ n'] = [\mathsf{nf}\ (m'\ n')]$, $[m\ n] = [\mathsf{nf}\ (m\ n)]$ (by 6.7.5)
      $\Rightarrow [m'\ n'] \equiv [m\ n]$ (from the two equations above)
      $\Rightarrow [m\ n] = [m] * [n]$ (based on the results earlier)

7 PROPERTY ENUMERATION

In Section 6, we showed that a term in a database is interpreted as a high-level function. We show in this section that an arbitrary function can be mapped back to a database.

Rather than show that a single total recursive function is computable, we show that the complete set of properties of a total recursive function can be mapped to, and therefore "stored," in an EP database. It will be true mathematically even if the properties of a function are infinitely long, as the database space is unlimited, hypothetically.

In practice, a database always stores a very small portion of a class of total recursive functions; however, we will show that the entire class of total recursive functions can be mapped to a database. This simplifies our work in this section in two folds. First of all, we don't have to deal with the issues between the functions being stored, and the rest of the functions not being stored, in a database. Therefore the issue of the data evolution process in an actual database is not addressed in the interpretation. Secondly, because each function in $\mathbf{R}$ is a curried and therefore a unary function, the iteration size of each term m in the database will not be more than 2, i.e., $\|m\| \leq 2$. This doesn't utilize all the syntactical flexibilities in the EP data model, i.e., allowing the

iteration size of a term to be as large as a business needs. Therefore, the issues that attend the management of dependent data via independent data in an actual database are not addressed in the interpretation. Nevertheless, the simplification doesn't impact our conclusion that the EP data model is equivalent to a class of total recursive functions.

Recall that we introduced an extra 0-ary function **i** in **R** in Section 6. This special constant is not necessary for this section, but it doesn't affect our conclusions by continuing to use **R** without worrying about if the special constant is in **R** or not. The set of constants over which a class of total recursive functions is constructed is inessential [3].

Now we need to develop a complementary mapping of the function #:

Definition 7.1 1. Υ^0 denotes an effective one-to-one map: Υ^0: $\mathbf{R}^0 \to C$.

2. Υ^1 denotes an effective one-to-one map: Υ^1: $\mathbf{R} - \mathbf{R}^0 \to P$,

3. Let $\mathbf{a} \in \mathbf{R}$, define $\lceil \mathbf{a} \rceil$ inductively as the following:

$\mathbf{a} \in \mathbf{R}^0 \Rightarrow \lceil \mathbf{a} \rceil = \Upsilon^0\ \mathbf{a}$

$\mathbf{a} \in \mathbf{R} - \mathbf{R}^0 \Rightarrow \lceil \mathbf{a} \rceil = \Upsilon^1\ \mathbf{a}$

This time, the 0-ary functions are mapped to the constants C, and the rest of functions in **R** are mapped to the identifiers P.

Lemma 7.2 $\forall\ \mathbf{a} \in \mathbf{R}$, $\|\lceil \mathbf{a} \rceil\| = 1$.

Proof. It is clear from 7.1.1 and 7.1.2.

Definition 7.3 1. Given a $\mathbf{m} \in \mathbf{R}$, let $\wp(\mathbf{m})$ be a set of assignments:

$\wp(\mathbf{m}) = \{\lceil \mathbf{m} \rceil \lceil \mathbf{n} \rceil =: \lceil \mathbf{o} \rceil \mid \mathbf{n} \in \mathbf{R}, \mathbf{m} * \mathbf{n} = \mathbf{o}, \mathbf{o} \mathrel{!} \equiv \perp\}$

2. $\Im = \cup_{\mathbf{m} \in \mathbf{R}}\ \wp(\mathbf{m}) \cup \mathbf{R}$

$\wp(\mathbf{m})$ is nothing but the collection of its properties of a function **m** in the form of EP assignments. $\Im$ is nothing but the collection of the properties of the entire class of total recursive functions **R**, i.e, $\cup_{\mathbf{m} \in \mathbf{R}}\ \wp(\mathbf{m})$.

To satisfy Property 3.7, we automatically consider that $\lceil \mathbf{m} \rceil \lceil \mathbf{n} \rceil$, $\lceil \mathbf{m} \rceil$, and $\lceil \mathbf{n} \rceil$ are the additional elements of $\wp(\mathbf{m})$ in Definition 7.3.1 though they were not explicitly spelled out.

As noted earlier, we didn't consider an accumulative process of adding data piece by piece during the mapping of **R**'s properties to a database; instead, we assume that **R**'s properties in the form of EP terms and EP assignments are available already. This assumption is valid, again because previous work establishes that the properties of **R**, a class of total recursive function, can be enumerated [the Corollary on page 169 of the text book 11 and the article 2].

Lemma 7.4 $\Im$ is an EP database.

Proof. We need to show that each assignment in $\Im$ satisfies the conditions of an EP database defined in 3.5, and also specify that $\Im$ is enumerable, i.e., it can be effectively produced under the assumption of infinite time and space.

I). Prove $\lceil \mathbf{m} \rceil \lceil \mathbf{n} \rceil =: \lceil \mathbf{o} \rceil$ is a valid assignment in a database D, for $\forall \mathbf{m}, \mathbf{n} \in \mathbf{R}$, $\mathbf{m} * \mathbf{n} = \mathbf{o}$, and $\mathbf{o} \mathrel{!} \equiv \perp$.

i). Prove that $\lceil \mathbf{m} \rceil$ is valid to be a left sub-term in D

a). $\mathbf{m}$ must be a non 0-ary function, i.e., $\mathbf{m} \in \mathbf{R} - \mathbf{R}^0$ (by 7.3.1)

b). $\lceil \mathbf{m} \rceil \in P$ (By 7.1.2)

c). $\lceil \mathbf{m} \rceil$ is valid to be a left sub-term in D (partially satisfy Definition 3.5.1)

ii) Prove that $\lceil \mathbf{n} \rceil$ is valid to be a right sub-term in D

a). $\|\lceil \mathbf{n} \rceil\| = 1$, $\|\lceil \mathbf{m} \rceil\| = 1$ (by Lemma 7.2)

b). For each $\lceil \mathbf{m} \rceil \lceil \mathbf{n} \rceil =: \lceil \mathbf{o} \rceil$ in $\Im$, $\|\lceil \mathbf{m} \rceil \lceil \mathbf{n} \rceil\| = 2$.

c). $\lceil \mathbf{n} \rceil$ must not have an assignment, i.e., $\lceil \mathbf{n} \rceil \in \Im - A$ (by 7.3.1, i.e., the only form of assignments is $\lceil \mathbf{m} \rceil \lceil \mathbf{n} \rceil =: \lceil \mathbf{o} \rceil$ in $\Im$)

d). $\lceil \mathbf{n} \rceil$ is valid to be a right sub-term in database. (partially and then completely satisfy Definition 3.5.1)

iii). According to Definition 7.3.1, there is not a third element **o** from **R** such that $\lceil \mathbf{m} \rceil \lceil \mathbf{n} \rceil \lceil \mathbf{o} \rceil$ is in $\Im$. Therefore Definition 3.5.2 is satisfied.

iv). Prove that there is not a set of assignments that forms a circle in $\Im$ in satisfying Definition 3.5.3. Assume that $\exists\ e_0, e_1, \ldots, e_j \in \Im$, here $j \geq 0$, $e_0 =: e_1$, $e_1 =: e_2 \ldots$, $e_j =: e_0 \in \Im$. Assume such a circle existed,

a). If $j = 0$, then we would have: $e_0 \equiv \lceil \mathbf{m} \rceil \lceil \mathbf{n} \rceil$, $\lceil \mathbf{m} \rceil \lceil \mathbf{n} \rceil =: \lceil \mathbf{m} \rceil \lceil \mathbf{n} \rceil$, and $\lceil \mathbf{m} \rceil \lceil \mathbf{n} \rceil \equiv \lceil \mathbf{o} \rceil$. The assigner's iteration size $\|\lceil \mathbf{m} \rceil \lceil \mathbf{n} \rceil\| = 2$. It is contradictory to Lemma 7.2 that $\lceil \mathbf{o} \rceil = 1$.

b). If j > 0, the proof is similar to the case of $j = 0$ when $e_j =: e_0$, and $e_0 \equiv \lceil \mathbf{m} \rceil \lceil \mathbf{n} \rceil$.

We conclude that $\Im$ satisfies the restrictions to be an EP database (by 3.5).

II). According to the conclusions from the Corollary on page 169 in [11] and the articles [2 and 3], there is an effective procedure to generate such a set $\Im$.

Lemma 7.5 $\forall \mathbf{m} \in \mathbf{R}$, $\lceil \mathbf{m} \rceil$ is a normal form in $\Im$

Proof. 1. If **m** is a 0-ary function, then $\lceil \mathbf{m} \rceil \in C$, and $\lceil \mathbf{m} \rceil \in NF$ (by 7.1.1 and 5.1.1).

2. If **m** is a non 0-ary function, then $\lceil \mathbf{m} \rceil \in P$ (by 7.1.2). Since the iteration size of an assignee in $\Im$ is 2 (By the definition 7.3.1), and $\lceil \mathbf{m} \rceil$'s iteration size is 1 (By Lemma 7.2), then **m** must not have an assignment. Then $\lceil \mathbf{m} \rceil$ is a normal form in $\Im$ (By 5.1.2).

Definition 7.6 $\forall \mathbf{m_1}, \mathbf{m_2}, \ldots, \mathbf{m_n} \in \mathbf{R}$, $\forall\ n \in \mathbb{N}$, if the equation:

$$\lceil \mathbf{m}_1 \rceil \lceil \mathbf{m}_2 \rceil \ldots \lceil \mathbf{m}_n \rceil == \lceil \mathbf{m}_1 * \mathbf{m}_2 * \ldots * \mathbf{m}_n \rceil$$

is true, then all the total recursive functions are EP-definable. Here $\lceil \mathbf{m}_1 * \mathbf{m}_2 * \ldots * \mathbf{m}_n \rceil \equiv [\mathbf{n}]$ while $\mathbf{m}_1 * \mathbf{m}_2 * \ldots * \mathbf{m}_n \equiv (\ldots(\mathbf{m}_1 * \mathbf{m}_2) * \ldots * \mathbf{m}_n) = \mathbf{n}$ for a $\mathbf{n} \in \mathbf{R}$ according to the property of functions in 6.6.

Theorem 7.7 (completeness) Total recursive functions are *EP*-definable.

Proof. Prove by induction.

1. If $n = 1$, prove $\lceil \mathbf{m}_1 \rceil == \lceil \mathbf{m}_1 \rceil$. It is true by itself and by the theorem 5.5.

2. If $n = 2$, prove $\lceil \mathbf{m}_1 \rceil \lceil \mathbf{m}_2 \rceil == \lceil \mathbf{m}_1 * \mathbf{m}_2 \rceil$. By 7.3, we have $\mathbf{m}_1 * \mathbf{m}_2 = \mathbf{o}$, here $\mathbf{o} \in \mathbf{R}$

i). If $\mathbf{o} \equiv \perp$, then $\lceil \mathbf{o} \rceil \equiv \lceil \mathbf{m}_1 * \mathbf{m}_2 \rceil \equiv$ `null` (by 7.1)

a). $\lceil \mathbf{m}_1 \rceil \lceil \mathbf{m}_2 \rceil \notin \Im$ (by Definition 7.3)

b). $\lceil \mathbf{m}_1 \rceil, \lceil \mathbf{m}_2 \rceil$ are in normal form (lemma 7.5)

c). $\lceil \mathbf{m}_1 \rceil \lceil \mathbf{m}_2 \rceil$ == `null` (By 5.2.3).

e). $\lceil \mathbf{m}_1 \rceil \lceil \mathbf{m}_2 \rceil == \lceil \mathbf{m}_1 * \mathbf{m}_2 \rceil$ is true.

ii). If $\mathbf{o} \neq \perp$

a). Because $\lceil \mathbf{m}_1 \rceil \lceil \mathbf{m}_2 \rceil =: \lceil \mathbf{o} \rceil$ (by 7.3), then
$\lceil \mathbf{m}_1 \rceil \lceil \mathbf{m}_2 \rceil == \lceil \mathbf{o} \rceil$ (by 5.2)

b). Since $\mathbf{m}_1 * \mathbf{m}_2 = \mathbf{o}$ (by 7.3), then
$\lceil \mathbf{m}_1 * \mathbf{m}_2 \rceil \equiv \lceil \mathbf{o} \rceil$ (by 7.6)

c). $\lceil \mathbf{m}_1 \rceil \lceil \mathbf{m}_2 \rceil == \lceil \mathbf{m}_1 * \mathbf{m}_2 \rceil$ is ture

3. If $n > 2$, prove $\lceil \mathbf{m}_1 \rceil \lceil \mathbf{m}_2 \rceil \ldots \lceil \mathbf{m}_n \rceil == \lceil \mathbf{m}_1 * \mathbf{m}_2 * \ldots * \mathbf{m}_n \rceil$. by induction. Assume $\lceil \mathbf{m}_1 \rceil \lceil \mathbf{m}_2 \rceil \ldots \lceil \mathbf{m}_{n-1} \rceil == \lceil \mathbf{m}_1 * \mathbf{m}_2 * \ldots * \mathbf{m}_{n-1} \rceil$. Then

a). $\mathbf{m}_1 * \mathbf{m}_2 * \ldots * \mathbf{m}_{n-1} = \mathbf{o}$, here $\mathbf{o} \in \mathbf{R}$ (by 6.6)

b). $\lceil \mathbf{m}_1 * \mathbf{m}_2 * \ldots * \mathbf{m}_{n-1} \rceil \equiv \lceil \mathbf{o} \rceil$ (by 7.6)

c). $\lceil \mathbf{m}_1 \rceil \lceil \mathbf{m}_2 \rceil \ldots \lceil \mathbf{m}_{n-1} \rceil == \lceil \mathbf{o} \rceil$ (by the assumption in step 3)

d). $\lceil \mathbf{m}_1 \rceil \lceil \mathbf{m}_2 \rceil \ldots \lceil \mathbf{m}_{n-1} \rceil \lceil \mathbf{m}_n \rceil == \lceil \mathbf{o} \rceil \lceil \mathbf{m}_n \rceil$ (by 5.5)

e). $\lceil \mathbf{m}_1 \rceil \lceil \mathbf{m}_2 \rceil \ldots \lceil \mathbf{m}_{n-1} \rceil \lceil \mathbf{m}_n \rceil == \lceil \mathbf{o} * \mathbf{m}_n \rceil$ (by step 2 above)

e). $\lceil \mathbf{m}_1 \rceil \lceil \mathbf{m}_2 \rceil \ldots \lceil \mathbf{m}_{n-1} \rceil \lceil \mathbf{m}_n \rceil == \lceil \mathbf{m}_1 * \mathbf{m}_2 * \ldots * \mathbf{m}_{n-1} * \mathbf{m}_n \rceil$

7 SUMMARY

There are many language systems, such as lambda calculus, Turing machine, and combinatory logic, that are equivalent to classes of partial recursive functions [5]. This article has proven that EP data model is a language system that is equivalent to a class of total recursive functions. Obviously, the relational data model and the hierarchical data model are the special cases. Our proof was accomplished by showing that a term in an EP database is a (high-order) function, and that the properties of a class of total recursive functions, presented in a certain format, form an EP database. This doesn't mean that all the software applications can be practically managed by using EP data model alone; rather, it offers a consistent and always-halting method to manage as much finite data as a business application requires and by so doing minimizes the effort of software development and maintenance.

Forcing software applications to be totally recursive is what distinguishes a data model from a programming language, but it is not the sole difference. Built-in operators of data model, derived from dependent and other relations, have been well recognized as a functionality that extends beyond a programming language. Without exception, the EP data model offers a set of rich, built-in operators. (A typical example is the Froglingo expression "Z <=+ A" for the query "Is there a path between A to Z in a directed graph?" [28, 26].)

A database is an evolving subset of the class of total recursive functions, rather than a whole class of total recursive functions. Obtaining the semantics of database evolving process in the context of the class of total recursive functions requires discussion beyond the scope of this article. This issue can be addressed by expanding Section 7 with various applicative structures.

The EP data mode is a type-free system. Since it is equivalent to a totally recursive function, the type-free aspect doesn't cause any exception or failure to terminate. Instead, it can easily express self-reference and totally recursive functions. (For example, the function F = {<0, 1>, <F, 1>}, which has no straightforward expression in Lambda Calculus, can be expressed in EP data model as F = {F 0 =: 1, F F =: 1}.)

The EP data model expresses functions by enumerating their properties. Its system performance is independent of the complexity of the functions themselves. No matter how long it takes to enumerate the properties of functions, the system performance of the EP data model (i.e., the time complexity of constructing the functions in the database and querying against the properties), is solely dependent on the generic data structure used to implement the EP data model.

Froglingo is not aimed to address the issue of "computable queries", i.e., the computable functions from relation to relations discussed in [9 and 19], though it can be a special case in Froglingo.

Provided that a data model is easier than a programming language in representing the semantics of the given data model, a programming language like Froglingo that is based on EP data model is ultimately the easiest to use in database application development and maintenance.

REFERENCES:

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

2 A. Ambainis and J. Smotrovs. "Enumerable Classes of Total Recursive Functions: Complexity of Inductive Inference". Lecture Notes in Computer Science: Vol. 872, Page 10 – 25. Proceedings of the 4th International Workshop on Analogical and Inductive Interence: Algorithmic Learning Theory. 1994.

3 A. Asperti and A. Ciabattoni. "Effective Applicative Structures". Category Theory and Computer Science, Lecture Notes in Computer Sicence, Volume 953/1995, Page 81-95.

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

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

6 P. Boncz, T. Grust, M.V. Keulen, S. Manegold, J. Rittinger, J. Teubner. "MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine", SIGMOD 2006, June 27-29, 2006, Chicago, Illinois, USA.

7 P. Buneman. "Functional Database Language and the Functional Data Model, A position paper for the FDM workshop". Workshop of Functional Data Model, June 1997.

8 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.

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

10 P. Chen. "The Entity Relationship Mode – Toward a Unified View of Data", TODS, 1:1, March 1976.

11 J. E. Hopcroft, J. D. Ullman. “Introduction to Automata theory, Languages, and Computation”. Addison-Wesley Publishing Company, Inc., 1979.

12 H. V. Jagadish, Laks V. S. Lakshmanan, D. Srivastava, K. Thompson. “TAX: A Tree Algebra for XML.

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

14 C. Laasch, M. H. Scholl. “A Functional Object Database Language”, Proceedings of DBPL4, 1993.

15 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.

16 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.

17 Report of the CODASYL Data Base Task Group, ACM, April 1971.

18 S. Rozen, D. Shasha. “Rationale and Design of BULK”, Proceedings of DBPL3, 1992, page 71-85.

19 Peter Schauble. “On the Expressive Power of Query Languages”. ACM Transactions on Information Systems, Vol. 12, No. 1, January 1994, Page 69 - 91.

20 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.

21 D. Scott. “Models for Various Type-Free Calculi”. Suppes et al. 1973, page 157 - 187.

22 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.

23 C. Small, A. Poulovassilis. “An Overview of PFL”, Proceeding of DBPL3, 1992, page 96-110.

24 K. H. Xu, Jingsong Zhang, Shelby Gao. “Froglingo, When a Data Model is Equivalent to the Class of Total Recursive Functions”. Submitted to SIGMOD 2010 Demo Proposal.

25 K. H. Xu, Jingsong Zhang, Shelby Gao. “Assessing Easiness with Froglingo”. The Second International Conference on the Application of Digital Information and Web Technologies, 2009, page 847 - 849.

26 K. H. Xu, Jingsong Zhang. “A User’s Guide to Froglingo, Database Application Management System”. To appear at the website: http://www.froglingo.com.

27 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.

28 K. H. Xu. “EP Data Model, a Language for Higher-Order Functions”. Manuscript unpublished, March 1999. http://www.froglingo.com/ep99.pdf.

29 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.