A Bi-directional Mapping between Froglingo Programming Language and the Lambda Calculus

Kevin Xu
2306 Johnson Circle
Bridgewater, New Jersey 08807, U.S.A.
kevin@froglingo.com

Abstract

The EP (Enterprise-Participant) data model is a language semantically equivalent to a class of total recursive functions. Because of the equivalence, Froglingo, a programming language incorporated with the EP data model, is a monolith that consolidates databases with programming languages. In this paper, we discuss Froglingo by providing bidirectional mappings between Froglingo and the lambda expressions. With the mappings, as a mathematical foundation of Froglingo, we can better relate Froglingo with the lambda calculus and other programming languages, and therefore objectively assess what is meant by the monolith of Froglingo.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definition and Theory – Syntax and Semantics; D.3.3 [Programming Languages]: Language Constructs and Features – abstract data types, control structures; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic – Lambda calculus and related systems.

General Terms Languages, Theory,

Keywords programming languages, lambda calculus, redex, normal forms.

1. Introduction

Many database applications were written in programming languages in the 1960s and 1970s, and are still in operation. The use of database management system (DBMS) came to database application software around the 1970s. It significantly improves the productivity in software development and maintenance.

The traditional data models and data structures, i.e., the relational data model, the hierarchical data model, and graph-oriented data structures, cannot express all desired business data. Hierarchical data, for example, can be folded into a relation, but its containment relationships cannot be captured by the relational data model with the expressive power of the relational algebra [3]. Another example would be relationships among the vertices in a directed graph, (e.g., is there a path from A to B?), which cannot be captured in both the relational data model and the hierarchical data model. As a result, database applications continuously require intense, though relieved, development and maintenance work, which could be avoided if a more expressive data model were realized and leveraged.

The EP (Enterprise-Participant) data model is semantically equivalent to a class of total recursive functions [35]. The equivalence says that programmers are not allowed to construct an application program that may not terminate on an input. At the same time, it says mathematically that any meaningful application programs, i.e., those with the semantics falling into the class of total recursive functions, could be expressed in the EP data model with the hypothesis of infinite space and time.

Because of the equivalence, the EP data model alone is able to consistently manage as much finite data as we need. Here the consistency meant that finite data from various application domains, including database management, data communication, access control, and file management, is collectable and exchangeable in the EP data model with a single data structure [39] and [37].

Froglingo, a programming language incorporated with the EP data model, extends the EP data model to manage infinite data. The extension is implemented by incorporating variables, i.e., comparably variables in the lambda calculus, into the data structure of the EP data model. As a result, business data and business logic are stored in the same data structure, and operations on the data structure are expressed by the same set of operators, that constitutes Froglingo to be a programming language, a monolith consolidating databases with programming languages [36].

Froglingo was originally proposed for database and data communication issues [40]. It is closely related to the lambda calculus. Therefore it has been continuously guided by the lambda calculus in its development. Here, we give a simple example to quickly show the connections between the two systems. The function $sq\ (x) = x * x$, i.e., calculating the square of a number, has a lambda expression:

$$\lambda x.\ (*^{\lambda}\ x\ \ x)$$

Here we assume that $*^{\lambda}$ is the lambda expression for the operator *. In Froglingo, we have a corresponding expression:

create sq $x := (*$x* * *$x*)

Here *sq* is an identifier, the name for the square function, *$x* is the variable. We added '*$*' in the front of variables to distinguish them from identifiers. This expression would not make Froglingo much different from Scheme:

(*define sq* (*lambda* (*x*) (((* *x*) *x*))

until we see another expression that instantly alters the function *sq* without modifying its original definition:

create sq 1 := 20

The above two statements result in two expressions in a database: {*sq 1 := 20*; *sq $x* := (*$x* * *$x*)}. When a query *sq 3* is provided, Froglingo will match the second expression in the database and return 9. But when a query *sq 1* is provided, Froglingo will match the first expression and returns 20.

From the examples above and from our discussion of this paper, we will see that Froglingo is similar to the lambda calculus because applications and abstractions, via identifiers, are the primary concepts in programming. At the same time, Froglingo construct abstractions without using variables, that is not a case in the lambda calculus.

Contributions of this paper. We give a formal theory for Froglingo. Our discussion reveals that Froglingo, like ML and Scheme, appears to be solely a syntactical variation of the lambda calculus, e.g., $\lambda x.M$ is rewritten as *p $x* := *M* (Section 6). Further more, Froglingo has exactly one-to-one mappings on normal forms, head normal forms, and weak head normal forms with the lambda calculus and therefore with other functional programming languages (Sections 6 and 4). The similarity is attributed to Froglingo's function as a programming language. Constructing (higher-order) functions without variables is the unique feature that differentiates Froglingo from the lambda calculus and other programming languages. Syntactically, it is done through applicative structure, e.g., *p q* := *3* (Section 2), again in a way similar to the lambda calculus. Semantically, however, it fundamentally introduces a new avenue to constructing (higher-order) functions by enumerating properties, in parallel to recursion. Enumeration makes database management easier, and it is a complement of recursion (Sections 4 and 5) while recursion is a complement of enumeration in describing infinite data with finite presentation.

The formal theory serves as a foundation to support the concept of Froglingo's monolith and some results from other work. In paper [34], a set of built-in operators, reflecting ordering relations among higher-order functions, were introduced. Because of the EP data model with the built-in operators, we can easily express many known structures and queries with high complexities, such as directed graphs with cycles and self-applications. (We will have some examples to reiterate this feature in this paper.) Again because of the EP data model, an application system [32] implemented in Froglingo (an award recipient in the ICCBR 2010 Computer Cooking Contest) demonstrated that programmers can take advantage of the built-in data structure for various data that would otherwise have to be constructed in many data types of traditional programming languages, that users have the flexibility of specifying queries in a unstructured textual string that would otherwise have to be chunked by users into pieces with an extra step of learning system-specific structure, and that similarities can be represented structurally that would otherwise have to be represented in weight in traditional technologies. Finally, because of the EP data model, the authors in the paper [31] contended that Froglingo is untyped and at the same time is safer than traditional programming languages.

In Section 2, we discuss the EP data model as a quick step to introduce the most important sub language of Frogling. Starting from 4, we discuss Froglingo as a whole, including the EP data model, about its normalization, reduction, semantics and computability.

2. EP Data model

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 the relational data model and the child-parent relationships in the hierarchical data model are typical examples. These relationships are too simple, and don't reflect the extent to which the complexities of the real world can be managed using a computer.

The logic of the EP data model is that if one entity is dependent on other entities, then those entities are precisely two in number. Drawing terminology from the structure of an organization or a party as in article [40], one dependent entity is called enterprise (such as organization and party), the other is called participant (such as employee and party participant), and the dependent entity is 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.

2.1 Syntax

DEFINITION 1 (EP terms) Terms T, ranged over by *t*, are given by the grammar

$$t ::= c \mid a \mid (t\ t)$$

where *c* ranges over a set of constants C where *null* is a special constant and *a* ranges over a set of identifiers P.

Similar to lambda expressions, EP terms have application as the most noticeable structure. Therefore, we adopt many notations from the lambda calculus, including sub-term, left (right) sub-term, SUB(*t*) for all sub-terms in *t*, FV(*t*) for all the variables in *t*, and *a b* instead of (*a b*) when there is no ambiguity.

Different from the lambda calculus, the reduction process of the EP data model is based on a database. We start to define the concept of assignments first.

DEFINITION 2 (assignments) Assignments A, ranged over by *s*, are given by the grammar

$$s ::= t \mid t := t$$

where *t* ranges over T.

Given an assignment, the assignee plays the dominant role in naming the assignment. A term without an assigner can be an assignee as well.

NOTATION 3 (assignees) Given an assignment $m := n \in A$, a left sub term of *m* is called an assignee. Further,
1. the term *m* is called the explicit assignee.
2. An assignee that is not explicit assignee is called a derivable assignee.
3. When we say that a term *m* is in a set of assignments *A*, i.e., $m \in A$, we say that *m* is an assignee.

Given an assignment *a b (c d) := e*, for example, the explicit assignee is *a b (c d)*, and it acts as the name of the assignment. *a, a b* are derivable assignees for *a b (c d)*.

DEFINITION 4 (database) An EP database D is a set of assignments $A \subset \mathcal{A}$ such that the following things are true:
1. An assignee cannot be a constant, i.e.,

$$m \in D \Rightarrow m \in (T - \mathcal{C})$$

2. If $m\ n$ is an assignee in D, n must not have an assigner, i.e.,

$$m\ n \in D \cap \forall k \in \mathcal{T},\ (n := k) \notin D$$

3. If an assignment $(m := n)$ is in D, m cannot be the left sub-term of another assignee in D, i.e.,

$$(m := n) \in D \Rightarrow \forall t \in \mathcal{T},\ m\ t \notin D$$

4. 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 $n \geq 1 \Rightarrow m_n := m_0 \notin D$.
5. An assignee can serve only one assignment in D, i.e,

$$(m := n_1), (m := n_2) \in D \Rightarrow n_1 \equiv n_2$$

EXAMPLE 5 1 (A school administration database)
SSD.gov John SSN := 123456789;
SSD.gov John birth := '6/1/1990';
SSD.gov John photo.jpg := ...; / a binary stream*/*
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";
2 (a directed graphs with circles) A directed graph with connections v_1 to v_2, v_2 to v_1, and v_2 to v_3 has the following assignments in a database:
$v_1\ v_2 := v_2;$
$v_2\ v_1 := v_1;$
$v_2\ v_3 := v_3;$

When we say that a term is in a database, in the Definition 4, we always say that the term is an assignee. The fact that a sub term of an assigner can also physically be in a database is not important and therefore is not discussed in this paper.

According to Definition 4, if an assignee is in a database, one of its left sub terms is also in the database, e.g., *SSD.gov* and *SSD.gov John* must be in when *SSD.gov John SSN* is in a database. We don't show them here because it is clear and implied by the following propositions:

PROPOSITION 6 1. If an application is in a database, so are its left sub-terms, i.e.

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

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

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

PROOF. 6.1. It is clear from Definition 1 that the existence of an application must imply the existence of the sub terms. A database always stores the sub-terms when it stores an application.

6 2. If is clear from Definition 2 that the existence of an assignment must imply the existence of the assignee. A database always stores the assignee and the assigner if it stores an assignment.

2.2 EP Normal Forms and Reductions

Given a database, each term in the $\mathcal{T}$ can be reduced to a normal form.

DEFINITION 7 Given a database D, the set of normal forms, denoted as EPNF, is defined as follows:
1. All the constants, i.e.,

$$c \in \mathcal{C} \Rightarrow c \in \mathit{EPNF}$$

2. All the derivable assignees in D, i.e.,

$$m \in D, \forall n \in \mathcal{T}, (m := n) \notin D \Rightarrow m \in \mathit{EPNF}$$

For example, terms *"F"*, *SSD.gov*, and *SSD.gov John* are normal forms, but not *SSD.gov John birth* in Example 5.1.

We will see that EPNF is a strict subset of WHNF to be discussed in Section 4.

DEFINITION 8 (EP one-step reduction rules) Given a database D, we have the one-step evaluation rules, denoted as $\rightarrow$:
1. A constant is reduced to itself, i.e., $c \in \mathcal{C} \Rightarrow c \rightarrow c$
2. An identifier not in D is reduced to *null*, i.e.,

$$p \in \mathcal{P} \cap p \notin \mathrm{D} \Rightarrow p \rightarrow null$$

3. An assignee in D is reduced to its assigner, i.e.,

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

4. If $m, n \in \mathit{EPNF}$, and $m\ n \notin D$, then $m\ n$ is reduced to *null*, i.e.,

$$m, n \in \mathit{EPNF}, m\ n \notin D \Rightarrow m\ n \rightarrow \texttt{null}$$

5. The application of two terms is reduced to the application of their normal forms, i.e.,

$$m, n \in \mathcal{T},\ m \rightarrow m',\ n \rightarrow n' \Rightarrow m\ n \rightarrow m'\ n'.$$

DEFINITION 9 Let $m, n \in \mathcal{T}$ with a given database D. If there is a finite sequence $l_0, \ldots, l_q \in \mathcal{T}$, where $q \geq 0$, such that $m \equiv l_0, l_0 \rightarrow l_1, \ldots, l_{q-1} \rightarrow l_q, l_q \equiv n$, then
1. m is effectively, i.e., in finite steps, reduced to n, written as $m \rightarrow_{\mathrm{EP}} n$.
2. If $m_1 \rightarrow_{\mathrm{EP}} n$ and $m_2 \rightarrow_{\mathrm{EP}} n$, then we say that m_1 is equal to m_2, denoted as $m_1 == m_2$. The relation == is the complete set of the equations derivable from the environment of D.

EXAMPLE 10 Below are a few equations from the databases in Examples 4 and Examples 6:
SSD.gov John SSN == 123456789;
(*college.edu admin* (*SSD.gov John*) *Major*) == *college.edu CS;*
$v_1\ v_2\ v_1 == v_1;$
$v_1\ v_2\ v_1\ v_2\ v_1 \ldots\ v_1 == v_1;$

The reduction of the EP data model is strongly normalizing because every term can be effectively reduced to a normal form.

2.3 Computability

The EP data model, as a formal theory, has been studied in [35]. It has been proved to be semantically equivalent to a class of total recursive functions. It means that the complete set $\mathcal{T}$, under a given database, is always interpreted as a total and higher-order function. Given a derivable assignee m, we can always find a set of distinguished terms (normal forms) n_i, here i is 0 or a positive nature number, such that $m\ n_i$ is in D for each i. The relationships between m, n_i, and $m\ n_i$ are exactly the relationship between function, argument, and the corresponding value of applying function to argument.

On the other hand, the equivalence says that any total recursive function, i.e., applying it to an arbitrary argument always terminates no matter it is finite or infinite, would be eventually enumerated and stored into a EP database with the complete set of the argument and value pairs that represent the complete properties of the function. The equivalence is based on the hypothesis that we had infinite time and space, in the same way that the computability of a Turing-machine was given. The base of this claim is: a class of total recursive functions can be enumerated [4]. This also can be intuitively concluded from the known conclusion in a computability text book: a class of partial recursive functions can be enumerated by using a Turing machine. It doesn't say that a machine can collect all the property of a total recursive function at a given time in the future, but that a machine will eventually collect every total recursive function with the hypothesis that the machine had infinite time and infinite space.

To understand the EP data model's computability, we can think of the following intuitive process: A Turing machine is enumerating the entire set of partial recursive functions, and it feeds output to a system in the EP data model. When the Turing machine computed the result of applying a function to an argument with termination, it passes the result to the EP system and the EP system records the argument and the result pair for the total function. What the EP system recorded are exactly those total recursive functions. With the hypothesis, the Turing machine would have enumerated the entire set of partial recursive functions and the EP system would have collected the entire set of total recursive functions at the end of an infinite time, that would never come.

A relational database or a programming language is able to collect the properties of a total recursive function with the given hypothesis. But only the EP data model, that is semantically equivalent to the class of total recursive functions, is able to replay the applicative behaviors of total recursive functions after the collection.

With the same intention, many strongly typed programming languages, e.g., the purely typed lambda calculus [5], Total functional programming [30], Gödel's system T, and Nominal system T [26], also limit the computations to total recursive functions. However the computability is limited to a strictly subset of a class of total recursive functions because of types. In other words, given a strongly typed programming language that always terminate on arbitrary inputs, we always can find more total functions that can not be described in the language. The self-application function $F = \{<F, 1>, <2, 0>\}$, for example, cannot be expressed in a strongly typed system, but simply expressed in the EP data model as:

F F := 1;
F 2 := 0;

Similarly, the EP data model expressions in Example 5.2 for a directed graphs with cycle cannot be expressed in a strongly typed system.

For additional discussion about the computability, consistency, and soundness of the EP data model, please reference [35] and more discussions in Section 4 and 5 of this paper.

3. Froglingo

With the EP data model that is equivalent to a class of total recursive functions, we can user the EP data model to model finite data as much as possible. But a programming language, i.e., a Turing-complete system, is still needed. First, constructing arbitrary functions for both queries and business logic on top of a managed data set requires a programming language. Although the built-in operators introduced in Section 2.4 can be used to construct many useful queries, they don't exhaust all the queries that are required for practicality and that are within a class of total recursive functions. Viewing an EP database as a finite set of higher-order functions, in addition to the built-in operators in Section 2.4, there are still an infinitely many total recursive functions that are potentially demanded by applications.

Second, some business data may be expressed more conveniently as business logic. By business data, we normally mean finite properties. By business logic, we emphasize its finite presentation for mostly infinite properties. To express the opening hours of a shopping center, e.g., from 9:00 am to 9:00 pm except on weekends, one may prefer not to repeat the same schedule 5 times for 5 workdays in a database, but instead to specify it only once. Representing this type of business data demands programming language or other specialized language systems such as those called constraint databases **[27]**.

In this section, we introduce variables which, along with the EP data model, constitute Froglingo as a programming language. For the features beyond variables, please reference [38].

A variable in Froglingo is represented by an identifier preceded by the symbol "*$*". For example, *$a_variable*, and *$student*. It is a new type of terms.

DEFINITION 14 ($\mathcal{F}$-terms) Terms $\mathcal{F}$, ranged over by t, are given by the grammar

$$t ::= c \mid x \mid a \mid (t\ t)$$

where c ranges over $\mathcal{C}$, x ranges over a set of variables $\mathcal{V}$ and a ranges over $\mathcal{P}$.

Obviously, $\mathcal{T}$ is a strictly subset of $\mathcal{F}$.

DEFINITION 15 (database extension) 1. If a variable is in an assigner in a database D, it must be in the assignee, i.e.,

$$(m := n) \in D, x \in \mathcal{V}, x \in \text{SUB}(n) \Rightarrow x \in \text{SUB}(m)$$

2. A variable cannot be an assignee by itself, i.e.,

$$x \in \mathcal{V}, x \in \text{SUB}(m), (m := n) \in D \Rightarrow x \neq m \in \text{SUB}(m) \wedge \forall b \in \mathcal{T}, (x\ b) \notin\in \text{SUB}(m)$$

With the addition of variables, we can have following valid assignments in a database:

EXAMPLES 16 (database with variables)
fac 0 := 1;
*fac $n := ($n * (fac ($n - 1)));*
fun $x 1 $y := ($x + $y);
*fun $x 2 $y := ($x * $y);*

A detailed discussion on the normalization and reduction rules will be fully discussed in Section 5. Before that point, we will use our intuition to reason reduction process. Here are a few sample query expressions and reduction results.
fac 4 $\rightarrow_{\mathbf{f}}$ *24;*
fun 3 2 4 $\rightarrow_{\mathrm{f}}$ *12;*

Semantically, the expressions above are equivalent to a database having infinite assignments. For example, the factorial function will have the following enumerations: *fac 0 = 1; fac 1 = 1; fac 2 = 2; fac 3 = 6;* …. This demonstrates that variables semantically add nothing new to the EP data model, but syntactically to the finite expressions for possible infinite entities (semantics). At the same time, we have to be aware that variables also add non-termination process back to Froglingo.

A variable can be restricted within a range to prevent unwanted data from being its instances and (or) to prevent an operation from not terminating. For example, the following expressions can be used to represent the tax rule: The tax rate is 20% if a salary is less than $100,000, and 40% otherwise.

EXAMPLE 17 (variables with ranges)

tax \$s1:[*\$s1 >= 0 and \$s1 < 100000*] := (*\$s1 * 0.2*);
tax \$s2 := (*\$s2 * 0.4*);

From the example above, we see that multiple variables are possible under a single assignee for multiple ranges. When multiple ranges are needed, we require that each variable is named differently (otherwise, user would be prompted with an error message).

In this paper, however, we limit ourselves to one variable without range because it is sufficient to reach out our goals in this paper.

4. Normalization and Reductions

With the addition of variables, the reduction process of Definition 8 must be extended. For example, given the expressions in a database *D*:

$$g\ 7\ 2 := 4;$$
$$g\ \$x\ \$y := (\$x + \$y);$$

will have the following reductions:

$$g\ 7\ 2 \rightarrow_F 4$$
$$g\ 2\ 3 \rightarrow_F 5$$

In addition to full reductions, a partial substitution, i.e., applying a function to too few arguments, is also allowed. For example, the expression *g 2* will be converted to *g \$x* [*\$x*:=*2*], which would be equivalent to $\lambda y.(2 + \$y)$. In this section, we formally discuss the extended normalization and reduction process of Froglingo.

We further adopt additional notations from the lambda calculus. We denote $\overline{m}$ as a sequence of terms, i.e., $\overline{m} \equiv m_1\ m_2 ...\ m_k$, here *k* is a nature number equal to or greater than 0. $|\overline{m}|$ is denoted the size of the sequence. Let $m\ \overline{n}$ be a term, here $\overline{n}$ is a sequence of terms, $m\ \overline{n}$ is denoted as $m\ n_1\ n_2 ...\ n_k$, here *k* is a nature number equal to or greater than 0. $|\overline{n}|$ is denoted the size of the sequence following *m*.

To accommodate various forms of assignees in Froglingo, e.g., *fun \$x 1 \$y* in Example 16, we also use the form: $m\ \overline{x}$ to denote a term: $m\ x_1\ x_2 ...\ x_k$, here each element in the sequence $\overline{x}$ must appear in $x_1\ x_2 ...\ x_k$, and $|\overline{x}| <= k$.

DEFINITION 18 (environment) Given a sequence of variables $x_1, ..., x_n$, and a sequence of *F*-terms $v_1, ..., v_n$, here $n >= 0$, we call the form:

$$[x_1 := v_1, ..., x_n := v_n]$$

an environment, denoted as $[\overline{x} := \overline{v}]$, or simply ε sometimes. An environment can be empty.

DEFINITION 19 (assignee under environment) Let $M\ \overline{x}$ be an assignee in a database *D*, $\overline{v}$ a sequence of terms, and $|\overline{x}| = |\overline{v}|$, we denote:

$$M\ [\overline{x} := \overline{v}]$$

as the assignee *M* under the environment $[\overline{x} := \overline{v}]$.

It actually can be viewed as a reduction:

$$M\ \overline{v} \rightarrow_F M\ [\overline{x} := \overline{v}]$$

Therefore, we have:

$$M\ \overline{v} == M\ [\overline{x} := \overline{v}]$$

When we see $M\ [\overline{x} := \overline{v}]$ in the rest of this section, it is equivalently viewed as the *F*-term $M\ \overline{v}$. For example

$$g\ 2\ 3 \rightarrow_F (g\ \$x\ \$y)\ [\$x := 2, \$y := 3]$$
$$g\ 2\ 3 == (g\ \$x\ \$y)\ [\$x := 2, \$y := 3]$$

DEFINITION 20 (substitution) Given a *F*-term *M*, and an assignment $[\overline{x} := \overline{v}]$, we denote

$$(\overline{x} :\rightarrow \overline{v})\ M$$

As the result of substituting $\overline{x}$ with $\overline{v}$ in *M*.

For example, $(\$x :\rightarrow 2)\ (\$x + \$y) \equiv (2 + \$y)$.

Like assignee under environment, substitution is a reduction process as well. For example:

$$(\$x + \$y)\ [\$x := 2] \rightarrow_F (\$x :\rightarrow 2)\ (\$x + \$y)$$
$$(\$x + \$y)\ [\$x := 2] == (\$x :\rightarrow 2)\ (\$x + \$y)$$

Therefore, when we see an expression like $(\overline{x} :\rightarrow \overline{v})\ M$ in this section, it is equivalently viewed as a *F*-term.

NOTATION 21 (open and closed *F*-term) 1. If a term includes variables, we say that the term is open.
2. If a term *t* doesn't include a variable, then we call the term closed.

An assignee under an environment, i.e., $M\ [\overline{x} := \overline{v}]$, is also called closed because its equivalent term $M\ \overline{v}$ doesn't include a variable.

In an assignment of a Froglingo database, every occurrence of a variable is bounded in the sense that the variable must also appear in the assignee if it appears in an assignee. As a matter of fact, assignments with variables in a database correspond to closed λ-terms as we will see in Section 6. The Froglingo term set *F* includes variables, such as *\$x fun* and *fun \$x*, but we are only interested in the terms without variables when we are evaluating Froglingo terms, as if we were only interested in the closed lambda terms when we evaluated lambda terms.

Because of the restriction of bounded variables in a database, we are able to implement Froglingo without dealing free variables during reduction processes for any closed terms as inputs. It does simplify not only our discussion in this paper, but also the implementation of Frogingo. Therefore, we are going to give the reduction rules and its semantics only on closed terms.

DEFINITION 25 (*F*-redex) Given a database *D*, a *F*-term *t* is an *F*-redex if one of the following conditions is not true:

1. a constant,
2. a variable,
3. a derivable assignee under an environment $m\ [\overline{x} := \overline{v}]$, here each *v* in $\overline{v}$ is not a *F*-redex, and $|\overline{x}| >= 0$.

DEFINITION 26 (WHNF, weak head normal forms) A term is a weak head normal form, denoted as WHNF, if it doesn't include a *F*-redex as a sub-term.

Clearly, a WHNF is a constant, a variable, a derivable assignee, or a derivable assignee under an environment.

We used the notion "weak head normal form" to indicate that Froglingo's implementation strategy can be comparable to those for conventional functional programming languages [25]. But we will not discuss in detail on how they are comparable in implementations.

To simplify the discussion in this paper, we only give a reduction strategy that is similar to the leftmost and outermost reduction strategy in the lambda calculus, which guarantees that a normal form can be reached if it does exist for a term.

DEFINITION 27 (one-step reduction rules, leftmost and outermost) Given a *D* and a *F*-term *t*, we have the following one-step reduction rules, denoted as $\rightarrow$:

1. A constant is reduced to itself, i.e., $c \in C \Rightarrow c \rightarrow c$.

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

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

Otherwise, $p \rightarrow p$.

3. If $t \equiv m\ n$ and m is not a WHNF, then $t \rightarrow (m'\ \varepsilon)\ n$, here $m \rightarrow m'\ \varepsilon$ by induction, ε will by empty when m' is a closed term.
4. If $t \equiv (m\ \varepsilon_1)\ n$, and $m\ \varepsilon_1$ is a WHNF and n is not a WHNF, $t \rightarrow (m\ \varepsilon_1)\ (n'\ \varepsilon_2)$, here by induction, $n \rightarrow n'\ \varepsilon_2$.
5. If $t \equiv (m\ \varepsilon_1)\ (n\ \varepsilon_2)$, here $m\ \varepsilon_1$ and $n\ \varepsilon_2$ are WHNFs,

5.1. if $m\ n$ is a assignee in D, then $t \rightarrow (m\ n)$. Note that $\varepsilon_1$and ε_2 must be empty, by induction.

5.2. else if there is a variable *$x* such that m *$x* is an assignee in D, then $t \rightarrow (m\ \$x)\ (\varepsilon_1 \cup [\$x{:=}\ (n\ \varepsilon_2)])$

5.3. else $t \rightarrow null$.

6. if t is reduced to $t'\ [\overline{x} := \overline{v}]$, and t' is an explicit assignee such that: $t' := t''$, then $t'\ [\overline{x} := \overline{v}] \rightarrow (\overline{x} :\rightarrow \overline{v})\ t''$.
7. A variable is reduced to itself.

EXAMPLE 28 (reductions) $fac\ 1 \rightarrow fac\ \$x\ [\$x := 1]$, by 27.5.2
$\rightarrow (\$n :\rightarrow 1)\ \$n * (fac\ (\$n - 1)) \equiv 1 * (fac\ (1 - 1))$, by 27.6
$\rightarrow 1 * (fac\ 0)$, by $1 - 1 = 0$, a rule not included in Definition 27.
$\rightarrow 1 * (fac\ 0\ [\ ])$, by 27.5.1
$\rightarrow 1 * 1$, by 27.6

DEFINITION 29 (reduction process and termination) Let $m, n \in F$ with a given database D. If there is a finite sequence $l_0, \ldots, l_q \in F$, where $q \geq 0$, such that $m \equiv l_0,\ l_0 \rightarrow l_1, \ldots, l_{q-1} \rightarrow l_q,\ l_q \equiv n$, then
1. m is effectively, i.e., in finite steps, reduced to n, written as $m \rightarrow_F n$.
2. If $m_1 \rightarrow_F n$ and $m_2 \rightarrow_F n$, then we say that m_1 is equal to m_2, denoted as $m_1 == m_2$.
3. If n is a WHNF, then the evaluation process terminates, and we say that m has a WHNF n.

Clearly, $\rightarrow_F$ includes $\rightarrow_{EP}$. The notations $\rightarrow$ and $==$, the relations originally defined for the EP data model in Definition 9, are reused and extended for Froglingo.

EXAMPLE 30 (a non-termination reduction process) Let's define

$$w\ \$x := \$x\ \$x$$

in correspondence to the lambda expression $\lambda x.\ (x\ x)$. We show that $w\ w$, doesn't terminate:
$w\ w \rightarrow w\ \$x\ [\$x{:=}w]$, by 24.3.2.2
$\rightarrow (\$x :\rightarrow w)\ (\$x\ \$x) \equiv w\ w$, by 24.4
…

Before we prove that a term ends up with a unique WHNF with the reduction rules in Definition 27 if it does have one, we prove that an assignee in a database is always unique and therefore a WHNF is always unique too.

COROLLARY 28 1 (derivable assignee uniqueness) Given a database D, a derivable assignee m is not reducible, i.e., $m \in D,\ m \rightarrow_F n \Rightarrow m \equiv n$.
2 (derivable assignee under environment uniqueness) Given a database D, a derivable assignee under environment $m\ [\overline{x} := \overline{v}]$, where each v in $\overline{v}$ is a WHNF, i.e., $m\ [\overline{x} := \overline{v}]$ is a WHNF, it is not reducible.

PROOF 1 (derivable assignee uniqueness). 1. If m is an identifier, then m is reduced to itself by 27.2.

4 If $m \equiv a\ b$, then a is a derivable assignee by Definition 4.1. By induction, assume it is not reducible. We prove $a\ b$ is not reducible. First, we prove that b is not reducible,

a). If b is a variable, it is not reducible, by Definition 27.7
b).Otherwise, b is also an derivable assignee by Definition 4.2. Therefore, b is not reducible by induction.
Since $a\ b$ is a derivable assignee, it is not reducible by 27.5.1.

2 (derivable assignee under environment uniqueness) Since m is a derivable assignee (with variable), it is unique by Definition 4.5, and it is not reducible by Corollary 28.1 above. Given an environment $[\overline{x} := \overline{v}]$, it is unique by itself because each value v in $\overline{v}$ is a WHNF. Since there is not a rule in Definition 27 to reduce $m\ [\overline{x} := \overline{v}]$ further, $m\ [\overline{x} := \overline{v}]$ is unique.

COROLLARY 28' (WHNF uniqueness) Given a database D, if two derivable assignees m an n are two distinguished WHNFs, i.e., $m\ !{\equiv}\ n$, then there are not rules in Definition 27 that reduce them to be equal, i.e.,

$$m\ !{\rightarrow_F}\ n \cap n\ !{\rightarrow_F}\ m$$

PROOF 1. A constant is not further reduced according to 27.1. If two constants are distinguished, then there is no rules in 27 that make them equal.
2. Given two distinguished derivable assignees, each derivable assignee is not reducible by Corollary 28. They could be identical only if two were identical in the given database D. It is not possible by Definition 4.5.
3. Since both constants and derivable assignees are not reducible, one constant and one derivable assignee cannot be equal since there is no rules in Definition 27 to make it happen.

COROLLARY 31 (reduction to unique WHNF) A closed F-term t under a database D has at most one WHNF.

PROOF. We have proved in Corollary 28' that each WHNF is unique. Since the reduction order in Definition 27 is fixed, each reduction would automatically result in a unique value. The remaining work is to prove that if a reduction process by the rules in Definition 27 terminates with a WHNF if it terminates.

1. If t is a constant or an identifier, then the process terminates with the constant as the WHNF by 27.1 and 27.2.
2. If $t \equiv m\ n$, m will be repeatedly reduced first by Definition 27. If the process does terminate, i.e., $m \rightarrow_F$ m' ε_1, here m' ε_1 is a WHNF, then

2.1. if m' ε_1 is a constant, then the entire process terminates with *null* (by Definition 4.1, i.e., a constant cannot be an assignee, and by 27.5.3), i.e.,

$$m' \in C,\ \varepsilon_1 = \phi \Rightarrow t \rightarrow_F null.$$

The process terminates with the WHNF *null*.

2.2. else we start to evaluate n. By induction, if the process terminates, i.e., $n \rightarrow_F n'\ \varepsilon_2$, here $n'\ \varepsilon_2$ is a WHNF, then

2.2.1. if $m'\ n'$ is in D, i.e. $m'\ n' \in D$, (by induction, ε_1 and ε_2 must be empty), then $t \rightarrow_F m'\ n'$, by 27.5.1. If the result $m'\ n'$ is a derivable assignee, the process will terminate with the derivable assignee as the WHNF. Otherwise ($m'\ n'$ is an explicit assignee) it will continue.

2.2.2. else if m' $\$x$ is in D, then by 27.5.2, $t \rightarrow (m'\ \$x)\ \varepsilon_1 \cup [\$x{:=}\ (n'\ \varepsilon_2)]$. If $(m'\ \$x)$ is a derivable assignee, the process will terminate with the WHNF: $(m'\ \$x)\ \varepsilon_1 \cup [\$x{:=}\ (n'\ \varepsilon_2)]$. Otherwise, i.e., $(m'\ \$x)$ is an explicit assignee, the process will continue.

2.2.3. else by 27.5.3, $t \rightarrow$ *null*. The process terminates with the WHNF *null*.

3. if t is reduced to $t'\ [\bar{x} := \bar{v}]$, and t is an explicit assignee such that: $t' := t''$, then by 27.6, $t\ [\bar{x} := \bar{v}] \rightarrow (\bar{x} :\rightarrow \bar{v})\ \ t''$. The process will continue.

The reduction strategy defined by Definition 27 is restricted to the leftmost and outermost order and ignored other possible reduction orders. We only discuss this specific reduction order in this paper by requesting that the reduction strategy in Section 6 and 7 for lambda expressions is also restricted to the leftmost and outermost order. An additional work is needed to show that Froglingo also obeys a reduction behavior similar to the Church-Rosser reduction behavior of the lambda calculus.

A formal theory with equations as formulas is consistent if the formal theory doesn't prove every equation. In other words, given the Froglingo with the syntax definition in Definitions 2, 4, 14, and 15, and the equation formulas, ==, defined in Definition 29, we will see that Froglingo is consistent if there is no reduction rules to reduce one WHNF to another WHNF.

THEOREM 32 (consistency) Froglingo is consistent.

PROOF. Given a database D, let N be the complete set of WHNFs. We need to show that at least two WHNFs are not reducible.

1. Assume N has at least two elements, then Corollary 28' has already proved that the two elements are not reducible.
2. To make sure that N has at least two elements, we first require that *null* is a mandatory constant because the rules in Definition 27 used *null*. Then the consistency is satisfied if we add an additional WHNF from either a constant in C or from a non empty database D.

5. Semantics

In this section, we will show that a F-term under a database can be mapped to a lambda expression, and two equal terms in Froglingo will be equal in the lambda calculus as well after converted to lambda expressions. This will result in the soundness if Froglingo. We start from the syntax of the lambda calculus first.

Through the discussion, we shall see that the EP data model is the system that makes database management easier. Many simple expressions in the EP data model have to converted to very complex lambda expressions with multiple fixed point combinators.

NOTATION 33 (lambda terms) Terms $\wedge$, ranged over by t, are given by the grammar

$$t ::= x \mid \lambda x.t \mid (t\ t)$$

where x ranges over a set of variables V.

Mapping F-terms to lambda expressions will take multiple steps. Given a F-term a, we use a^λ for the corresponding λ-expression.

DEFINTION 34 (mapping variables) a variable x in F is also a variable in $\wedge$, i.e., $x^\lambda = x$.

DEFINTION 35 (mapping constants) 1. Each constant c in C is mapped to a closed term c^λ that doesn't have a head normal form in $\wedge$. By denoting C^λ for the entire set of such lambda terms, we further require that each c^λ for a c is distinguished in C^λ.

2. The special constant *null* is mapped to $\Omega \equiv (\lambda x.(x\ x))\ (\lambda x.(x\ x))$.

Since each element in C^λ is unique and doesn't have a normal form, we often use Ω_i to denote an element. A sample definition of such lambda expression is $\Omega_i \equiv (\lambda x.(x\ x\ i^\lambda))\ (\lambda x.(x\ x\ i^\lambda))$, here i is a nature number, and i^λ is a lambda expression modeling the number. Also we use Ω_{null} for the F-term *null*.

We chose a term without head normal form for a constant to counterpart the reduction rule specified in Definitions 8.1 and reiterated in 27.5.3. The choice of resulting in *null* from applying *null* to an arbitrary term was to simplified our discussion. In reality, many built-in operators such as those ordering relations in Definitions 2.11 and 2.12 are constants and should have been mapped to closed terms that have head normal forms.

DEFINTION 36 (mapping identifiers not defined in database) Given a database D, each identifier p not defined in D is mapped to Ω_{null}.

NOTATION 38 (derivable assignee set) Given a database D,

1. We use DA for the complete set of the derivable assignees in D.
2. We use EA for the complete set of the explicit assignees in D.
3. The remaining F-terms, denoted as F_-, that we haven't counted is: $F - (\text{DA} \cup \text{EA} \cup C^\lambda)$.
4. Given a set $X \in F$, we use X^λ for the set of the lambda expressions mapped from the elements in X. (We use |X| for the size of the set, i.e., the number of elements in the set.)

Given a derivable assignee m in a database D, the plan to map m is to find all the n_i, here i is zero or a positive number, such that $m\ n_i$ is in D. By induction, we assume the n_i and $m\ n_i$ had already being mapped to n_i^λ and $(m\ n_i)^\lambda$, then m can be written as a function consisting a set of argument and value pairs:

NOTATION 38X (idea of mapping F-term to λ-term).
$m = \{<n_1, m\ n_1)>, \ldots, <n_k, m\ n_k>\}$. The idea of mapping the function m is the following:

$$\begin{aligned} m^\lambda = \lambda arg.\ &\text{if } arg = n_1^\lambda \text{ then } (m\ n_1)^\lambda \\ &\ldots \\ &\text{else if } arg = n_k^\lambda \text{ then } (m\ n_k)^\lambda, \\ &\text{else } \Omega_{null}. \end{aligned}$$

Recall that a derivable assignee d in D is defined by Definition 27 as a unique WHNF, even if there is not any term t such that $d\ t$ is in database. It means that each identifier in derivable assignee must have a unique lambda expression to make a sound mapping. However the mapping will not work because two derivable assignees which are defined with the same set of argument and value pairs would be calculated with the same lambda expression. To resolve the issue, we add an extra pair that can make each d unique among DA in D.

DEFINITION 38XX (numbering DA symbols) Given DA under a database D, we find a set of lambda expressions, denoted as $(\#\mathrm{DA})^{\lambda}$, such that each element $d \in$ DA has an element, denoted as $\Omega_{\mathrm{d}} \in (\#\mathrm{DA})^{\lambda}$, that Ω_{d} is unique and has no normal form.

Then we can add another pair $< \Omega_{\mathrm{d}}\ \Omega_{\mathrm{d}}>$ to Notation 38X.

The biggest challenge in the mapping is on derivable assignees that may include self-applications and circular references among them when they are defined in a database. For example, a database like: $\{A\ B := B;\ B\ A := A;$ and $A\ A := M;\}$, in which the first two form a bidirectional circular reference, and the third one is a self-application. If that is the case, the size of the lambda expression in Notation 38X will grow infinite. To overcome this problem, we are going to use multiple fixed point combinators [5] and [17]. In preparing the mapping by using multiple fixed point combinators, we need to predefine a set of variables dedicated, i.e., never being used for other purpose, for the derivable assignees in DA.

DEFINITION 39 (dedicated variables for DA) Given DA under a database D, we choose a set χ such that $|\ \chi\ | = |\mathrm{DA}|$ and each variable $x \in \chi$ is different from others appeared in D. Further we write $\chi\ (d_i)$ for the variable in corresponding to $d_i \in$ DA.

An assignee under an environment, i.e., $M\ [\bar{x} := \bar{v}]$, appears not a *F*-term, but it is actually equivalent to a *F*-term. Therefore a term with the form of $M\ [\bar{x} := \bar{v}]$ has a corresponding lambda expressions. The following equations are taken as granted due to the syntactical structure of an environment defined in Definition 19:

NOTATION 40X (mapping $m\ [\bar{x} := \bar{v}]$)

$$(\mathrm{m}\ \varepsilon)^{\lambda} = \mathrm{m}^{\lambda}\ \varepsilon^{\lambda}$$
$$[\bar{x} := \bar{v}]^{\lambda} = [\bar{x} := \bar{v}^{\lambda}]$$

Now we are ready to map DA, EA, and *F*-. We start with explicit assignees first. For each explicit assignee, we want the occurrences of all the derivable assignees in the assigner to be replaced with their variables from χ.

DEFINTION 40 (mapping explicit assignees) Let $m\ \bar{y}\ := V \in D$, here $|\ \bar{y}| >= 0$. We have the following definitions:

1. $$(m\ )^{\lambda} = \lambda\,\bar{y}.\ V^{\lambda}$$
2. Let $\bar{d} \equiv \{d_1, \dots, d_j\}$ be the complete set of derivable assignees appeared in v, here j >= 0. We identify the corresponding variables $\bar{x} \equiv \{x_1, \dots, x_j\}$ from χ. Then we substitute all the instances of $\bar{d}$ in v with $\bar{x}$. Then we have:

$$m^{\lambda} = V''\ [\bar{x}\ := \bar{d}^{\,\lambda}]$$

Here $V'' \equiv\ \lambda\,\bar{y}.\ (V')^{\lambda}$, $V \equiv (\bar{x}\ :\rightarrow \bar{d})\ V'$.

For example, Given *fac $n := ($n * (fac ($n – 1))* in D, then $fac^{\lambda} = V''\ [d{:=}fac^{\lambda}]$, here $V'' \equiv \lambda n.\ (n * (d(n\text{-}1)))$. Here V'' will be used in Definition 42, and $V''\ [d{:=}fac^{\lambda}]$ will be used in Theorem 45.

NOTATION 40Y (Mapping *F*-) Given $m\ n \in$ *F*- under a database D, then define

$$(m\ n)^{\lambda} = m^{\lambda}\ n^{\lambda}$$

NOTATION 41 (Curry's multiple fixed point combinators) Given a natural number n, we have the following fixed point combinators:

$$\mathsf{Y}_{Curry}{}^{n}{}_{j} = \lambda f f_2 \ldots f_n.((\lambda x_1 \lambda x_2 \ldots x_n.\ f_j\,(x_1 x_1 \ldots x_n)$$
$$(x_2 x_1 \ldots x_n)$$
$$\ldots$$
$$(x_n x_1 \ldots x_n))$$
$$(\lambda x_1 \lambda x_2 \ldots x_n.\ f_1\,(x_1 x_1 \ldots x_n)$$
$$(x_2 x_1 \ldots x_n)$$
$$\ldots$$
$$(x_n x_1 \ldots x_n))$$
$$(\lambda x_1 \lambda x_2 \ldots x_n.\ f_2\,(x_1 x_1 \ldots x_n)$$
$$(x_2 x_1 \ldots x_n)$$
$$\ldots$$
$$(x_n x_1 \ldots x_n))$$
$$\ldots$$
$$(\lambda x_1 \lambda x_2 \ldots x_n.\ f_n\,(x_1 x_1 \ldots x_n)$$
$$(x_2 x_1 \ldots x_n)$$
$$\ldots$$
$$(x_n x_1 \ldots x_n)))$$

for each nature number j, where $j >= 1$ and $j <= n$.

DEFINTION 42 For each $d_j \in$ DA under a database D, here $0 <= j <= |\mathrm{DA}|$, we find all $d_j\ e_1, \dots, d_j\ e_n$ such that each $d_j\ e_i \in D$, here i and n are natural numbers, and $0 <= i <= n$.

1. We inductively assume that each pair e_i and $d_j\ e_i$ have already been mapped to the lambda expressions: $e_i^{\ \lambda}$ and $(d_j\ e_i)^{\lambda}$.
2. For each $e_i^{\ \lambda}$ obtained above, we find the corresponding variables $x_{ei} \in \chi$.
3. We further define a lambda expression for d_j

$$\begin{aligned} \mathrm{G_j} = \lambda x_1\, x_2\, \ldots\, x_{|DA\,|}.\lambda arg.\ &\text{if } arg \equiv^{\lambda}\ x_{e1}{}^{\lambda} \text{ then } (d_j\ e_1)^{\lambda} \\ &\text{else if } arg \equiv^{\lambda}\ x_{e2}{}^{\lambda} \text{ then } (d_j\ e_2)^{\lambda} \\ &\ldots \\ &\text{else if } arg \equiv^{\lambda}\ x_{en}{}^{\lambda} \text{ then } (d_j\ e_n)^{\lambda} \\ &\text{else if } arg \equiv^{\lambda}\ \Omega_{\mathrm{d}} \text{ then } \Omega_{\mathrm{d}} \\ &\text{else } \Omega \end{aligned}$$

   Here, $x_1,\ x_2,\ \ldots,\ x_{|DA\,|} \in \chi$; $x_{e1}, x_{e2\ldots}, x_{en} \in \chi$; $\Omega \equiv (\lambda x.(xx))\,(\lambda x.(xx))$. The symbol $\equiv^{\lambda}$ is a lambda expression to compare the syntactical body of a parameter of *arg* with G_k that is replicated from an parameter of $x_{ek}{}^{\lambda}$, called a "self-replicator", here 0 <= k <= n. (we didn't provide a definition for $\equiv^{\lambda}$, but take the fact that it exists as granted). The clause of "if", "then", and "else" are the standard lambda expressions for the "if … then … else …" conditional statement.
   (Note that we will have: $d_j = G_j\ d_1\ d_2\ \ldots\ d_j\ \ldots\ d_{|DA\,|}$)
4. $d_j = \mathsf{Y}_{Curry}{}^{n}{}_{j}\ G_1\ \ldots\ G_j\ \ldots\ G_{|DA\,|}$.

In the definition above, G_j was a modification from Notation 38X, which makes every derivable assignee to be defined independent of itself or others that it depends on.

EXAMPLE 43 (*F*-term in lambda expression) Given a database $D = \{A\ B := B;\ B\ A := A;$ and $A\ A := M\}$, we define the following functions:

1. $M^{\lambda} = \Omega_M$.
2. We define G_1 for A and G_2 for B:

$$\begin{aligned} G_1 = \lambda x_1\, x_2.\lambda arg.\ &\text{if } arg \equiv^{\lambda} x_2 \text{ then } x_2 \\ &\text{else } \textit{if } arg \equiv^{\lambda}\ x_1 \text{ then } \Omega_M \\ &\text{else if } arg \equiv^{\lambda}\ \Omega_A \text{ then } \Omega_A \\ &\text{else } \Omega \end{aligned}$$

$$G_2 = \lambda x_1\, x_2.\lambda\ arg.\ \text{if } arg \equiv^{\lambda}\ x_1 \text{ then } x_1$$

else if $arg \equiv^{\lambda}$ Ω_B then Ω_B
else Ω

$d_1 = \mathsf{Y}_{Curry}{}^{2}{}_{1}\ G_1\ G_2$
$d_2 = \mathsf{Y}_{Curry}{}^{2}{}_{2}\ G_1\ G_2$

$$\mathsf{Y}_{Curry}{}^{2}{}_{1} = \lambda f_1 f_2.((\lambda x_1 \lambda x_2.\ f_1\ (x_1 x_1 x_s)\ (x_2 x_1 x_2))\ (\lambda x_1 \lambda x_2.\ f_1\ (x_1 x_1 x_2)\ (x_2 x_1 x_2))\ (\lambda x_1 \lambda x_2.\ f_2\ (x_1 x_1 x_2)\ (x_2 x_1 x_2)))$$

$$\mathsf{Y}_{Curry}{}^{2}{}_{2} = \lambda f_1 f_2.((\lambda x_1 \lambda x_2.\ f_2\ (x_1 x_1 x_s)\ (x_2 x_1 x_2))\ (\lambda x_1 \lambda x_2.\ f_1\ (x_1 x_1 x_2)\ (x_2 x_1 x_2))\ (\lambda x_1 \lambda x_2.\ f_2\ (x_1 x_1 x_2)\ (x_2 x_1 x_2)))$$

3. Then we can call the functions. For example, d_1 $d_1 \equiv \mathsf{Y}_{Curry}{}^{2}{}_{1}$ $G_1\ G_2$ $d_1 \rightarrow_{\lambda} \Omega_M$. During the reduction process, the "self-replicator" was $H_1\ H_1\ H_2$, here $H_1 \equiv \lambda x_1 \lambda x_2.\ G_1\ (x_1 x_1 x_2)\ (x_2 x_1 x_2)$, and $H_2 \equiv \lambda x_1 \lambda x_2.\ G_2\ (x_1 x_1 x_2)\ (x_2 x_1 x_2)$. The replicated G_j was G_1.

COROLLARY 44 (applicative structure preservation)
$M, N \in \mathsf{F}$, $(M\ \ N)^{\lambda} == M^{\lambda}\ N^{\lambda}$.

PROOF 1. If $M\ N$ is an (explicit or derivable) assignee, then $M \in$ DA, here DA is the complete set of the derivable assignees in a database D, then Definition 42 maps all the derivable assignees in DA, including M, to a set of lambda expressions, accordingly including M^{λ}, such that $M^{\lambda}\ N^{\lambda} \rightarrow_{\mathsf{F}} (M\ \ N)^{\lambda}$. Therefore $(M\ \ N)^{\lambda} == M^{\lambda}\ N^{\lambda}$.
2. When M is not an assignee, neither is $M\ N$. Then Definition 40Y says that $(M\ \ N)^{\lambda} == M^{\lambda}\ N^{\lambda}$.

THEOREM 45 (soundness) $\forall$M, N $\in \mathsf{F}$, $\exists D$,
$M ==_{\mathsf{F}} N \Rightarrow M^{\lambda} ==_{\lambda} N^{\lambda}$.

PROOF Given a database D and an arbitrary closed F-term t, we prove that if t' is the result from a one-step reduction, according to the reduction rules in Definition 27, then the lambda expressions t'^{λ} and t^{λ} mapped from t to t', according to the mapping rules in Definitions 34, 35, 36, 40X, 40, 40Y, and 42, will be equal. (See the body of the proof in the Appendix).

The symbol → and == were introduced in Definition 29 between two F-terms. During the proof, we also used them between two λ-terms.

1. If t is a constant, i.e., $t \in \mathsf{C}$, then $t' == t$ by Definition 27.1. By Definition 35, $t \rightarrow_{\mathsf{F}} t^{\lambda}$. Therefore: $t == t \Rightarrow t'^{\lambda} == t^{\lambda}$.
2. If t is an identifier not defined in D,
   $t \rightarrow_{\mathsf{F}} null$, by Definition 27.2.
   $(null)^{\lambda} = \Omega_{null}$, by Definition 35
   $t^{\lambda} = \Omega_{null}$, By Definition 36,
   $t \rightarrow_{\mathsf{F}} null \Rightarrow t^{\lambda} \rightarrow_{\lambda} (null)^{\lambda}$.
3. If $t \equiv m\ n$ and m is not a WHNF, then
   $m\ n \rightarrow_{\mathsf{F}} (m'\ \varepsilon)\ n$, by Definition 27.3.
   $m^{\lambda} == (m'\ \varepsilon)^{\lambda}$, by induction
   $(m\ n)^{\lambda} == m^{\lambda}\ n^{\lambda}$, *by Corollary 44*
   $((m'\ \varepsilon)\ n)^{\lambda} == (m'\ \varepsilon)^{\lambda}\ n^{\lambda}$, *by Corollary 44.*
   $m\ n \rightarrow_{\mathsf{F}} (m'\ \varepsilon)\ n \Rightarrow (m\ n)^{\lambda} == ((m'\ \varepsilon)\ n)^{\lambda}$.
4. If $t \equiv m\ n$, and m is a WHNF and n is not a WHNF, then $m\ n \rightarrow_{\mathsf{F}} m\ (n'\ \varepsilon)$, by Definition 27.4. The proof is done in a similar way as we did for Step 3 above.
5. If $t \equiv (m\ \varepsilon_1)\ (n\ \varepsilon_2)$, here $m\ \varepsilon_1$ and $n\ \varepsilon_2$ are WHNFs, then

5.1. if $m\ n$ is a derivable assignee, then
   $(m\ n) \rightarrow_{\mathsf{F}} (m\ n)$, by 27.5.1
   $(m\ n)^{\lambda} ==_{\lambda} (m\ n)^{\lambda}$

5.2. else if $m\ \$x$ is an assignee, then
   $(m\ \varepsilon_1)\ (n\ \varepsilon_2) \rightarrow_{\mathsf{F}} (m\ \$x)\ \ \varepsilon_1 \cup [\$x{:=}(n\ \varepsilon_2)]$, *by 27.5.2.*
   We need to prove:
   $((m\ \varepsilon_1)\ (n\ \varepsilon_2))^{\lambda} == ((m\ \$x)\ \ \varepsilon_1 \cup [\$x{:=}(n\ \varepsilon_2)])^{\lambda}$
   $((m\ \varepsilon_1)\ (n\ \varepsilon_2))^{\lambda}$
   $= (m\ \varepsilon_1)^{\lambda}\ (n\ \varepsilon_2)^{\lambda}$, by Corollary 44
   $= (m^{\lambda}\ \ \varepsilon_1^{\lambda})\ (n^{\lambda}\ \ \varepsilon_2^{\lambda})$, by Notation 40X
   $((m\ \$x)\ \ \varepsilon_1 \cup [\$x{:=}(n\ \varepsilon_2)])^{\lambda}$
   $= (m\ \$x)^{\lambda}\ \ (\varepsilon_1 \cup [\$x{:=}(n\ \varepsilon_2)])^{\lambda}$, by Definition 40X
   $(m\ \$x)^{\lambda}\ \ = m^{\lambda}\ \$x^{\lambda}$, by Definition 42 (Multi fixed Point)
   $= m^{\lambda}\ \$x$, by Definition 34
   $(\varepsilon_1 \cup [\$x{:=}(n\ \varepsilon_2)])^{\lambda}$
   $= \varepsilon_1^{\lambda} \cup [\$x{:=}(n\ \varepsilon_2)^{\lambda}]$, by Notation 40X
   $(m^{\lambda}\ \$x)\ \ \varepsilon_1^{\lambda} \cup [\$x{:=}(n\ \varepsilon_2)^{\lambda}]$, *by lambda beta reduction*
   $= ((m^{\lambda})\ \ \varepsilon_1^{\lambda} \cup\ [\$x{:=}(n\ \ \varepsilon_2)^{\lambda}])\ \ (\$x\ \ \varepsilon_1^{\lambda} \cup\ [\$x{:=}(n\ \ \varepsilon_2)^{\lambda}])$, by beta reduction,
   $=(m^{\lambda}\ \varepsilon_1^{\lambda})\ (n\ \varepsilon_2)^{\lambda}$, By lambda beta-reduction.
   $= (m^{\lambda}\ \ \varepsilon_1^{\lambda})\ (n^{\lambda}\ \ \varepsilon_2^{\lambda})$, by Notation 40X
   $((m\ \varepsilon_1)\ (n\ \varepsilon_2))^{\lambda} == ((m\ \$x)\ \ \varepsilon_1 \cup [\$x{:=}(n\ \varepsilon_2)])^{\lambda}.$

5.3. else (if there is no variable $\$x$ such that $m\ \$x$ is an assignee, i.e., $(m\ \varepsilon_1)\ (n\ \varepsilon_2)$ is not a WHNF), then
   $(m\ \varepsilon_1)\ (n\ \varepsilon_2) \rightarrow_{\mathsf{F}} null$, by 27.5.3.
   We need to prove:
   $((m\ \varepsilon_1)\ (n\ \varepsilon_2))^{\lambda} == null^{\lambda}$
   $((m\ \varepsilon_1)\ (n\ \varepsilon_2))^{\lambda}$
   $=(m^{\lambda}\ \ \varepsilon_1^{\lambda})\ (n^{\lambda}\ \ \varepsilon_2^{\lambda})$, by Notation 40X
   $= (m^{\lambda}\ \ n^{\lambda})\ (\varepsilon_1^{\lambda} \cup \varepsilon_2^{\lambda})$, by the lambda beta conversion.
   Since $m\ \varepsilon_1$ is a WHNF, m is a derivable assignee. Then
   $m^{\lambda}\ \ n^{\lambda} = \Omega$, by Definition 42.
   $((m\ \varepsilon_1)\ (n\ \varepsilon_2))^{\lambda}$
   $= \Omega\ (\varepsilon_1^{\lambda} \cup \varepsilon_2^{\lambda})$
   $= \Omega$, by the lambda beta reduction
   $null^{\lambda} = \Omega$, by Definition 35.
   $((m\ \varepsilon_1)\ (n\ \varepsilon_2))^{\lambda} == null^{\lambda}$

6. if t is reduced to $t'\ [\overline{x} := \overline{v}]$, and t' is an explicit assignee such that: $t' := t''$, then
   $t'\ [\overline{x} := \overline{v}] \rightarrow (\overline{x} :\rightarrow \overline{v})\ \ t''$, by Definition 27.6.
   We need to prove:
   $(t'\ [\overline{x} := \overline{v}])^{\lambda} == ((\overline{x} :\rightarrow \overline{v})\ \ t'')^{\lambda}$
   $(t'\ [\overline{x} := \overline{v}])^{\lambda}$
   $= t'^{\lambda}\ [\overline{x} := \overline{v}]^{\lambda}$, by Notation 40X.1
   $((\overline{x} :\rightarrow \overline{v})\ \ t'')^{\lambda}$
   $= (\overline{x} :\rightarrow \overline{v})^{\lambda}\ \ t''^{\lambda}$, by 40X.1
   $= t''^{\lambda}[\overline{x} := \overline{v}]^{\lambda}$
   Now we need to prove:
   $t'^{\lambda} =\ t''^{\lambda}$

Since (t' := t'') (rewritten as $k\,\bar{z} := t''$) $\in D$, Definition 40.2 defines the mapping for the assigner given an assignee. Let $\bar{d}$ be the complete set of all the derivable assignees in t'', and $\bar{y}$ be the corresponding variables from χ. From Definition 40.2, we have,

$$t'^{\lambda} \equiv (k\,\bar{z})^{\lambda} = (V''\,[\bar{y} := \bar{d}^{\lambda}])\,\bar{z}^{\lambda}$$

Here $V'' \equiv \lambda\bar{z}.(V')^{\lambda}$, $t'' \equiv (\bar{x} :\rightarrow \bar{d})\,V'$. Then

t'^{λ}

$= (V''\,[\bar{y} := \bar{d}^{\lambda}])\,\bar{z}^{\lambda}$, by the formula above

$= (V''\,[\bar{y} := \bar{d}^{\lambda}])\,\bar{z}$, by Definition 34

$\equiv (\lambda\bar{z}.(V')^{\lambda}\,[\bar{y} := \bar{d}^{\lambda}])\,\bar{z}$, by the formula above

$= (\lambda\bar{z}.((V')^{\lambda}\,[\bar{y} := \bar{d}^{\lambda}]))\,\bar{z}$, by the lambda beta conversion, because $\bar{y}$ are free in $\lambda\bar{z}.(V')^{\lambda}$

$= (\lambda\bar{z}.(((V')\,[\bar{y} := \bar{d}])^{\lambda}))\,\bar{z}$, by Notation 40X

$= (\lambda\bar{z}.t''^{\lambda})\,\bar{z}$, by the lambda beta reduction

$= t''^{\lambda}$

6. Computability

So far, we have considered constants as a part of the theory. Constants, specially *null*, is a critical component of the EP data model. Without *null*, we can still construct an EP database, such as the one for circular directed graph in Example 5.2. But the reduction process in Definition 8 would be stuck when a term was not defined as an assignee in a database. This is exactly the reason that we map *null* to a λ-term Ω_{null} that doesn't have a normal form in Section 6.

To have an exact correspondence with the lambda calculus, we exclude constants from Froglingo. That is:

DEFINITION 14' (*F*-terms without constants) Terms *F*, ranged over by *t*, are given by the grammar

$$t ::= x \mid a \mid (t\ t)$$

where *x* ranges over a set of variables *V*, and *a* ranges over a set of identifiers *P*.

Without constants, we will continue to use the reduction rules in Definition 27 except that a reduction will not terminate if the special constant *null* is encountered.

Excluding constants doesn't mean that we will not use constants like 3.14 or "a string" in examples. We view the constant symbols as identifiers that take built-in function as default meanings and will never by explicitly defined or having their meanings altered via a database.

In an assignment of a database, a variable is always bounded, i.e., if a variable is in the assigner, it must be in the assignee. an assignment with variables is comparable to a closed abstraction in the lambda calculus. Therefore we must convert all the abstractions with free variables to closed abstractions.

DEFINITION 47 (closed abstractions) 1. Given a lambda expression *t*, i.e., $t \in \wedge$, we define the following rules to convert *t* to another term, denoted as t^0, such that each abstraction in *e* is closed, i.e., $\forall e \in \mathrm{SUB}(t^0)$, $e \in \wedge^0$:

1. $x^0 = x$, here *x* is a variable.
2. $(a\ b)^c == a^0\ b^0$.
3. For an abstraction $\lambda\bar{x}.m$, let $\bar{y}$, here $|\bar{y}| >= 0$, is the complete list of free variables in *m*, i.e., $\bar{y} \in \mathrm{FV}(m)$, define

$$(\lambda\bar{x}.m)^0 = (\lambda\bar{y}.\ \lambda\bar{x}.\ m^0)\ \bar{y}.$$

For example, $(\lambda x.\ (lz.\ y\ x)\ \ x)^0 = (\lambda h.\ (\lambda x.\ (\lambda z.\ h\ z)\ \ x))\ y$.

From now on, we assume all the closed λ-terms, before being mapped to a *F* -term, are converted to one in which each abstraction is closed.

DEFINITION 48 (mapping λ-terms) Given a database *D*, a λ-term *t*, we define the following rules to convert *t* to a *F* -term during which *D* is enhanced to be *D*':

1. $x^f = x$,
2. $(a\ b)^f = a^f\ b^f$
3. $(\lambda\bar{x}.m)^f = p$, here $|\bar{x}|>0$, $p \in P$, and *p* is obtained by the following process: If $\lambda\bar{x}.m$ has been mapped to $p' \in D$, then $p \equiv p'$, and $D' \equiv D$. Otherwise, we find a $p \in P$, $p \notin D$, such that $D' = D \cup (p\ \ \bar{x} := m^f)$. Further remember $(\lambda\bar{x}.m)^f$ has been mapped to *p* in *D*.

Since $(\bar{x}_1 :\rightarrow \bar{n})\ m$ is nothing by a lambda expression, the mapping can be distributed into $(\bar{x}_1 :\rightarrow \bar{n})$ and *m* according to Definition 48.2. We record it as a notation:

NOTATION 49 (mapping distribution)

$$((\bar{x}_1 :\rightarrow \bar{n})\ m)^f \equiv (\bar{x}_1{}^f :\rightarrow \bar{n}^f)\ m^f$$

EXAMPLE 50 (sample mappings) $(\lambda z.\ z\ (\lambda x.\ x\ x))^f = p$,
here $D = \{p\ \$z := \$z\ \ q;\ q\ \$x := \$x\ \$x\}$.

We will show that when two λ-terms are convertible under the β-reduction rule, their corresponding *F*-terms are convertible as well in Froglingo. To do this, we need to extend the rules in Definition 27, where given an assignment, e.g., ($m\ \bar{x}$:= n), applying a derivable assignee, e.g., $m\ \bar{y}$, here $\bar{y}$ is a sub sequence of $\bar{x}$ and $|\bar{y}| < |\bar{x}|$, to a too few arguments, e.g., $\bar{n}$, here $|\bar{n}| = |\bar{y}|$, results in its own derivable assignee with an environment as its WHNF, e.g., $(m\ \bar{y})\ [\bar{y} := \bar{n}]$. Given the assignment *g \$x \$y* := (*\$x* + *\$y*), for example, *g 6* is converted to *g \$x* [*\$x* := *6*] and no further reduction is performed. To extent the reduction to the body of the assigner, Definition 27.6 is extended as a new rule:

DEFINITION 51 (intermediate reduction result) Given a database *D*, if *t*, a *F*-term, is reduced to $t'\ [\bar{x} := \bar{v}]$, and $t'\ \bar{z}$, here $|\bar{z}| > 0$, is an explicit assignee, i.e., ($t'\ \bar{z} := t''$) $\in D$, then we do the followings:

1. If there is $p \in P$, $p \in D$, such that $(p\ \ \bar{z} := (\bar{x} :\rightarrow \bar{v})\ \ t'') \in$ D, then *p* is returned as the WHNF. Otherwise,
2. Enhance *D* with $(p\ \ \bar{z} := (\bar{x} :\rightarrow \bar{v})\ \ t'')$, here *p*, an identifier, was not in *D* before. Then return *p*.

For a *F*-term *g 6* with (*g \$x \$y* := (*\$x* + *\$y*)) in a database *D*, for example, we will temporarily add a new assignment *p \$y* := (6 + \$y) to *D*.

By the way, the rule above is not necessary in an implementation of Froglingo. It is defined solely for the purpose of relating Froglingo with the lambda calculus.

COROLLARY 52 (β-reduction and equivalence) For a closed term $(\lambda\bar{x}.m)\ \bar{n}$, here $\bar{x} \equiv \bar{x}_1\,\bar{x}_2$, $|\bar{x}_1| = |\bar{n}|$, and an empty database $D = \phi$,

$$((\lambda\bar{x}.m)\bar{n})^f = (\lambda x_2.((\bar{x}_1 :\rightarrow \bar{n})\ m))^f.$$

PROOF $((\lambda\bar{x}.m)\bar{n})^f$

$= (\lambda\bar{x}.m)^f\ \bar{n}^f$, by Definition 48.2.

$=\ t\ \bar{n}^f$, here $t \in P$, $t \notin D$, and

D is added with $t\ \bar{x} := m^f$, by Definition 48.3.

$= t\ [\bar{x}_1 := \bar{n}^f]$, by Definition 27.5.2

Case 1: $|\bar{x}_2| = 0$
$t\,[\bar{x}_1 := \bar{n}^{\mathrm{f}}]$
$= (\bar{x}_1 : \rightarrow \bar{n}^{\mathrm{f}})\; m^f$, by Definition 27.6.
$(\lambda x_2.((\bar{x}_1 : \rightarrow \bar{n})\; m))^{\mathrm{f}}$
$= ((\bar{x}_1 : \rightarrow \bar{n})\; m)^{\mathrm{f}}$, since $|\bar{x}_2| = 0$
$= (\bar{x}_1 : \rightarrow \bar{n}^{\mathrm{f}})\; m^f$, by Definition 48.2.
$((\lambda\bar{x}.m)\bar{n})^{\mathrm{f}} = (\lambda x_2.((\bar{x}_1 : \rightarrow \bar{n})\; m))^{\mathrm{f}}$ is true.
Case 2: $|\bar{x}_2| > 0$
$t\,[\bar{x}_1 := \bar{n}^{\mathrm{f}}]$
$= p$, here $(p\;\bar{x}_2 := (\bar{x}_1 : \rightarrow \bar{n}^{\mathrm{f}})\; m^f) \in D$, by Definition 51.
$(\lambda x_2.((\bar{x}_1 : \rightarrow \bar{n})\; m))^{\mathrm{f}}$
$= p$, by Definition 48.3.
$((\lambda\bar{x}.m)\bar{n})^{\mathrm{f}} = (\lambda x_2.((\bar{x}_1 : \rightarrow \bar{n})\; m))^{\mathrm{f}}$ is true.

Most languages in practice don't have a concept comparable to η-reduction of the lambda calculus, i.e., $\lambda x.Mx \rightarrow M$, here x $\notin$ FV(M). This is especially a case in Froglingo. For example, one may create *p \$x := q \$x* in a database first, and then add *p 3 := 6* later. Then most likely *p* and *q* are not equal. Actually, two arbitrary identifiers (or generally two derivable assignees in a database), are two distinguished WHNFs, and intentionally make them not convertible.

To ensure Froglingo's full correspondence with the lambda calculus in computability, we need a special attention to avoid η-redex before a lambda expression is converted to a *F*-term, as if a programmer needed a special attention to avoid a doubled effort of writing two procedures for the exactly same function. To this end, we add another requirement to automatically remove η-redex for our λ-term to *F*-term mapping process:

DEFINITION 53 (η-redex elimination) Given a lambda expression *t*, i.e., $t \in \wedge$, we define the following rules to convert *t* to another term *t*', denoting the conversion process as

$$t \rightarrow_{\eta\text{-f}} t', \text{ or}$$
$$t^{\eta\text{-}} = t',$$

such that t' doesn't contain a η-redex as a sub term:
1. if $t \equiv \lambda x.Mx$, here x $\notin$ FV(M), then $t^{\eta\text{-}} = M^{\eta\text{-}}$
2. if $t \equiv x, x \in V$, $t^{\eta\text{-}} = t$
3. if $t \equiv m\; n$, $m, n \in F$, $t^{\eta\text{-}} = m^{\eta\text{-}}\; n^{\eta\text{-}}$
4. $t \equiv \lambda x.M$, $M \in F$, $t^{\eta\text{-}} \equiv \lambda x.M^{\eta\text{-}}$

We understand that a λ-term that doesn't contain a η-redex may be reduced to a term containing a η-redex. For example, $\lambda x.((\lambda yz.z)xM)x$ doesn't contain a η-redex. But it can be converted to $\lambda x.Mx$, a η-redex. Therefore the definition 53 needs to applied whenever a lambda expression is to be converted to a *F*-term.

Again the definition above is not a part of Froglingo implementation. We defined it here to analyze Froglingo's computability and will be referenced in Corollary 57.3. In practice, we rely on programmers to write one function only once.

The α–reduction rule in the lambda calculus, i.e., $\lambda x.\ M \rightarrow ly.\ M[x{:=}y]$, has never been discussed. We simply assume that each variable is unique in the context where it is used.

Now let's show that the concepts of normal forms and head normal forms in the lambda calculus have correspondence in Froglingo and therefore we prove that Froglingo is Turing-complete while the lambda calculus is Turing-complete. The result should not be any surprise. But by doing so, we have a better understanding on Froglingo by relating it with the lambda calculus.

DEFINITION 54 (redex) Given a database *D*, a closed term $t \in F$ is an redex if
1. If *t* is an identifier not defined in *D* to represent an abstraction, then it is a redex, i.e.,

$$t \equiv p, \text{ here } p \in P,\ \forall \bar{x}, v \in F,\ |\bar{x}| > 0,\ (p\;\bar{x} := v) \notin D.$$

2. If t is an application, then it is a redex,

$$t \equiv p\;\bar{m}, \text{ here } p \in P,\ \bar{m} \in F,\ |\bar{m}| > 0$$

DEFINITION 55 Given a database *D*,
1. (normal form, NF) a term $t \in F$ is a normal form if it doesn't contain a redex as a sub term.
2. (head normal form, HNF) a term $t \equiv a\;\bar{m} \in F$, *t* is a head normal form if *a* is not a redex.

Here we map some standard lambda expressions to F-term as examples.

EXAMPLE 56 Given a database *D* =
{ *w \$x := \$x \$x;*
w' \$y \$x := y(xx);
Ω^{f} *:= w w;*
FIX$^{\mathrm{f}}$ *\$f := (w' f) (w' f);*
A B := B;
B A := A;
}, where *w, w', A, B,* Ω^{f}, *FIX*$^{\mathrm{f}}$ $\in P$, *\$x, \$y* $\in V$.
Then *w, A, \$x, w'* are in normal forms. The terms *w \$x, A B,* and Ω^{f} are not in normal forms.

COROLLARY 57 Given a term *t*, $t \in \wedge$, and a database D,
1. (redex correspondence) *t* is a redex if and only if t^{f} is a redex.
2. (HNF correspondence 1) *t* is a HNF if and only if t^{f} is a HNF.
3. (HNF correspondence 2) *t* has a HNF if and only if t^{f} has a HNF.
4. (NF correspondence 1) *t* is a NF if and only if t^{f} is a NF.
5. (NF correspondence 2) *t* has a NF if and only if t^{f} has a NF.

PROOF 1 (redex correspondence)
a) If *t* is a redex, i.e., $t \equiv (\lambda x.M)\; N$. then
$t^{\mathrm{f}} \equiv ((\lambda x.M)\; N)^{\mathrm{f}}$
$= (p\; N^{\mathrm{f}})$, here $(p\;\$x := M) \in D$
Then t^{f} is a redex in Froglingo.
b) Let *t* be a *F*-term, and it is a redex in Froglingo,
Case 1: $t \equiv p$, here $p \in P,\ \forall \bar{x}, v \in F,\ |\bar{x}| > 0,\ (p\;\bar{x} := v) \notin D$,
When *p* is in *D*, $p^{\lambda} = \Omega_{\mathrm{p}}$, by Definition 42.
When *p* is not in *D*, $p^{\lambda} = \Omega_{\mathrm{null}}$, by Definition 36.
Ω_{p} and Ω_{null} are not redex.
Case 2: $t \equiv p\;\bar{m}$, here $p \in P,\ \bar{m} \in F,\ |\bar{m}| > 0$
$(p\;\bar{m})^{\lambda} = p^{\lambda}\;\bar{m}^{\lambda}$, by Corollary 44
$p^{\lambda} \equiv \lambda x.M$, *for some* $M \in F$, by Definition.
We have proved that t^{λ} is a redex.

PROOF 2 (HNF correspondence 1) If a lambda term *t* is in HNF, then it doesn't contain a redex. According to Corollary 57.1, t^{f} will not contain a redex either. If a *F*-term *t* is in HNF, t^{λ} will not contain a redex according to Corollary 57.1. Proved.

PROOF 3 (HNF correspondence 2). If a lambda term *t* has a HNF *t*', then $(t')^{\mathrm{f}}$ is the HNF for t^{f} by 57.2. We know that $t \rightarrow_{\lambda} t'$ is done by β-reduction and η-reduction. For each reduction step $m \rightarrow_{\lambda} n$, here $t \rightarrow_{\lambda} m$,
Case 1: $m \rightarrow_{\lambda} n \equiv m \rightarrow_{\beta} n$. By Corollary 52, we have
$m^{\mathrm{f}} \rightarrow_{\mathrm{f}} n^{\mathrm{f}}$
Case 2: $m \rightarrow_{\lambda} n \equiv m \rightarrow_{\eta} n$. By Definition 53, we have
$m^{\mathrm{f}} \rightarrow_{\eta\text{-f}} n^{\mathrm{f}}$.
This has proved that if a λ-term *t* has a HNF *t*', then t^{f} can be effectively reduced to $(t')^{\mathrm{f}}$ as the HNF.

Reversely, if t, $t \in F$, has a HNF t', i.e., $t == t'$, we prove that t^{λ} has a HNF $(t')^{\lambda}$. When t' is a HNF, so is $(t')^{\lambda}$ in Froglingo by Corollary 47.2. Now we need to prove that $t^{\lambda} == (t')^{\lambda}$. That is true by Theorem 45.

PROOF 4 (NF correspondence 1). Similar to Proof 2

PROOF 5 (NF correspondence 2). Similar to Proof 3

Froglingo is Turing-complete since the lambda calculus is Turing-complete, and we proved in this paper that Froglingo, with Definition 53, is equivalent to the lambda calculus.

7. Related Work

Integrating databases with programming languages has been a longstanding and hard issue. Many database programming languages have been proposed between the late 1970s and the early 1990s. The work on Machiavelli [24] was a typical example that used functional programming language over relations. Driven by the concept of the semantic web [16], many descriptive languages geared toward the management of the web-related data have been currently proposed. The examples are OWL (Web Ontology Language), a language taking ontology as the underneath data structure, RuleML (Rule Marked Language), a family of Web rule language using XML as the underneath data structure ([7] and [23]), the Linked Data, a language to publish and to connect data available on the web by taking RDF as the underneath data structure ([6], [21], and [41]), and XQuery, a language taking XML as the underneath data structure [8]. Obviously each approach has its unique features in terms of methodologies and the scopes of their applications. All the approaches are based on the traditional data models, i.e., the relational data model, the hierarchical data model, or data structures, such as graph-oriented data structures.

The work in searching for a better data model had been started as soon as we realized the limitations of the relational and the hierarchical data models in the early 1970s. The most active work in this area have focused on graph-oriented data structures, such as CODASYL [42], ER (Entity-Relational) [12], DAPLEX[28], and semi-structured data [10]. The graph-based approaches attempt to be more universal, in the sense of expressive power, and at the same time to have set-oriented operations similar to those in the traditional data models.

8. Summary

Application software started with a monolith where a programming language was the only component in the 1960s. To achieve a better productivity and to adapt to a rapid change of business requirements, a typical database application today consists of multiple components including database management system, programming language, web server, data exchange server, and access control server. With the EP data model that is semantically equivalent to a class of total recursive function, a monolithic architecture becomes available again for database applications. The new monolith is not a physical combination of traditional multiple components, but a logical consolidation of functions out of the traditional multiple components.

In this paper, we discussed the syntax, reductions, semantics, and computability of Froglingo. By going through the areas common to all languages, we gave Froglingo a precise definition, that identifies the features common to other languages, and the features that make Froglingo a system integrating databases with programming languages.

Acknowledgments

The author thanks Harold Boley for sharing his thoughts related to the paper and Mike Hicks for his encouragement and advice in developing this paper. The author specially thanks Olivier Danvy who spent extensive time in sharing his knowledge in programming languages and the lambda calculus. Without his help, many parts of the paper would not be complete.

References

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

[2] F. Afrati, S. Cosmadakis, and M. Yannakakis. "*On Datalog vs. Polynomial Time*". In Proc. ACM Symp.on Principles of Database Systems, Page 13-25, 1991.

[3] A. V. Aho and J. D. Ullman. "*Universality of Data Retrieval Languages*". Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Taxes, January, 1979, page 110 – 120.

[4] 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 Inference: Algorithmic Learning Theory. 1994

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

[6] C. Bizer, T. Heath, T. Berners-Lee. "Linked Data – the story so far". Journal on Semantic Web and Information Systems, 2009.

[7] H. Boley, S. Tabet, G. Wagner. "Design Rationale of RuleML: A Markup Language for Semantic Web Rules". Proc. SWWS'01, Stanford, July/August 2001.

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

[9] P. Buneman. "Functional Data Models – Position Paper", Workshop of Functional data model 1997.

[10] P. Buneman, M Fernandez, D. Suciu. "*UnQL, A Query Language and Algebra for Semistgructured Data Base on Structural Recursion*", VLDB Journal: Very Large Database, Volume 9, Number 1, page 76 – 110, 2000.

[11] L. Cardelli, P. Wegner. "*On Understanding Types, Data Abstraction, and Polymorphism*". Computing Surveys, Vol 17 n. 4, page 471-522, December 1985.

[12] P. Chen. "*The Entity-Relationship Management Model – Toward a Unified View of Data*". ACM Transactions on Database Systems. Vol. 1, No. 1, March 1976, Pg 9 – 36.

[13] J. C. Cleaveland. "*An Introduction to Data Types*". Addison-Wesley Publishing Company, 1986.

[14] A. J. T. Davie. "*An Introduction to Functional Programming Systems Using Haskell*". Cambridge University Press, 1992.

[15] R. Elmasri, and S. B. Navathe. "*Fundamentals of Database Systems, Second Editions*". The Benjamin/Cummings Publishing Company, Inc., 1994.

[16] L. Feigenbaum, I. Herman, T. Hongsermeier, E. Neumann, S. Stephens. "The Semantic Web in Action". Scientific American, 297(6), pp. 90-97, (December 2007)

[17] M. Goldberg. "The Lambda Calculus Outline of Lectures". Mary

31, 2011, available at: http://little-lisper.org/website/tutorials.html

[18] M. Gyssens, J. Paredaens, J. V. Bussche, and D. V. Gucht. "*A Graph-Oriented Object Database Model*". IEEE Transactions on Knowledge and Data Engineering. Vol. 6, No. 4. August 1994, page 572 - 586.

[19] M. Hammer, D. McLeod. "*Database Description with SDM: A Semantic Database Model*". ACM Transactions on Database Systems. Vol. 6, No. 3, September 1981, page 351 – 386.

[20] K. J. Hammond. "*CHEF: A model of case-based planning*." AAAI Proceedings of the 5th National Conference on Artificial Intelligence, page 267-271. Morgan Kaufmann, 1986.

[21] B. Heitmann and C. Hayes. "Enabling Case-Based Reasoning on the Web of Data". Workshop Proceedings of the Eighteenth International Conference on Case-Based Reasoning (ICCBR 2010), pp. 133 – 140

[22] D. Leake. "Case-*Based Reasoning: Experience, Lessons, and Future Directions*". Menlo Park: AAAI Press/MIT Press, 1996.

[23] J. K. Lee, M. M. Sohn. "The eXtensible Rule Markup Language". Communication of the ACM, Volume 46, Issue 5, pp. 59-64, May 2003

[24] A. Ohori, P. Buneman, V. Breazu-Tannen. "Database Programming in Machiavelli – a polymorphic language with static type inference". In ACM SIGMOD, 1989, pp. 46 – 57.

[25] S. Payton-Jones, D. Lester. "Implementation Functional Language Tutorial", March 23, 2000. Available at: http://research.microsoft.com/en-us/um/people/simonpj/Papers/pj-lester-book

[26] A. M. Pitts. "*Nominal System T*". POPL'10, January 17-23, 2010, Madrid, Spain.

[27] P. Revesz. "*Introduction to Constraint Databases*". 2002 Springer-Verlag New York, Inc.

[28] D. W. Shipman. "*The Functional Data Model and the Data Language DAPLEX*". ACM Transactions on Database Systems, Vol. 6, No. 1, March 1981, page 140 – 173.

[29] D. Steedman. "*X.500 - The Directory Standard and its Application*". Technology Appraisals, 1993.

[30] D. Turner. "Total Functional Programming". Journal of Universal Computer Science, vol. 10, no. 7 (2004), page 751-768.

[31] K. H. Xu. J. Zhang. S. Gao. "Froglingo, a Programming Language empowered by a Total-Recursive-Equivalent Data Model". To appear in Journal of Digital Information Management, 2011.

[32] 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.

[33] K. H. Xu, J. Zhang, S. Gao. "An *Assessment on the Easiness of Computer Languages*". The Journal of Information Technology Review, May, 2010, page 67 - 71.

[34] K. H. Xu, J. Zhang, S. Gao. "*Higher-order Functions and their Ordering Relations*". The Fifth International Conference on Digital Information Management, 2010.

[35] K. H. Xu, J. Zhang, S. Gao, R. R. McKeown. "*Let a Data Model be a Class of Total Recursive Functions*". The International Conference on Theoretical and Mathematical Foundations of Computer Science (TMFCS-10), 2010, page 15 – 22.

[36] K. H. Xu, J. Zhang, S. Gao. "*Froglingo, A Monolithic Alternative to DBMS, Programming Language, Web Server, and File System*". The Fifth International Conference on Evaluation of Novel Approaches to Software Engineering, 2010.

[37] 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.

[38] 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.

[39] 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.

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

[41] "Resource Description Framework (RDF): Concepts and Abstract Syntax". W3C Working Draft 08 November 2002, available at: http://www.w3.org/TR/2002/WD-rdf-concepts-20021108.

[42] Report of the CODASYL Data Base Task Group, ACM, April 1971.