Let a Data Model be equivalent to a Class of Total Recursive Functions

Kevin H. Xu, Shelby Gao, Jingsong Zhang, Roger R. McKeown
Bigravity Business Software
{kevin, shelby, jingsong, roger}@froglingo.com

Abstract

A formal language - the EP database is defined in this article. We prove that it is semantically equivalent to a class of total recursive functions. The conclusion serves as the theoretical foundation of Froglingo that consolidates the multi-component system architecture of the traditional software technologies into a single component. It also suggests how one might assess easy-of-use more objectively.

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, as described in [11, 12, and 13], 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 combining 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.

This article is to formally prove that the EP data model, Froglingo without variables, is a language that is semantically equivalent to a class of total recursive functions. This conclusion is the theoretical foundation for the consolidations following Froglingo [8].

Beyond the fact that the traditional software architecture is consolidated in Froglingo, the authors in the article [10 and 7] related the work of Froglingo with the rest in the fields programming language and database management by objectively assessing a language's ease-of-use. 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 the ease-of-use reached the limit mathematically when a data model was semantically equivalent to a class of total recursive functions.

To formalize the notions discussed in the article [10], the authors in the technical report [9] carefully chose a mathematical definition for the concept of "data model" such that:

1. The definition preserves the essence of a data model that DBMSs started in the 1970s, i.e., a data model is set-oriented in queries and update operations on finite data. The EP data model, along with many traditional data models and data structures, is bound to the definition.
2. The definition quantitatively distinguishes a data model from a programming language, i.e., the operations and queries from a data model are decidable and therefore always halt.

In this article solely focus on proving the conclusion that the EP data model is semantically equivalent to a class of total recursive functions. Rather than being burdened by the concept of data model, we simply use the terminology "EP database" (often shorten as "database") referring to the formal language to be discussed throughout this article. Rather than exploring the full features of the EP database, we introduce those only relevant to this article. To further cut the length of this paper, many lengthy proofs are not provided here. Please

reference the technical report [9] for the detail. By putting the irrelevant notions and the corresponding notations aside, we attempt to discuss the EP database as a regular formal language more effectively.

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 is an integer. It can be equivalently expressed by its properties: $s = \{<0, 0>, <1, 1>, <2, 4>, \ldots\}$.

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 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 the article [14], 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 2.1: 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 2.2 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 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 the 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; The lambda calculus doesn't have identifiers.

In addition to term, assignment is another important concept in the EP database.

Definition 2.3: 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 2.4 1. Given an expression $a \equiv b$, the symbol ≡ indicates that the two symbols a and b are identical.

2. Given an expression $a \not\equiv b$, the symbol !≡ indicates that the two symbols a and b are not identical.
3. Let $\mathbb{N}$ be the set of natural numbers {0, 1, 2, …}.
4. Let $m, n_0, \ldots, n_i \in T$, here $i \in \mathbb{N}$. We write:
$m\ n_0 \ldots n_i \equiv m\ \overline{n} \equiv (\ldots((m\ n_0)\ n_2) \ldots n_i)$.
We further write: $\overline{n} \cong n_0, \ldots, n_i \in T$; and $\|\ \overline{n}\ \| = i$.
5. Given a term $m\ \overline{n}$, m is called a left-most term of $m\ \overline{n}$, here $\|\ \overline{n}\ \| \geq 0$. Note that the terms $m\ n_0$, $m\ n_0\ n_1$, …, and $m\ \ n_0 \ldots n_i$ are also called left-most terms of $m\ \ n_0 \ldots n_i$.

Now we introduce the formal definition of the EP database:

Definition 2.5 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 a left most term of another term in D, i.e.,

$(m =: n) \in D \Rightarrow \forall t \in T$ 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 two terminologies may cause a little confusion, but they are distinguishable: "the EP database" is referred as a language and "an EP database" or "a database" is referred as an instance of database in the EP database language.

The above restrictions enforce 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 4 and 5.

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

Example 2.6 1. The function $s(x) = x^2$ in Example 1.1.1 can be expressed in an EP database S = {*s*, *s 1 =: 1*, *s 2 =: 4*, …, *1*, *2*, *3*, …}.

2. The function f(x, y) = x + y in Example 1.1.2 can be expressed in an 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*, *d*, *3*}.

5. H = {*p s =*: *1*, *q s =*: *1*, *p*, *q*, *s*, *1*}.

6. *DG = {v1 v2 =: v2; v2 v1 =: v1}.* (A directed graph with a circle)

7. The union of the sets above $S \cup F \cup D \cup G \cup H \cup DG$ forms an EP database.

There are two obvious propositions for a database D:

Proposition 2.7 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.

3 Normal Form and Reduction

The normal form and reduction rules are adduced below in order to prove that the EP database is equivalent to a class of total recursive functions. A term can be reduced to another term; and a normal form is a term not further reducible by applying the reduction rules.

Definition 3.1 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.,
   $t \in D - A \Rightarrow t \in NF$.

The following terms are some of the normal forms in the database of Example 2.6.7: *s*, *f*, *f 1*, *SSD, College CS, a b, p, v1, 1*, *2*, *3*.

Definition 3.2 Given a database D, the one-step evaluation rule is denoted as →:

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

Given the database in Example 2.6.7, the following samples are valid one-step evaluations:

an_undefined_id → `null`;
f 1 2 → *3;*
College Admin (SSD John) Major → *College CS;*
SSD College → `null`*;*
v1 v2 v1 ≡ *(v1 v2) v1* → *v2 v1;*

Definition 3.3 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: $\mathrm{nf}(m) = n$.
3. If $m_1 \rightarrow_{EP} n$ and $m_2 \rightarrow_{EP} n$, then we write: $m_1 == m_2$.

Given the database in Example 2.6.7, the following examples are effective reductions:

f 1 2 →$_{EP}$ *3;*
College Admin (SSD John) →$_{EP}$ *College Admin (SSD John);*
College Admin (SSD John) Major →$_{EP}$ *College CS;*
nf(*College Admin (SSD John) Major*) = *College CS;*
v1 v2 v1 →$_{EP}$ *v1;*
v1 v2 v1 v2 v1 == v1 v2 v1

Now we are ready to show that the reduction rules give a term a unique normal form (note that the intermediate Lemma 3.4 is provided in the Proof 5.4 of [9]):

Theorem 3.5 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. See the Proof 5.5 in [9].

EP database, 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 3.6 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 [3] for Turing-equivalent formal systems such as the lambda calculus, where a reduction process may not terminate with a normal form.

Corollary 3.7 The EP database is consistent.

Proof. It is clear from Theorem 3.5.

4 Applicative Structure

We will show in this section that a database is interpreted as a high-order function. An applicative structure is commonly used to interpret a Turing-equivalent language [3]. A class of total recursive functions is a strict subset of a class of partial recursive functions, and therefore it can be fit well into an applicative structure [2]. We will develop an applicative structure in this section such that each term of the EP database, under a database, is interpreted as an element in the applicative structure. Conversely, an element from the applicative structure can be expressed as a database. It is done by proving that the entire applicative structure can be expressed as a database in the next section.

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

Notation 4.1 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, that don't have assigners, and that are not the left sub-terms of any comb-terms in D, 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., $NF^+ = NF - NF^0 - NF^1$

Example 4.2 For the database $G = \{a, b, a\ b, c, 3, (a\ b\ c =: 3), d, a\ b\ d\}$ defined in Example 2.6.4:

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

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.

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 (4.5.6 of [3]):

Notation 4.3 # is an effective one-to-one map: $T - \{\texttt{null}\} \rightarrow \mathbb{N}$, i.e., given an element $m \in T - \{\texttt{null}\}$, we can find one and only one corresponding Gödel number $\# m$ in finite steps.

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 **i**.

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

Definition 4.4 $\mathbf{R}^0 = \mathbb{N} \cup \{\mathbf{i}, \perp\}$, where **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 4 or the article 1]. We simply denote such a class as $\mathbf{R}$, and represent it in a form of effective applicative structures.

Definition 4.5 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 better understand the applicative behavior of a total recursive function, we give an alternative notion of an element $\mathbf{f} \in \mathbf{R}$.

Notation 4.6 1. let $\mathbf{f} \in \mathbf{R}$, $\mathbf{f}$ is alternatively expressed as:

$\mathbf{f} = \{<\mathbf{e}, \mathbf{j}> \mid \mathbf{e} \in \mathbf{R} \cap \mathbf{f} * \mathbf{e} = \mathbf{j}\}$

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 4.7 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.,
   $\forall m \in NF^1, [m] = \{<\mathbf{i}, \#m>\} \cup \{<[o], \perp> \mid o \in T\}$,
4. A normal form in NF^+ is mapped to the function containing its syntactical information and mainly its derived information, i.e.,
   $\forall m \in NF^+, [m] = \{<\mathbf{i}, \#m>\}$
   $\cup \{<[n_i], [m\ n_i]> \mid$ for all $n_i \in T$, such that $m\ n_i \in D\}$
   $\cup \{<[o_i], \perp> \mid$ 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.,
   $\forall m \in T, [m] = [\text{nf}(m)]$.

Example 4.8 Give the database $G = \{a, a\ b, (a\ b\ c =: 3), a\ b\ d, b, c, d, 3\}$ defined in Example 2.6.4, here are sample mappings according to the rules above:

$[3] = \#3$;

$[b] = \{<\mathbf{i}, \#b>, <[0], \perp>, <[1], \perp>, \ldots, <[a], \perp>, <[b], \perp>, \ldots, <[elseTerm], \perp>, \ldots\}$;

$[a\ b\ d] = \{<\mathbf{i}, \#(a\ b\ d)>, <[0], \perp>, <[1], \perp>, \ldots, <[a], \perp>, <[b], \perp>, \ldots, <elseTerm, \perp>, \ldots\}$;

$[a\ b] = \{<\mathbf{i}, \#(a\ b)>, <[c], \#3>, <[0], \perp>, \ldots, <[elseTerm], \perp>, \ldots\}$;

$[a] = \{<\mathbf{i}, \#a>, <[b], [a\ b]>, <[0], \perp>, \ldots, <[elseTerm], \perp>, \ldots\}$;

Example 4.9 Given the database $DG = \{v1\ v2 =: v2;\ v2\ v1 =: v1\}$ defined in Example 2.6.6, here are a sample mappings according to the rules 4.7.5:

$[v1\ v2\ v1] = [v1]$.

In Definition 4.7.4 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 a strong induction, we assumed that n_i and $m\ n_i$ have their semantics $[n_i]$ and $[m\ n_i]$. Then the collection of all the elements $<[n_i], [m\ n_i]>$ is the derived semantics for m. To facilitate a further discussion in the section 6, we give a notation for the phrase "derived semantics":

Notation 4.10 The derived semantics of m is notated as: $\{<[n_i], [m\ n_i]> \mid$ for all $n_i \in T$, such that $m\ n_i \in D\}$.

For each normal form $m \in NF^1 \cup NF^+$, we added the syntactical information $\{<\mathbf{i}, \#m>\}$ to its semantics. This ensures that the normal form has a unique interpretation. There are cases in which an element $<[n_i], [m\ n_i]>$ 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 2.6.5, for example, both $[p]$ and $[q]$ would end up with the same semantics: $\{<[\#s], [\#1]>\}$ if $<\mathbf{i}, \#p>$ and $<\mathbf{i}, \#q>$ 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 yet to be a high-level function (a member in $\mathbf{R}$) until Theorem 4.14.

Lemma 4.11 $m \rightarrow_{\text{EP}} n \Rightarrow [m] \equiv [n]$

Proof. It is clear from Definition 4.7.

Notation 4.12 If a term m is reduced to its normal form n and n is not identical to another normal form k, then we write $m\ !\rightarrow_{\text{EP}} k$.

Lemma 4.13 $m\ !\rightarrow_{\text{EP}} n \Rightarrow [m]\ !\equiv [n]$.

Proof. See the Proof 6.10 in [9].

Theorem 4.14 (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. See the Proof 6.11 in [9].

From Lemmas 4.11 and 4.13, and Theorem 4.14, we have a sound interpretation that a database is semantically a high-level function. The soundness

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

Corollary 4.15 $[m\ n] = [m] * [n]$

Proof. See the Proof 6.12 in [9].

5 Property Enumeration

In Section 4, 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. It can be done 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 an 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 database, 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 database is equivalent to a class of total recursive functions.

Recall that we introduced an extra 0-ary function $\mathbf{i}$ in $\mathbf{R}$ in Section 6. This special constant is not necessary for this section, but it doesn't affect our conclusions by continuing to use $\mathbf{R}$. The set of constants over which a class of total recursive functions is constructed is inessential [2].

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

Notation 5.1 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 $\mathbf{R}$ are mapped to the identifiers P.

Lemma 5.2 $\forall\ \mathbf{a} \in \mathbf{R}, \|\lceil \mathbf{a} \rceil\| = 1$.

Proof. It is clear from 5.1.1 and 5.1.2.

Definition 5.3 1. Given a $\mathbf{m} \in \mathbf{R}$, let $\wp(\mathbf{m})$ be a set of assignments: $\wp(\mathbf{m})$ =

$\{\lceil \mathbf{m} \rceil \lceil \mathbf{n} \rceil =: \lceil \mathbf{o} \rceil \mid \mathbf{n} \in \mathbf{R}, \mathbf{m} * \mathbf{n} = \mathbf{o}, \mathbf{o} ! \equiv \bot\}$

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

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

To satisfy Property 2.7, 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 5.3.1 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 $\mathbf{R}$'s properties to a database; instead, we assume that $\mathbf{R}$'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 $\mathbf{R}$, a class of total recursive function, can be enumerated [the Corollary on page 169 of the text book 4 and the article 1].

Lemma 5.4 $\Im$ is an EP database.

Proof. See the Proof 7.4 in [9].

Lemma 5.5 $\forall \mathbf{m} \in \mathbf{R}, \lceil \mathbf{m} \rceil$ is a normal form in $\Im$

Proof. See the Proof 7.5 in [9].

Definition 5.6 $\forall \mathbf{m_1}, \mathbf{m_2}, \ldots, \mathbf{m_n} \in \mathbf{R}, \forall\ n \in \mathbb{N}$, 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 4.6.

Theorem 5.7 (completeness) Total recursive functions are *EP*-definable.

Proof. See the Proof 7.7 in [9].

6 Remarks

There are many language systems, such as the lambda calculus, the Turing machine, and the combinatory logic that are equivalent to classes of partial recursive functions [3]. The EP database is a language system that is equivalent to a class of total recursive functions. Obviously, the relational

database and the hierarchical database are the special cases. Our proof was accomplished by showing that a term in a database is a (high-order) function, and that the properties of a class of total recursive functions, presented in a certain format, form a database. This doesn't mean that all the software applications can be practically managed by using the EP database alone; rather, it offers a consistent and always-halting method to manage as much finite data as a business application requires and by doing so minimizes the effort of software development and maintenance. Enforcing software applications to be totally recursive is an essential factor that distinguishes a data model from a programming language.

A database is an evolving subset of a 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 a class of total recursive functions requires a discussion beyond the scope of this article. This issue can be addressed by expanding Section 5 with various applicative structures.

The EP data mode is a type-free system. Since it is totally recursive, the type-free doesn't cause exception or non-termination. Instead, we can use terms in the EP data model to easily express the functions that are infinitely high-order and at the same time totally recursive. A typical example of an infinitely high-order function is a self-reference function such as $F = \{<0, 1>, <F, 1>\}$, which is expressed as: $F = \{F\ 0 =: 1,\ F\ F =: 1\}$ in the EP database. Another example is Example 2.6.6: $DG = \{v1\ v2 =: v2;\ v2\ v1 =: v1\}$ for a directed graph with a circle.

We say that a function expressed in an EP database is in infinitely high-order when its derived information cannot be finitely expressed. The term *v1* in Example 2.6.6, as an example, was said being an infinitely high-order function because its derived information is:

{<[*v2*], [*v2*]>}
(-- by 4.7.4 and 4.7.6 due to *v1 v2 =: v2*)
= {<{... <[*v1*], [*v1*]> …}, {... <[*v1*], [*v1*]> …}>}
(-- by 4.7 due to *v2 v1 =: v1*)
…
= …

The deriving process above would never end due to a circle in the directed graph.

The infinitely high-order functions, a native subset of a class of total recursive function, caused us to use a strong induction in introducing Definition 4.7.4 and in proving Theorem 4.14. (That is, [*v1*] in Example 4.9, as an example, was proved to be a higher-order function by assuming that [*v2*] was a lower-order function while it was difficult to express and therefore to prove any 0-ary functions as the base of [*v2*].) Nevertheless, the usage of the strong induction didn't impact our results since it has the same effectiveness as a weak induction does.

Discussing the properties of infinitely high-order functions is not in the scope of this article. The authors would like to redirect readers to the topic of continuous functions or the function space initiated in the articles [5 and 6] for the concept of function limits, i.e., infinitely high-order functions, approximated by finitely high-order functions. It covers the complete information about how infinitely high-order functions co-exist with finitely high-order functions in a class of total (and further partial) recursive functions.

The EP database 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 database (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 database.

REFERENCES:

1 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.

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

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

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

5 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.

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

7 K. H. Xu, J. Zhang, S. Gao. "An Assessment on the Easiness of Computer Languages". To appear in the Journal of Information Technology Review, 2010.

8 K. H. Xu, Jingsong Zhang, Shelby Gao. "Consolidations Following Froglingo, An Alternative to DBMS, Programming Language, Web Server, and File System". Submitted to the

5th International Conference on Evaluation of Novel Approaches to Software Engineering", 2010.

9 K. H. Xu, J. Zhang, S. Gao, R. R. McKeown. "Data Model and Total Recursive Functions". Technical Report 2009-11, http://www.froglingo.com/TR200911.pdf.

10 K. H. Xu, J. Zhang, S. Gao. "Assessing Easiness with Froglingo". The Second International Conference on the Application of Digital Information and Web Technologies, 2009, page 847 - 849.

11 K. H. Xu, J. Zhang. "A User's Guide to Froglingo, An alternative to DBMS, Programming Language, Web Server, and File System". Available at the website: http://www.froglingo.com/FrogUserGuide10.doc.

12 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.

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

14 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.