The Enterprise-Participant (EP) Model, a Untyped and Recursive Language Semantically Approximating the Lambda Calculus

Kevin H. Xu
kevin@froglingo.com

Abstract. Many recursive languages, e.g., the simply typed lambda calculus in which computations always terminate, define infinite functions, i.e., with infinitely many ordered pairs, that don't include self application functions, i.e., applying a function to itself is allowed. In this article, we introduce the Enterprise-Participant (EP) data model, a recursive language that defines bounded functions, i.e., the co-domains are finite while domains are possibly infinite. The bounded functions include self-application functions. The notion self application is a synonym of the notion untyped in programming languages. Self application and untyped are normally associated together with the notion of non-termination, because we put ourselves in the context of programming languages that define partial computable functions. By limiting ourselves to the bounded functions in this article, we show that an untyped language doesn't have to go together with non-termination and can be practically useful in database management.

Keywords. The lambda Calculus, Partial Computation (Finite Apprxomiation), Partial Continuous Functions, Computability, Expressiveness, Data Structure.

1 Introduction

We have been trying to avoid non-terminating computations by using recursive languages, such as the simply typed lambda calculus, that don't express self-applications. To make programming language fully expressible, however, we always include fixed-point combinators that bring in fixed points including self-applications and at the same time bring in non-terminating computations. If we have a recursive language that expresses some of the fixed points and at the same time guarantees terminating computations, we can improve software development productivity by using the recursive language. The Enterprise-participant data model, abbreviated as EP, is such as a recursive language for database management.

Example 1.1 The set $\{x\ x := x\}$ is an EP database, where x can be viewed as a function that yields to itself when it is applied to itself, and yields to *null* (a special constant in EP) when it is applied to other arguments. Given the set $\{x\ x := x\}$, there are infinitely many expressions x, $x\ x$, $x\ x\ x$, $x\ x\ x\ x$, … that are reducible to x. The letter x actually represents an approximation to all the functions that yield to themselves after being applied to themselves, e.g., the identify function $\lambda x.x$.

Example 1.2 A directed graph with connections from v_1 to v_2, from v_2 to v_1, and from v_2 to v_3 can be expressed in an EP database: $\{v_1\ v_2 := v_2;\ \ v_2\ v_1 := v_1;\ \ v_2\ v_3 := v_3\}$,

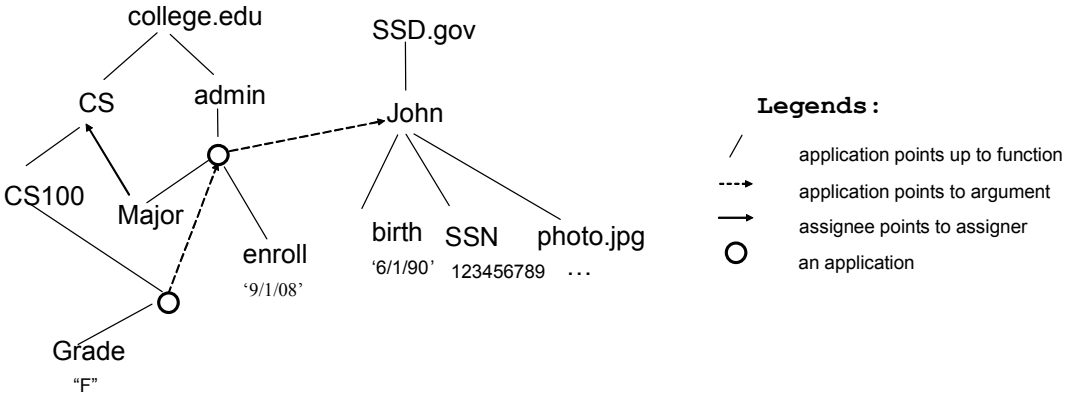
where v_1, v_2, and v_3 represent vertices, and an assignment represents a directed connection. In this example, the vertex v_1 is viewed as a function that yields to v_2 when it is applied to v_2, and yields to *null* when it is applied to others. The function v_2 yields to v_1 when it is applied to v_1, v_3 when it is applied to v_3, and *null* otherwise. The vertex v_3 is also a function, but a constant one for the time being. It may become a non-constant function later with an EP update operation. With this database, we can reduce infinitely many expressions $v_1\ v_2$, $v_1\ v_2\ v_1$, …, $v_1\ v_2\ v_1$ ... v_1, … to v_1. It could be thought of as if one walked along the circle from v_1 to v_2 and from v_2 to v_1 as many times as she liked and eventually stopped at the vertex v_1. The query "is there a path from v_1 to v_3?" is simply expressed in EP as v_3 <=+ v_1, where the binary operator <=+ reflects a pre-ordering relation among functions, arguments, and values that exist in a database. The answer would be the truth value *true*.

By adding constants, such as integers, strings, and truth values, the EP data model can express common business data in software development practice.

Example 1.3 A school administration can be expressed in the following database:

SSD.gov John SSN := 123456789;
SSD.gov John birth := '6/1/1990';
SSD.gov John photo.jpg := ...; /* a binary stream for a photo*/
college.edu admin (SSD.gov John) enroll := '9/1/2008';
college.edu admin (SSD.gov John) Major := college.edu CS;
college.edu CS CS100 (college.edu admin (SSD.gov John)) grade := "F";

This database can be alternatively expressed in a diagram:



In this article, we introduce the EP data model, a recursive language that defines EP databases and a set of built-in operations over the databases. An EP database defines a finite set of functions, where each function is finite, i.e., finitely many ordered argument-value pairs. When applying a built-in operator to an EP database, however, the finite functions in a database are expanded potentially to be infinite. Put differently,

an EP database under the build-in operator as a whole actually defines a bounded function, i.e., the domain is infinite while the co-domain is finite. Such bounded functions actually include self application functions. (The built-in operator is a constant function – the application that applies an EP term to another, in an analogy to the application operation of the lambda calculus. We will discuss it in detail later.)

To better understand what the bounded functions are in a relation with computability, we develop an approach of partial computations for the lambda calculus, in which a partial computation enumerates a finite set of properties from the semantics of the lambda calculus. Recall that the properties of partial computable functions repeat themselves in an enumeration. Therefore the finite set of properties in the discussion contains redundant information. We refine the finite set of properties to exclude redundant information before being transformed to an EP database. Conversely from an EP database, we show that the finite set of properties is recoverable by applying the built-in application operator. In addition, the built-in application operator actually defines, under a given database, a bounded function, which approximates the semantics of the lambda calculus as well.

Studying bounded functions definable in the EP data model pertains to have the following objectives:

1) The nature of being untyped of the lambda calculus, i.e., being without ground level objects such as integers or allowing self applications, is captured in the bounded functions and is found practically useful through the EP data model. It is a contrast to the common view that a system without ground level objects [4] or having self-applications [3] is not practically useful and a contrast to contemporary language systems where ground level objects normally are in the place before more complex and meaningful objects can be constructed.
2) Provided that all data desired to be managed in a database is bounded functions definable in the EP data model, the EP data model is a mathematical underpinning universally for arbitrary data management and data exchange. The desire of managing the bounded functions for business data is quite reasonable because all a computer can do is to produce finite approximations to partial computable functions. Even more, the desire is a little luxurious because bounded functions are more than finite approximations.
3) With the availability of Turing-machine equivalent programming languages, a data management system equivalent to the EP data model may not be welcomed in practice if it forces users to manage redundant data, i.e., repeatedly store a function multiple times. By relating EP databases with finite approximations where a function many repeatedly appear in the approximations, the EP data model is allows users to manage business data without data redundancy.

In Section 2, we introduce the EP data model. In Section 3, we introduce an approach to partial computations of the lambda calculus. The development of the partial computations is aimed to describe the EP data model in the lambda calculus. In Section 4, we exclude recoverable (redundant) data from finite approximations produced by partial computations. Further the refined finite approximations are transformed to EP databases, and therefore we show that every finite approximation can be defined in an EP database. In Section 5, we give an algorithm of enumerating all EP databases.

With the enumeration, we can show that the EP data model, as an independent language, does nothing else but approximate the denotational semantics of the lambda calculus. In Section 6, we briefly relate the EP data model with other approaches of approximating partial computable (continuous) functions, with which we can compare the EP data model with purely typed language systems. We also question if there is a more expressive recursive language than the EP data model that would be interesting in database management.

In the article, the proofs to most of the lemmas, corollary, and theorems are provided in an appendix at the end.

2 The EP Data Model

Rather than defining functions in formulas or algorithms, the EP data model defines functions in ordered argument-value pairs. A collection of ordered pairs, in the form of assignments, is called a database. Starting from a set of identifiers, we construct a set of EP terms.

Definition 2.1 Given a set of constants $\boldsymbol{C}$ including a special constant *null* and a set of (functional) identifiers $\boldsymbol{F}$, the set of *EP terms*, denoted as $\boldsymbol{E}$, is defined inductively[*] as follows:

$$x \in \boldsymbol{C} \cup \boldsymbol{F} \Rightarrow x \in \boldsymbol{E}$$
$$x, y \in \boldsymbol{E} \Rightarrow x\ y \in \boldsymbol{E}$$

We adopt many notations from the lambda calculus, including *sub-term*[†] (SUB(t) for all sub-terms in t), *left-most*[‡] sub-term (LMS (t) for all left-most sub-terms in t), and applications without surrounding parentheses, e.g., $a\ b$, instead of ($a\ b$) when there is no ambiguity. Given a term $a\ b\ (c\ d)$, for example, SUB ($a\ b\ (c\ d)$) = {$a\ b\ (c\ d)$, $a\ b$, $c\ d$, a, b, c, d}, LMS ($a\ b\ (c\ d)$) = {$a\ b\ (c\ d)$, $a\ b$, a}. Given a term t, we further call a sub-term of t, except for t itself, a *proper* sub-term, and therefore we use $\mathrm{SUB}^+(t)$ to denote all the proper sub-terms of t, i.e., $\mathrm{SUB}^+(t) = \mathrm{SUB}(t) - \{t\}$.

An EP database (or simply a database) is a finite set of assignments where assignees and assigners are terms. Some terms that have a constant as a left sub term, e.g., $3\ t$, are not allowed to be in a database. We identify a sub set out of $\boldsymbol{E}$, denoted as $\boldsymbol{E^0}$, which is inductively defined as:

$$x \in \boldsymbol{F} \Rightarrow x \in \boldsymbol{E^0}$$
$$x \in \boldsymbol{E^0}, (y \in \boldsymbol{C} \cup \boldsymbol{E^0}) \Rightarrow x\ y \in \boldsymbol{E^0}$$

It means that if an application $a\ b$ is a term in $\boldsymbol{E^0}$, then the left sub term a cannot be a constant.

An *assignment* is in the form of

---

[*] When an inductive definition is given, it will always be understood that the class defined is the smallest set satisfying the conditions. It is applied to other inductive definitions introduced later in the article.

[†] Precisely, $t \in \Lambda \Rightarrow t \in \mathrm{SUB}(t)$, and $m\ n \in \mathrm{SUB}(t) \Rightarrow m, n \in \mathrm{SUB}(t)$.

[‡] Precisely, $t \in \Lambda \Rightarrow t \in \mathrm{LMS}(t)$, and $m\ n \in \mathrm{LMS}(t) \Rightarrow m \in \mathrm{LMS}(t)$.

$$a := b$$

where $a, b \in \boldsymbol{E^0}$, called the assignee and the assigner respectively.

Definition 2.2 A *database D* is a finite set of assignments with the following constraints:

1) Each assignee has only one assigner, i.e.,

$$a := b_1 \text{ and } a := b_2 \in D \Rightarrow b_1 \equiv b_2$$

2) A proper sub term of an assignee cannot be an assignee, i.e.,

$$a := b \in D \Rightarrow \forall x \in \text{SUB}^+(a)\ [\forall c \in \boldsymbol{E^0}\ [x := c \notin D]]$$

3) An assigner is a non-*null* constant, an identifier, or a proper sub-term of an assignee, i.e.,

$$a := b \in D \Rightarrow b \in \boldsymbol{C} - null, b \in \boldsymbol{F}, \text{ or } \exists c, d \in \boldsymbol{E^0}\ [c := d \in D \text{ and } b \in \text{SUB}^+(c)]$$

4) If there is a sequence of assignments $a_0 := a_1, a_1 := a_2, \ldots, a_{j-1} := a_n \in D$ for a $n \geq 1$, then a_n must not be identical to a_0, i.e.,

$$a_0 := a_1, a_1 := a_2, \ldots, a_{j-1} := a_n \in D \Rightarrow a_0 \neq a_n$$

The system being discussed in this article was originally called the Enterprise-Participant data model [8], where given an application $a\ b \in \boldsymbol{E}$, a was meant an enterprise and b a participant. The notion term in Definition 2.1 says a lot about how objects in the real world interact to each others, such as a person participates in a party. The constraints in Definition 2.2 require that a participation has a outcome, i.e., applying a function to an argument yields a value. Put it differently, a database is a collection of functions, each of which has a finite set of triplets function-argument-value, i.e., $a\ b := c$ where a, b, c are respectively a function, an argument, and the corresponding value. An assignee is normally an application, and an assigner is a value. The first constraint ensures that each application yields only one value. The second constraint prevents two equivalent applications from being assigned with multiple values. Given assignments $a_0 \ldots a_i \ldots a_k := b$, $a_i := c \in D$, and $b \neq e$, for example, we prevent the assignment $a_0 \ldots c \ldots a_k := e$ from being in D.

The value of an application is a term which is in turn interpreted as a function as well. We could choose any term in $\boldsymbol{E}$ to be a value along with an adjustment to the definition 2.2. But we limit it by the third constraint in an ultimate form desired to represent a value. The ultimate form is called a normal form.

As a notation, we say that terms a and b are in database, denoted as $a, b \in D$, if $a := b \in D$. Similarly, we say that b is in database, denoted as $b \in D$, when there is a term $a \in D$ and $b \in \text{SUB}(a)$.

Definition 2.3 Given a database D, a term $n \in \boldsymbol{E}$ is a *normal form* in D if it is a constant or a term in D except for an assignee in D, i.e., $n \in \boldsymbol{C}$, or $n \in D$ and $\forall b \in \boldsymbol{E^0}\ [n := b \notin D]$.

In other words, a normal form is a constant, an identifier in D, or a proper sub-term of an assignee, i.e., $n \in \boldsymbol{C}$, $n \in F \cap D$, or $\exists a, b \in \boldsymbol{E^0}\ [a := b \in D \text{ and } n \in \text{SUB}^+(a)]$. For

example, the terms x, v_1, v_2, v_3, *John*, *SSD.gov John* are normal forms of the example databases in Section 1. We use NF^D to denote all normal forms with a database D. Limiting values to be in formal forms doesn't limit the expressiveness of the EP data model, as we will see more in detail in coming sections. Instead, it has many benefits: help users in finding typos during development, improve system performance, and simplify the theoretical discussion of the EP data model.

Now, we want every term in $\boldsymbol{E}$ to have a value. This becomes possible by introducing a set of reduction rules.

Definition 2.4 Given a database D, we have one-step *reduction* rules, denoted as $\rightarrow$:
1. An assignee is reduced to the assigner, i.e.,

$$a := b \in D \Rightarrow a \rightarrow b$$

2. An identifier not in database is reduced to *null*, i.e.,

$$a \in \boldsymbol{F}, a \notin \boldsymbol{D} \Rightarrow a \rightarrow null$$

3. If a and b are normal forms and $a\ b \notin D$, then $a\ b$ is reduced to *null*, i.e.,

$$a, b \in NF^D, a\ b \notin \boldsymbol{D} \Rightarrow a\ b \rightarrow null$$

4. $a \rightarrow a'$, $b \rightarrow b' \Rightarrow a\ b \rightarrow a'\ b'$.

Definition 2.5 Let $a \rightarrow a_0, \ldots, a_{n-1} \rightarrow a_n$ for a number $n \in \mathbf{N}$. Then we say that a is effectively, i.e., in finite steps, reduced to a_n, denoted as $a \rightarrow_{EP} a_n$.

Definition 2.6 A term a has a normal form b if b is a normal form and $a \rightarrow_{EP} b$.

If $a_1 \rightarrow_{EP} b$ and $a_2 \rightarrow_{EP} b$, then we say that b, a_1 and a_2 are equal, denoted as $b == a_1 == a_2$.

Here are a few sample equations under the databases of Section 1:

SSD.gov John SSN == 123456789
college.edu admin (*SSD.gov John*) *Major == college.edu CS*
$v_1\ v_2\ v_1 == v_1$
$v_1\ v_2\ v_1\ v_2\ v_1 \ldots\ v_1 == v_1$
$x\ x\ x == x$

With the reduction rules, we want the value of an assignee to be the value of its assigner. Note that a reduction may not terminate if there is a set of circular assignments, i.e., there are a chain of assignment $a_0 := a_1, a_1 := a_2, \ldots, a_{j-1} := a_n \in D$ for a $n \geq 1$ such that $a_0 := a_1, a_1 := a_2, \ldots, a_{j-1} := a_n \in D$, and $a_0 \equiv a_n$. The constraint 2.2.4 ensures that all reductions terminate, which lead to the following conclusion: a term has one and only one normal form, i.e., it is strongly normalizing.

Lemma 2.7 An assignee has a unique normal form in a database.

Theorem 2.8 A term $a \in \boldsymbol{E}$ has one and only one normal form under a database D.

Our discussion has emphasized that under a database D, a term a has a normal form as its value, either a constant, an identifier in D, or a proper sub-term of an assignee. In comparing with the contemporary programming language and database management

systems, it is easy to understand that *a* takes a constant as its value, or *a* takes a proper sub-term of an assignee which is not *a* itself as its value. One difference in the EP data model is that a proper sub term of an assignee, say *a* as a term in ***E***, takes itself as its normal form. It has been a misleading in this article so far to say that *a* takes *a* itself as it's own value. Precisely, we say that *a* is a name to *a*'s value which is not explicitly given in *D* but implicitly derivable from *D*. For example, the term *SSD.gov John* is a normal form in the database of Example 1.3. Rather than saying that *SSD.gov John* has a value *SSD.gov John*, we should have said that it has a value of a function: {(*birth*, '*6/1/90*'), (*SSN*, *123456789*), (*photo.jpg*, '...')}, here '…' stands for a binary stream as the content of a file named photo.jpg. Allowing a proper sub term of an assignee to take itself as its normal form and to reference an implicit value provides a syntactical mean to express meaningful applications without constants and to express self applications in the EP data model. That is the distinguishing feature making the EP data model untyped.

In the discussion so far, we have said that an assignee is normally an application. The definition 2.2 also allows an identifier to be an assignee, which is consistent with the lambda calculus and its partial computations to be discussed in coming sections. The definition 2.2.3 doesn't allow *null* to be an assigner to prevent users from entering meaningless assignments into databases because any meaningless terms is reduced to *null* according to 2.4.

Before ending this section, we show that given a database *D*, the EP data model defines a total function with a bounded support. A function $f : X \to Y$, where X and Y are arbitrary sets of objects, has a *finite support* if and only if there exists a finite set $A \subset X$ and a member $a \in Y$ such that

$$f(x) = \begin{cases} b, \text{ where } b \in Y \text{ and } b \neq a & \text{if } x \in A \\ a & \text{if } x \notin A \end{cases}$$

A function $f : X \to Y$, where X and Y are arbitrary sets of objects, has a *bounded* support, if and only if there exists a finite set $A \subset Y$ such that

$$f(x) \in A \text{ for all } x \in X.$$

In this article, we simply call a function finite if it has a finite support, and bounded if it has a bounded support. A finite function is bounded, but a bounded function may not be finite.

Lemma 2.9. Given a database *D*, the set of all non-constant normal forms, i.e., $NF^D - \boldsymbol{C}$, is finite.

Given a database *D* and a term $m \in \boldsymbol{E}$, we use m^{nf} to denote *m*'s normal form.

Theorem 2.10 Given a database *D*, there exists a computable function Y^D: $\boldsymbol{E} \to \boldsymbol{E}$ such that

$$Y^D(m) = m^{\text{nf}}$$

for all $m, n \in \boldsymbol{E}$.

Theorem 2.11 Given a database *D*, there exists a finite function X^D: $D \to NF^D$ such that

$$X^D(m) = m^{\mathrm{nf}}$$

for all $m \in \boldsymbol{E}$ and $m \in D$.

Theorem 2.12 If there exists a sequence of assignments $a_0 := a_1$, $a_1 := a_2$, …, $a_{j-1} := a_n \in D$ for a $n \geq 1$ such that $a_0^{\mathrm{nf}} \neq null$ and $a_n \in \mathrm{SUB}(a_0)$, then Y^{D} is not finite but bounded.

The application function that applies an *EP* term to another, e.g., $a\ b \in \boldsymbol{E}$ for arbitrary $a, b \in \boldsymbol{E}$, is the constant application operator we discussed in Section 1. There are many more operators including <=+ mentioned in Example 1.2 that stem from the relationships between functions, argument, and value and are not discussed here. For more information, please reference [6] and [7].

3 Partial Computations of the Lambda Calculus

In Section 2, we showed that EP terms are structurally similar to lambda applications and the EP data model defines bounded functions including self application functions. In this section, we develop a partial computation of the lambda calculus which enumerates finitely many closed lambda terms. Given a closed lambda term[§], all terms that share the same term as the left most sub term and have normal forms are said in this article to be the properties of the given term. A partial computation produces a finite set of such properties which approximate and converge to the whole properties of a given term. Finite approximations as syntactical objects, i.e., a finite set of properties from partial computations, are refined to exclude redundant information, and eventually expressed in an EP database. As a result, we say that an EP database approximates the properties of a lambda term, or simply we say that an EP database approximate a lambda term semantically.

Through discussing partial computations of the lambda calculus, we are also able to relate the syntax of the EP terms with the syntactical structure of lambda applications. It further helps us to better understand what the EP data model is exactly. In this section, we introduce the approach of partial computations of the lambda calculus. We will see how we transform a finite approximation to an EP database in the coming section.

Given a *n*-ary number-theoretical partial recursive function φ_e: $\mathbf{N}^n \rightarrow \mathbf{N}$, e.g., in the form of Kleene's systems of equations, and a number $s \in \mathbf{N}$, i.e., $\{0, 1, \ldots\}$, for a finite computational steps, we have a partial computation that produces a finite approximation $\varphi_{e,s}$ to φ_e such that:

$$\varphi_{e,s}(x_1, x_2, \ldots, x_n) = y \quad \text{if } x_1, \ldots, x_n, y \leq s \text{ and } \varphi_e(x_1, x_2, \ldots, x_n) \downarrow_s y$$

Here $\varphi_e(x_1, x_2, \ldots, x_n) \downarrow_s y$ denotes that $\varphi_e(x_1, x_2, \ldots, x_n)$ converges within s steps and the output is y. (In other words, there exists a Turing program P for computing φ_e that

[§] Note that we only consider closed lambda terms that sufficiently gives us a full account of partial continuous functions to be discussed in this article.

terminates within s steps and produces the output y.) As a finite set, the approximation $\varphi_{e,\,s}$ can be rewritten as a finite set of n-ary tuples:

$$\varphi_{e,\,s} = \{<(x_1, x_2, \ldots, x_n), y> \mid x_1, \ldots, x_n, y \leq s \text{ and } \varphi_e(x_1, x_2, \ldots, x_n) \downarrow_s y\}$$

and the union of all approximations is φ_e itself:

$$\varphi_e = \cup_{s \in \mathbf{N}} \varphi_{e,\,s}$$

Given a closed lambda term $M_0 \in \Lambda^0$ in the lambda calculus, we would like to have a similar partial computation for M_0. It comes with a few adjustments. First of all, we need to consider all the terms that have M as a left most sub term, i.e.,, M_0, $M_0\, M_1$, …, $M_0\, M_1\, M_2 \ldots M_n$, … for all $M_1, M_2, \ldots M_n, \ldots \in \Lambda^0$, in contrast to a fixed number n. Formally, we define a set as the complete properties of a lambda term M_0:

Definition 3.1 Given $M_0 \in \Lambda^0$, the set $[M_0]$ is defined as:

$$[M_0] = \{(M_0\, M_1 \ldots M_n, Q) \mid n \geq 0;\, M_1, \ldots, M_n \in \Lambda^0;\, \text{and } M_0 M_1 \ldots M_n \downarrow Q\}$$

here we use $M_0\, M_1 \ldots M_n \downarrow Q$ to denote that $M_0\, M_1 \ldots M_n$ has a normal form $Q \in \Lambda^0$.

The set $[M_0]$ considers all terms that vary in their sizes, i.e., the number n in $M_0\, M_1 \ldots M_n$ varies from 0 to any number in **N**. It is necessary because of head normal forms. Recall that a term M_0 has a head normal form if and only if there exist $M_1, \ldots, M_n \in \Lambda^0$, where $n \geq 0$, such that $M_0\, M_1 \ldots M_n$ has a normal form [4]. If all terms $M_0\, M_1 \ldots M_n$ with a fixed n don't have normal forms, we would not be able to expose potential normal forms of applications $M_0\, M_1 \ldots M_n\ M_{n+1} \ldots M_{n+i}$ for some $i > 0$.

Even if a normal form has been found for a term $M_0\, M_1 \ldots M_n$, we continue to count terms $M_0\, M_1 \ldots M_n\ M_{n+1} \ldots M_{n+i}$ for $i > 0$ in the set $[M_0]$ as long as they have normal forms. Let $I = \lambda x.x$, for example, all the pairs $(I\, I, I)$, $(I\, I\, I, I)$, … will be in $[I]$. We are interested in all terms that have M_0 as a left most sub terms when we study the properties of M_0, particularly in studying the relationship of the EP data model with the lambda calculus.

To give a partial computation, we assume (and we know it is possible) that all terms in Λ^0 are in a sequence, e.g., $N_0, N_1, \ldots$. Given $M \in \Lambda^0$, we use $\#M$ to denote its index in the sequence. Using Contor's diagonal method, we have the following definition:

Definition 3.2 Given $M_0 \in \Lambda^0$ and $s \in \mathbf{N}$, the set $[M_0]_s$ is defined as:

$$[M_0]_s = \{(M_0\, M_1 \ldots M_n, Q) \mid n \geq 0;\, n, \#M_1, \ldots, \#M_n, \#Q \leq s;\, M_0\, M_1 \ldots M_n \downarrow_s Q\}$$

here we use $M_0\, M_1 \ldots M_n \downarrow_s Q$ to denote that $M_0\, M_1 \ldots M_n$ is reduced to a normal form Q within s steps and further $M_0\, M_1 \ldots M_n \neq Q$.

Note that if $M_0\, M_1 \ldots M_n$ itself is a normal form (in this case, n must be 0), the pair (Q, Q) is not included in the set $[M_0]_s$. There is no need to keep (Q, Q) in the set, and the corresponding EP data model doesn't allow it as well.

In the definition above and in the rest of the article, the symbol $\neq$ is always meant to "is not identical",

To come up with a truly finite set in which each element is also finite as an approximation to a partial continuous function, we need another adjustment to the ap-

proach of the approximations of partial recursive functions. Recall that x_i for $1 \le i \le n$ and y in $\varphi_{e,\,s}$ are numbers, ground level values, that cannot be further approximated by other objects. Therefore $\varphi_{e,\,s}$ is in fact a finite approximation. Terms M_1, …, M_n, and P in $[M_0]_s$, however, don't have their own finite approximations because they represent infinite functions. Therefore $[M_0]_s$ is not a truly finite approximation to partial continuous functions. In stead of enumerating the properties of just one term, we enumerate all terms $M \in \Lambda^0$:

Definition 3.3 Given $s \in \mathbf{N}$, the set, denoted as $[\Lambda^0]_s$, is defined as

$$[\Lambda^0]_s = \cup_{\#M \le s} [M]_s$$

Writing differently, we have

$$[\Lambda^0]_s = \{(M_0\ M_1\ \ldots\ M_n, Q) \mid n \ge 0;\ n, \#M_0, \#M_1, \ldots, \#M_n, \#Q \le s;\ M_0\ M_1\ \ldots\ M_n \downarrow_s Q\}$$

Definition 3.4 The properties of all the closed lambda terms can be represented in a set $[\Lambda^0]$ such that

$$[\Lambda^0] = \{(M_0\ M_1\ \ldots\ M_n, Q) \mid n \ge 0;\ M_0, M_1, \ldots, M_n \in \Lambda^0;\ \text{and}\ M\ M_1\ \ldots\ M_n \downarrow Q\ \}$$

Theorem 3.5 Given $s \in \mathbf{N}$,

1) $[\Lambda^0]_s$ is finite and recursive.
2) $[\Lambda^0]_s \subseteq [\Lambda^0]_{s+1}$.
3) $[\Lambda^0] = \cup_{s \in \mathbf{N}} [\Lambda^0]_s$

Proof. The results are clear from the definitions 3.2, 3.3, and 3.4 themselves.

4 EP Databases Defining Approximations

With a set $[\Lambda^0]_s$, our attention is no longer with lambda terms and their reductions, e.g., M_0, M_1, …, M_n, and Q in $[\Lambda^0]_s$, but relationships among them, e.g., how M_i for an $i \le n$ and sub terms of M_i are related to terms $M_0\ M_1\ \ldots\ M_n$ and Q. As a matter of fact, each lambda term, as far as the set $[\Lambda^0]_s$ itself is concerned, doesn't represent its own semantics any longer, but acts as a name (an identifier) representing an approximation to its semantics. In other words, the relationships among the terms in $[\Lambda^0]_s$ are preserved if the lambda terms, e.g., M_0, M_1, …, M_n, and Q, are substituted with identifiers, e.g., those from $\boldsymbol{F}$ in the EP data model. The set

$$[\Lambda^0]_s = \{(I\ I, I), (I\ W, W), (W\ I, I)\}$$

where $I \equiv \lambda x.x$ and $W \equiv \lambda x.\ x\ x$, for example, is equivalent to an EP database

$$\{a\ a := a;\ a\ b := b;\ b\ a := a\}$$

in the sense that two sets give the same relationships among two objects.

The relationships among the terms in $[\Lambda^0]_s$ are represented in the EP data model. First we convert a refined subset of $[\Lambda^0]_s$ into an EP database. In Section 3, we allowed two terms sharing the same left most sub terms to be in $[\Lambda^0]_s$ as long as they have normal forms, i.e.,

$$(M_0\, M_1 \ldots M_n,\ Q) \in [\Lambda^0]_s,$$

$$(M_0\, M_1 \ldots M_n\, M_{n+1\ldots}M_{n+i},\ Q') \in [\Lambda^0]_s \text{ for some } i > 0.$$

The lambda calculus tells us that there exists another expression $Q\ M_{n+1\ \ldots}\ M_{n+i}$ such that it can be reduced to Q'. Therefore we may have a third pair in $[\Lambda^0]_s$, i.e.,

$$(Q\ M_{n+1\ \ldots}M_{n+I},\ Q') \in [\Lambda^0]_s \text{ for some } s \in \mathbf{N}.$$

To fit $[\Lambda^0]_s$ in an EP database, we need to exclude the second pair from $[\Lambda^0]_s$ due to the constraint posted in Definition 2.2.2. On the other hand, the resulting set, denoted as $[\Lambda^0]_s^-$, contains enough information to recover the second pair by using the EP data model when $[\Lambda^0]_s^-$ is converted to an EP database.

Definition 4.1 Given $s \in \mathbf{N}$, the set, denoted as $[\Lambda^0]_s^-$, is defined as

$$[\Lambda^0]_s^- = \{(M_0 \ldots M_n, Q) \mid n \geq 0;\ n, \#M_0, \ldots, \#M_n, \#Q \leq s;\ M_0 \ldots M_n \downarrow_s Q;\ \text{and}$$
$$\forall x \in \text{SUB+}(M_0 \ldots M_n)\ [x \uparrow_s] \}$$

here $x \uparrow_s$ is to denote that x is either a normal form, or after s-step reduction, x is still not reduced to a normal form.

Now we are ready to replace λ-terms in $[\Lambda^0]_s^-$ with some identifiers in $\boldsymbol{F}$. We let the set of identifiers in $\boldsymbol{F}$ to be bijective to the closed terms in Λ^0. Given an identifier $a \in \boldsymbol{F}$, we used $\#a$ to denote its index in the sequence $a_0, a_1, \ldots$ of $\boldsymbol{F}$. Given an identifier $m \in \boldsymbol{F}$ and a closed term $M \in \Lambda^0$, m is said to be the identifier for the term M if $\#M = \#m$. In the discussion, we always use capital letters to represent closed lambda terms, e.g., $M \in \Lambda^0$, and small letters to represent identifiers, e.g., $m \in \boldsymbol{F}$. Given a letter with different cases, e.g., m and M, we imply that $\#M = \#m$.

Definition 4.2 Given a set of pairs of closed lambda term S, a new set, denoted as $S(\boldsymbol{F}/\Lambda^0)$, is obtained by replacing closed terms in S with their corresponding identifiers in $\boldsymbol{F}$ and rewriting a pair as an assignment, that is

$$S(\boldsymbol{F}/\Lambda^0) = \{m_0 \ldots m_i := q \mid (M_0 \ldots M_n, Q) \in S\}$$

A set S in the definition above can be $[\Lambda^0]$, $[\Lambda^0]_s$, or $[\Lambda^0]_s^-$. We may rewrite $[\Lambda^0]_s$, and $[\Lambda^0]_s^-$ as

$$[\Lambda^0]_s(\boldsymbol{F}/\Lambda^0) = \{m_0 \ldots m_i := q \mid n \geq 0;\ n, \#m_0, \ldots, \#m_i, \#q \in s;\ M_0 \ldots M_n \downarrow_s Q\}$$

$$[\Lambda^0]_s^-(\boldsymbol{F}/\Lambda^0) = \{m_0 \ldots m_i := q \mid n \geq 0;\ n, \#m_0, \ldots, \#m_i, \#q \in s;\ M_0 \ldots M_n \downarrow_s Q;\ \text{and}$$
$$\forall x \in \text{SUB+}(M_0 \ldots M_n)\ [x \uparrow_s]\}$$

Now we prove that a $[\Lambda^0]_s^-(\boldsymbol{F}/\Lambda^0)$ is an EP database.

Theorem 4.3 $[\Lambda^0]_s^-(\boldsymbol{F}/\Lambda^0)$ is an EP database.

From the proof above, we see that Q in a pair $(M_0 \ldots M_i \ldots M_n,\ Q) \in [\Lambda^0]_s$ or $[\Lambda^0]_s^-$ is always a normal form. Therefore the corresponding q in the database $[\Lambda^0]_s^-(\boldsymbol{F}/\Lambda^0)$ is always an identifier. It tells us that the constraint of 2.2.3 that doesn't allow arbitrary

terms but a limited set of terms including identifiers to be assigners doesn't impact the expressiveness of the EP data model.

In 2.2.3, we allow an application to be an assigner as long as it is a proper sub-term of an assignee, e.g., if $a\ (b\ c) := d$ is in a database, then $e := b\ c$ is allowed to be in a database. Actually it is a way to maximize redundancy reduction by more closely following the process of partial computations. At the above, we refined a finite approximation $[\Lambda^0]_s$ to $[\Lambda^0]_s^-$ by excluding a pair $(M_0 \ldots M_i \ldots M_n,\ Q) \in [\Lambda^0]_s$ from $[\Lambda^0]_s^-$ if there is another pair $(M_0 \ldots N_i \ldots M_n,\ Q) \in [\Lambda^0]_s$ such that $(M_i, N_i) \in [\Lambda^0]_s$ for an $i \geq 0$, where N_i is a normal form and M_i is not. Actually the same redundancy reduction can be done even if N_i is not a normal form but as long as it is reducible from M_i with s steps. The resulting databases will have applications as assigners. As a matter of fact, the resulting databases utilize the full syntactical flexibilities given in Definition 2.2.3. (Note that constants as assigners are not discussed here because we excluded constants as part of the discussion starting from Section 3).

Another issue that is worth to mention but we don't have space to elaborate here is also related to applications as assigners. When an assigner is an application, we may face multiple choices in converting a lambda term Q to an EP term q in Definition 4.2 where the definition $S\ (\boldsymbol{F}/\Lambda^0) = \{m_0 \ldots m_i := q \mid (M_0 \ldots M_n, Q) \in S\}$ is given. Taking Q as $M\ N$, for example, q can be $m\ n$ or an identifier, say $p \in \boldsymbol{F}$, such that $\#p = \#(M\ N)$. Nevertheless, it makes no difference in choosing either $m\ n$ or p in $[\Lambda^0]_s\ (\boldsymbol{F}/\Lambda^0)$ because the semantics of p and $m\ n$ converge to be identical when the number of the computation steps s approaches infinite.

In the rest of the section, we show that the EP data model is as informative as a Turing-machine equivalent programming language, i.e., all finite approximations of partial computable functions can be defined in the EP data model.

Definition 4.4 Given a database D, a function Z is defined as

$$Z(D) = \{(m, n) \mid m \in \boldsymbol{E}, n == m^{nf}\}$$

Theorem 4.5 Given a finite approximation $[\Lambda^0]_s$, we can always find a database D such that

$$[\Lambda^0]_s(\boldsymbol{F}/\Lambda^0) \subseteq Z(D)$$

Theorem 4.6 Given a sequence of approximations $[\Lambda^0]_0$, $[\Lambda^0]_1$, …, we can find a sequence of databases D_0, D_1, … such that

$$[\Lambda^0]\ (\boldsymbol{F}/\Lambda^0) \subseteq \cup_{s \in \mathbf{N}}\ Z(D_s)$$

5 An Enumeration of EP Databases

In this section, we give an enumeration for all EP databases. The aim is to show that the EP data model does nothing else but approximates the complete properties $[\Lambda^0]$ of all partial computable functions.

There are three dimensions we have to consider: infinitely many constants in $\boldsymbol{C}$, infinitely many identifiers in $\boldsymbol{F}$, and infinitely many left-most sub-terms a term may

contain. The idea is to set an up bound $s \in \mathbf{N}$ for the number of constants, the number of identifiers, and the number of the left-most sub-terms of a term, e.g., the size of a term, a database is allowed to contain. We can find all the databases with the bound s, denoted as $\boldsymbol{D}_s$. Using Cantor's diagonal method, we can find all databases with the bound $s + 1$, where the bounds of constants, identifier, and the size of a term in a database is increased by 1.

Instead of the notation $\boldsymbol{D}_s$, we use a triplet $[C, F, k]$ to denote all the databases generated with up bounds C – a finite set of constants, F – a finite set of functional identifier, and k – the size of a term. Correspondingly $[C + 1, F, k]$ denotes databases with an extra constant, $[C, F + 1, k]$ denotes database with an extra identifier, and $[C, F, k + 1]$ denotes databases with a term whose size is up to $k + 1$. Therefore $[C + 1, F + 1, k + 1] = \boldsymbol{D}_{s+1}$ when $[C, F, k] = \boldsymbol{D}_s$.

Before precisely giving the algorithm of enumerating databases, we demonstrate how we obtain [{*c*}, {}, 2], [{}, {f}, 2], [{}, {f}, 3], and [{*c*}, {f}, 2].

To save space, we use a pair (m, n) to denote an assignment $m := n$. Further, we ignore an assignment where the assignee is an identifier because the identifier as the assignee doesn't add a new function into databases but serves as an alias (an alternative name) of the assigner.

Example 5.1 An enumeration of [{*c*}, {}, 2]
1. The allowed symbols are *c*.
2. All the terms with 2 as the maximum size are c, <u>c c</u>. According to 2.2, c c cannot be an assignee. We underline it, indicating that it cannot be an assignment.
3. There is not a valid assignment (a constant cannot be an assignee).
4. There is no database generated.

Example 5.2 An enumeration of [{}, {f}, 2]
1. The allowed symbols are f.
2. All the terms with 2 as the maximum size are f, f f.
3. All the possible assignments are (f f, f), <u>(f f, f f)</u>. (the assignment f f := f f is not allowed by Definition 2.2. We underline it, indicating that it is not a valid assignment.)
4. There is only one valid database: {(f f, f)}.

Example 5.3 An enumeration of [{}, {f}, 3]
1. The allowed symbols are f.
2. All the terms with 3 as the maximum size are f, f f, f f f.
3. All the possible assignments are (f f, f), <u>(f f, f f)</u>, <u>(f f, f f f)</u>, (f f f, f), (f f f, f f), <u>(f f f, f f f)</u> (the underlined assignments are invalid by 2.2.)
4. There are 3 valid databases: {(f f, f)}, {(f f f, f)}, {(f f f, f f)}, <u>{(f f, f), (f f f, f)}</u>, <u>{(f f, f), (f f f, f f)}</u>, <u>{(f f f, f), (f f f, f f)}</u>. (The underlined sets are invalid databases by 2.2.)

Example 5.4 An enumeration of [{c}, {f}, 2]
1. The allowed symbols are c, f.
2. The allowed terms with 2 as the maximum size are c, f, f f, f c , <u>c f, c c</u>.

3. All the possible assignments: (f f, f), (f f, c), (f f, f f), (f f, f c), (f c, f), (f c, c), (f c, f f), (f c, f c). (The underlined assignments are invalid by 2.2.)
4. All the database candidates: {(f f, f)}, {(f f, c)}, {(f f, f c)}, {(f c, f)}, {(f c, c)}, {(f c, f f)}, {(f f, f), (f f, c)}, {(f f, f), (f f, f c)}, {(f f, f), (f c, f)}, {(f f, f), (f c, c)}, {(f f, f), (f c, f f)}, {(f f, c), (f f, f c)}, {(f f, c), (f c, f)}, {(f f, c), (f c, c)}, {(f f, c), (f c, f f)}, {(f f, f c), (f c, f)}, {(f f, f c), (f c, c)}, {(f f, f c), (f c, f f)}. The databases with underlines are invalid by 2.2. There are total 15 valid databases, as those without underlines.

Definition 5.5 An algorithm to enumerate all the databases $[C, F, k]$ with the up bounds C – a finite set of constants, F – a finite set of identifiers, and k – the size of a term is defined as the follows:

1. Let $L = C \cup F$, and $|L|$ be the number of members in L.
2. Let T_i be the set of all the terms, each of which take i as its size, where $i \le k$. It is the permutations of i members from L with repetition. Let $|T_i|$ be the number of the permutations. $|T_i| = k^i$.
3. Let T be the set of all the terms, each of which take k as the maximum size. Then $T = \cup_{i \le k} T_i$, $|T| = \sum_{i \le k} |T_i|$.
4. Let A be all the assignments with assignees and assigners from T. It is the permutations of 2 members from T. The size is $|A| = |T|^2$.
5. Let H_i be all the database candidates, each of which takes i members from A, where $1 \le i \le |A|$. H_i is the combinations of i members from A. $|H_i| = |A|! / (i! \times (|A| - i))!$.
6. Let H be all the database candidates, i.e., $H = \cup_{i \le |T|} H_i$, where $|H| = \sum_{i \le |T|} |H_i|$.
7. For each database candidate in H, we create a database space and feed its assignments into the database space. If all the assignments pass the constraints defined in 2.2, the candidate database is valid and recorded. Otherwise, it is invalid and discarded.

The maximum size, the number of assignments, of a database candidate is $|T|$. The actual size of a database is always smaller than $|T|$. The total number of database candidates is $|H|$. The actual number of databases in $[C, F, k]$ is always smaller than $|H|$.

Theorem 5.6 1. $[C, F, k] \subset [C + 1, F, k]$
2. $[C, F, k] \subset [C, F + 1, k]$
3. $[C, F, k] \subset [C, F, k + 1]$
4. $\boldsymbol{D}_s \subset \boldsymbol{D}_{s+1}$

Proof All of them are clear from the algorithm in Definition 5.5 and the examples 5.1, 5.2, 5.3, and 5.4. □

Let $\boldsymbol{D} = \cup_{s \in \mathbf{N}} \boldsymbol{D}_s$

Theorem 5.7 Given a database $D \in \boldsymbol{D}$, there is a lambda term, denoted as $\lambda(D)$, such that $Z(D) \subseteq [\lambda(D)]$ $(\boldsymbol{F}/\Lambda^0)$.

In the following we let $\lambda(D)$ to denote any lambda term such that $Z(D) \subseteq [\lambda(D)]$ $(\boldsymbol{F}/\Lambda^0)$. And further we denote $Z(\boldsymbol{D})$ as $\{[\lambda(D)]\ (\boldsymbol{F}/\Lambda^0) \mid D \in \boldsymbol{D}\}$.

Theorem 5.8 $Z(\boldsymbol{D}) \subseteq [\Lambda^0]\ (\boldsymbol{F}/\Lambda^0)$

Theorem 5.9 $Z(\boldsymbol{D}) = [\Lambda^0]\ (\boldsymbol{F}/\Lambda^0)$

6 Conclusion

The EP data model is a distinguished solution to approximate the denotational semantics of the lambda calculus, e.g.., a domain isomorphic to the space of all continuous functions from itself to itself, e.g., $E_\infty = (E_\infty \to E_\infty)$. Different from others such as Scott's higher-order function space: $E_{n+1} = (E_n \to E_n)$ for all $n \in \mathbf{N}$, where a sub domain E_n doesn't allow to have self application functions [2], the bounded functions definable in the EP data model do contain self application functions. It tells us that the EP data model and a purely typed language system compensate each other, where the former defines finitely many total functions allowing self references and the latter infinitely many functions prohibiting self references [1].

Approximations to partial computable functions are certainly not the partial computable functions themselves. However the union of the approximations and the union of all partial computable functions amount to the same object: the universe of all computable information. An approximation, e.g., a bounded function, that has finite functions each of which has finite ordered pairs (properties) is particularly important to database management where only finite objects are desired to be manageable. The relational data model, where tables can be interpreted as level-2 functionals, certainly cannot define approximations to partial computable functions. A hierarchical data model, e.g., XML, can be interpreted as finite approximations to partial computable functions (a detailed discussion is not provided here). However, only the EP data model defines self application functions (such that more expressible operations can be applied to databases) and provides users a way to eliminate data redundancy.

Lacking a higher expressive data types in programming languages for finite data is a reason to utilizing multiple data types such as lists, trees, graph-based structures, and relations imported from a relational database. As a result, the communications among the multiple data types complicate software development efforts. The EP data model eliminates the extra burden in expressing finite data in programming languages and in data communications.

There are many kinds of approximations to partial computable functions. Primitive recursive functions can be viewed to be approximations to partial recursive functions. Purely typed systems such as a purely typed lambda calculus define approximations to the denotational semantics of the lambda calculus. They are not interesting to us in this article because they are infinite and not suitable to database management. Finite approximations to partial recursive functions [3] are interesting because they are finite and suitable to database management. The bounded functions discussed in the article, larger than the finite approximations, are interesting us because they are actually finite sets of finite functions and the corresponding EP data model is recursive for database

management. Is there another kind of approximations that are larger than the bounded functions and have a corresponding recursive language useful in database management?

Acknowledgement: Author thanks Dag Normann for his comments and encouragement.

Reference:

[1] J. R. Longley. "*Notions of Computabiloity at Higher Types I*". In Logic Colloquium 2000, Vol. 19 of Lecture Notes in Logic, pp 32 – 142.

[2] D. Scott. "*Outline of a Mathematical Theory of Computation*". Proc. 4th Ann. Princeton Conf. On Information Sciences and Systems, Princeton Univeristy, Princeton, N.J., 1970, pp. 169-176.

[3] R. Soare. "*Computability and Recursion*". Bulletin of Symbolic Logic 2 (1996), pp. 284 -321.

[4] C. P. Wadsworth. "*The Relation Between Computational and Denotational Properties For Scott's D_∞-Models of the Lambda-Calculus*". SIAM J. Comput. Vol. 5, No. 3, September 1976, pp 448 – 521.

[5] K. H. Xu. "*A Bi-directional Mapping between Froglingo Programming Language and the Lambda Calculus*". Technical Report, September 2011, available at http://www.froglingo.com.

[6] K. H. Xu, J. Zhang, S. Gao. "*Approximating Knowledge of Cooking in Higher-order Functions, a Case Study of Froglingo*". Workshop Proceedings of the Eighteenth International Conference on Case-Based Reasoning (ICCBR 2010), page 219 – 228.

[7] 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.

[8] K. H. Xu, B. Bhargava. "*An Introduction to the Enterprise-Participant Data Model*". The Seventh International Workshop on Database and Expert Systems Applications, September, 1996, Zurich, Switzerland, page 410 - 417.

7 Appendix

Lemma 2.7 An assignee has a unique normal form in a database.

Proof When a term a_0 is an assignee in a database D, we may find a chain of assignments: $a_0 := a_1, \ldots, a_{j-1} := a_j \in D$ for $j \geq 1$. By 2.4.1, if a_j has a normal form, then a_0 takes a_0's normal form as its unique normal form. First, j must be a finite number because there are only finite assignments in D and there is not a set of circular assignments in D by 2.2.4. Secondly, a_j must be a normal form, i.e., a constant or a proper sub-set of an assignee in the database D because it can no longer be an assignee according to 2.2.3. Because j is finite, we obtain a_j as the normal form of a_0 in finite steps. □

Theorem 2.8 A term $a \in \boldsymbol{E}$ has one and only one normal form under a database D.

Proof Here is the algorithm that calculates the normal form of a:

1. If a is a constant or an identifier in D, i.e., $a \in \boldsymbol{C} \cup (\boldsymbol{F} \cap D)$, a itself is its only normal form by Definition 2.3. In one step, we prove $a \rightarrow_{\text{EP}} a$.
2. If a is an identifier not in database D, a is reduced to *null* (the only normal form) by Definition 2.4.2, i.e., $a \rightarrow_{\text{EP}} \mathit{null}$. The reduction takes finite steps because D is finite.
3. If a is an identifier as an assignee in database D, it has a unique normal form by 2.7.
4. If a is an application, e.g., $a \equiv p\ q$, we assume that p and q have their unique normal forms p' and q' respectively, obtained in finite steps.
   a) If $p'\ q'$ is not in database, i.e., $p'\ q' \notin D$, then $p'\ q' \rightarrow_{\text{EP}} \mathit{null}$ by Definition 2.4.3. We further have $a \equiv p\ q \rightarrow_{\text{EP}} p'\ q' \rightarrow_{\text{EP}} \mathit{null}$. Therefore a is uniquely reduced to *null*. The reduction takes finite steps because D is finite.
   b) If $p'\ q'$ is a proper sub-term of an assignee. By the definition of normal forms (Definition 2.3), $p'\ q'$ is the unique normal form by itself. The reduction takes finite steps because D is finite.
   c) If $p'\ q'$ is an assignee, it has a unique normal form by 2.7. □

Lemma 2.9. Given a database D, the set of all non-constant normal forms, i.e., $NF^D - \boldsymbol{C}$, is finite.

Proof There are finitely many assignments and each assignment has finitely many sub terms. Therefore there are finitely many terms m such that $m \in \boldsymbol{E}$ and $m \in D$. By the Definition 2.3, the total number of non-constant normal forms in D is finite. □

Theorem 2.10 Given a database D, there exists a computable function Y^D: $\boldsymbol{E} \rightarrow \boldsymbol{E}$ such that

$$Y^D(m) = m^{\text{nf}}$$

for all $m, n \in \boldsymbol{E}$.

Proof By Theorem 2.8, m has one and only one normal form. Therefore Y^D is a function. Y^D is computable because the reduction process from a term to its normal form is effective.□

Theorem 2.11 Given a database D, there exists a finite function X^D: $D \rightarrow NF^D$ such that

$$X^D(m) = m^{\text{nf}}$$

for all $m \in \boldsymbol{E}$ and $m \in D$.

Proof: There are finitely many assignments and each assignment has finitely many sub terms. Therefore there are finitely many terms m such that $m \in \boldsymbol{E}$ and $m \in D$. Since Y^D is a function, so is $X^D \subset Y^D$ is a finite function. □

Theorem 2.12 If there exists a sequence of assignments $a_0 := a_1$, $a_1 := a_2$, …, $a_{j-1} := a_n \in D$ for a $n \geq 1$ such that $a_0^{\text{nf}} \neq null$ and $a_n \in \text{SUB}^+(a_0)$, then Y^{D} is not finite but bounded.

Proof: Given any term $m \in \boldsymbol{E}$, we have $m^{\text{nf}} \in NF^D$, a finite set. Therefore, Y^{D} is a bounded function.

When $a_n = \text{SUB}^+(a_0)$, we can rewrite a_0 as a context filled with a_n, i.e., $a_0 \equiv \text{C}[a_n]$. Due to the sequence of the assignments, we have $a_0 == a_n$. It means that a_0 is equal to a infinitely many terms $\text{C}[a_n]$, $\text{C}[\text{C}[a_n]]$, …, $\text{C}[\text{C}\ldots[\text{C}(a_n)]\ldots]$. It concludes that there are infinitely many elements $m \in \boldsymbol{E}$ such that m^{nf} != *null* because a_0 has a non *null* normal form. On the other hand, there are infinitely many terms equal to *null*, e.g, infinitely many identifiers having *null* as the normal form. Therefore Y^{D} is not finite.□

Y^{D} is the constant application operator we discussed in Section 1. There are many more operators including <=+ mentioned in Example 1.2 that are not discussed here. For more information, please reference [6] and [7].

Theorem 4.3 $[\Lambda^0]_{\text{s}}^{-}(\boldsymbol{F}/\Lambda^0)$ is an EP database.

Proof We show that $[\Lambda^0]_{\text{s}}^{-}(\boldsymbol{F}/\Lambda^0)$ satisfies the constraints posted in Definition 2.2.

1) $[\Lambda^0]_{\text{s}}^{-}(\boldsymbol{F}/\Lambda^0)$ is a finite set of assignments because $[\Lambda^0]_{\text{s}}$ is a finite set.
2) Each assignee has only one assigner (in a satisfaction of 2.2.1), i.e.,
$$a := b_1 \text{ and } a := b_2 \in [\Lambda^0]_{\text{s}}^{-}(\boldsymbol{F}/\Lambda^0) \Rightarrow b_1 \equiv b_2$$
because there are not two assignees that are identical in the $[\Lambda^0]_{\text{s}}^{-}(\boldsymbol{F}/\Lambda^0)$.
3) A proper sub term of an assignee cannot be an assignee (in a satisfaction of 2.2.2). If there is an assignment $a := b \in [\Lambda^0]_{\text{s}}^{-}(\boldsymbol{F}/\Lambda^0)$ and let $x \in \text{SUB}^+(a)$, then we cannot have a c such that $x := c \in [\Lambda^0]_{\text{s}}^{-}(\boldsymbol{F}/\Lambda^0)$ according to the definition of $[\Lambda^0]_{\text{s}}^{-}(\boldsymbol{F}/\Lambda^0)$ given in Definition 4.1.
4) There is not a set of circular assignments in $[\Lambda^0]_{\text{s}}^{-}(\boldsymbol{F}/\Lambda^0)$, i.e., if there is a sequence of assignments $a_0 := a_1$, $a_1 := a_2$, …, $a_{j-1} := a_n \in [\Lambda^0]_{\text{s}}^{-}(\boldsymbol{F}/\Lambda^0)$ for a $n \geq 1$, then a_n must not be a_0. In each $a := b \in [\Lambda^0]_{\text{s}}^{-}(\boldsymbol{F}/\Lambda^0)$, the corresponding lambda term B is a normal form. Therefore, we cannot have a c such that $b := c \in [\Lambda^0]_{\text{s}}^{-}(\boldsymbol{F}/\Lambda^0)$ according to Definitions 3.2 and 3.3. Therefore, there is not a set of circular assignments in $[\Lambda^0]_{\text{s}}^{-}(\boldsymbol{F}/\Lambda^0)$, a satisfaction of 2.2.4.

5) In each $a := b \in [\Lambda^0]_s^- (\boldsymbol{F}/\Lambda^0)$, the corresponding lambda term B is a normal form, and therefore the corresponding b is an identifier in $[\Lambda^0]_s^- (\boldsymbol{F}/\Lambda^0)$. It satisfies the constraint of 2.2.3 that an identifier is allowed to be an assigner in a database.

Therefore $[\Lambda^0]_s^- (\boldsymbol{F}/\Lambda^0)$ is a database.□

Theorem 4.5 Given a finite approximation $[\Lambda^0]_s$, we can always find a database D such that

$$[\Lambda^0]_s(\boldsymbol{F}/\Lambda^0) \subseteq Z(D)$$

Proof We let D to be initially $[\Lambda^0]_s^- (\boldsymbol{F}/\Lambda^0)$. For each $(M_0\ M_1\ \ldots\ M_n,\ Q) \in [\Lambda^0]$ where there is another pair $(N_0\ N_1\ \ldots\ N_k,\ P) \in [\Lambda^0]$ for a $k \geq 0$ such that $N_0\ N_1\ \ldots\ N_k \in$ SUB+$(M_0\ M_1\ \ldots\ M_n)$, i.e., $M_0\ M_1\ \ldots\ M_n \equiv C[N_0\ N_1\ \ldots\ N_k]$ for a context C[ ]. In this case, $(M_0\ M_1\ \ldots\ M_n,\ Q) \notin [\Lambda^0]_s^-$ by Definition 4.1. Then we add the pair $(C[P], Q)$ $(\boldsymbol{F}/\Lambda^0)$ into D. The resulting $Z(D)$ will include $(m_0\ m_1\ \ldots\ m_n,\ q)$. Note that the expression $(C[P], Q)$ $(\boldsymbol{F}/\Lambda^0)$ is a pair of lambda expressions substituted by their corresponding identifiers in $\boldsymbol{F}$, a notation borrowed from Definition 4.2. □

Theorem 4.6 Given a sequence of approximations $[\Lambda^0]_0$, $[\Lambda^0]_1$, …, we can find a sequence of databases D_0, D_1, … such that

$$[\Lambda^0]\,(\boldsymbol{F}/\Lambda^0) \subseteq \cup_{s\in\mathbf{N}}\, Z(D_s)$$

Proof By Theorem 4.5, we have $\cup_{s\in\mathbf{N}}\,([\Lambda^0]_s(\boldsymbol{F}/\Lambda^0)) \subseteq \cup_{s\in\mathbf{N}}\, Z(D)$. By Theorem 3.5, we have $[\Lambda^0] = \cup_{s\in\mathbf{N}}\,[\Lambda^0]_s$, and therefore $[\Lambda^0]\ (\boldsymbol{F}/\Lambda^0) = (\cup_{s\in\mathbf{N}}\,[\Lambda^0]_s)\ (\boldsymbol{F}/\Lambda^0) = \cup_{s\in\mathbf{N}}$ $([\Lambda^0]_s\,(\boldsymbol{F}/\Lambda^0))$. Then we have $[\Lambda^0]\,(\boldsymbol{F}/\Lambda^0) \subseteq \cup_{s\in\mathbf{N}}\, Z(D_s)$.□

Theorem 5.7 Given a database $D \in \boldsymbol{D}$, there is a lambda term, denoted as $\lambda(D)$, such that $Z(D) \subseteq [\lambda(D)]\ (\boldsymbol{F}/\Lambda^0)$.

Proof According to Church's thesis, D, representing a bounded function Y^D, can be expressed in a lambda term $\lambda(D)$ such that $Z(D) = [\lambda(D)]\ (\boldsymbol{F}/\Lambda^0)$. Actually the lambda term is expressed in multiple fixed points of the lambda calculus [5].

Further we can find more lambda terms, e.g., $\lambda_0(D)$, $\lambda_1(D)$, …, such that $Z(D) \subseteq [\lambda_i(D))]\ (\boldsymbol{F}/\Lambda^0)$ for each $i \geq 0$. □

Theorem 5.8 $Z(\boldsymbol{D}) \subseteq [\Lambda^0]\,(\boldsymbol{F}/\Lambda^0)$

Proof For each $D \in \boldsymbol{D}$, we have $\lambda(D) \in \Lambda^0$. Then we have $\{\lambda(D) \mid D \in \boldsymbol{D}\} \subseteq \Lambda^0$. Therefore $Z(\boldsymbol{D}) \subseteq [\Lambda^0]\,(\boldsymbol{F}/\Lambda^0)$ by Theorem .57. □

Theorem 5.9 $Z(\boldsymbol{D}) = [\Lambda^0]\,(\boldsymbol{F}/\Lambda^0)$

Proof It is clear from Theorem 4.6 and Theorem 5.8.□