Froglingo, a Programming Language empowered by a Total-Recursive-Equivalent Data Model

Kevin Xu, Jingsong Zhang, Shelby Gao
Bigravity Business Software
2306 Johnson Circle, Bridgewater
New Jersey, U.S.A
{kevin, jingsong, shelby}@froglingo.com

Journal of Digital Information Management

ABSTRACT: *The EP (Enterprise-Participant) data model is a language homomorphic to and semantically equivalent to a class of total recursive functions. It takes a unique position in the fields of programming languages and database management. Mathematically, it enables a programming language to achieve the greatest possible ease of use in software development and maintenance. (This assumes that a programming language incorporated with a higher expressive data model was easier than another one with a lower expressive data model, or no data model at all.) Practically, Froglingo -- a programming language incorporated with the EP data model is a monolith that consolidates multiple software components of traditional software architecture. In addition, Froglingo is untyped, i.e., programmers write application programs without a necessity of user-defined types. Finally, Froglingo is a novel approach to many challenges facing traditional technologies, including feature scalability, user interface flexibility, and similarity. In this paper, we show with further clarity the concept of ease of use by proposing a mathematical definition for the concept of data models and by relating Froglingo with other programming language through the analysis of types and higher-order functions.*

Categories and Subject Descriptors
D.3.2 [Language Classifications]; Data-flow languages: **D.3.3 [Language Constructs and Features];** Data types and structures; **H.2 [Database Management]:** Data models

General Terms: Programming languages, Froglingo, Data models types, Computability, Algebra, Homomorphism

Keywords: Froglingo programming langauge, EP data model, Higher-order functions, Total-recursive equivalence, Knowledge management

Received: 11 February 2011, Revised 31 March 2011, Accepted 9 April 2011

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 data models, i.e., the mathematical underpinnings of DBMSs, complement programming languages with three essential characteristics: 1) a data model offers set-oriented operations to update and query data, 2) each operation always terminates, and 3) like a data type, a data model is a language homeomorphic to an algebra, i.e., programmers specify what a program is in terms of business requirements, but don't necessarily specify how it will be implemented.

The traditional data models, i.e., the relational data model and the hierarchical data model, 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 (abbreviated as a total-recursive-equivalent data model in this paper) [25]. 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.

The EP data model is a data model because it possesses the three above-mentioned essential characteristics. To demonstrate the significance of the EP data model being both a data model and a language semantically equivalent to a class of total recursive functions, the authors of the articles [27 and 23] suggested an objective view on easiness. This view contends that: 1) a data model is easier to use than a programming language in the development and maintenance of those applications expressible in the data model, 2) if one data model is more expressive than another data model, the former is easier than the latter in the development and maintenance of the applications where a programming language is involved, and 3) a programming language, by incorporating with a total-recursive-equivalent data model, is the easiest to use in software development and maintenance. In these articles, Froglingo, incorporated with the EP data model with an implementation [28], was therefore determined to be the easiest programming language. Calling Froglingo the easiest programming language to use has the following practical implications: 1) Froglingo is a monolith that consolidates multiple software components of traditional software architecture [26], 2) The EP data model is a consistent tool to manage as much finite data as a business application needs [25 and 27], and 3) The EP data model arranges business data in a manner such that a rich set of built-in operators are available to use [24].

In this paper we press the notion of easiness further with the following contributions:

1. Taking Froglingo as a complete programming language, we examine how it adopts procedural statements of imperative programming languages in conjunction with the EP data model to specify business logic, and provides built-in facilities to interact effectively with users and to communicate with other applications.
2. Through an analysis of an application system implemented in Froglingo (an award recipient in the ICCBR 2010 Computer Cooking Contest), we observe that Froglingo, as a untyped system, is a novel approach to many challenging issues facing traditional technologies, including feature scalability, user interface flexibility, and similarity.
3. We propose a mathematical definition to formalize the concept of data models. Through the definition, we show that we can clearly see the differences between programming (or other specialized) languages and data models, and the differences among data models themselves.
4. We review typed and untyped systems, and conclude that Froglingo is untyped. Unlike the other untyped systems, Froglingo is the safest tool to use. Unlike all the other systems, Froglingo fully utilizes the properties of the class of total recursive functions that enable it to be a novel methodology approaching many challenging issues facing traditional technologies.

The last 2 contributions above show with further clarity the concept of ease of use.

In Section 2, we introduce the EP data model. In Section 3, we describe the features of Froglingo beyond the EP data model. In Section 4, we analyze the recipe advisor in Froglingo that participated in the ICCBR 2010 Computer Cooking Contest. In Section 5, we give the related work, including the proposal for a mathematical definition of the concept of data models and a discussion on types.

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 are restrictions, and they 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 [29], one such 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 Terms and Databases

The EP data model will be described as a formal language. We will discuss terms, assignments, and databases in this sub section, and normal forms and reductions in Section 2.2. In section 2.3, we will introduce the set of operators stemming from the ordering relations among managed data.

Definition 2.1
Let P be a set of identifiers, and C a set of constants where `null` is a special constant. The set of terms T is formed by the following rules:

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

$$m \in T, n \in T \Rightarrow (m\ n) \in T$$

For example, the expressions *3.14, "a string", an_id*, $(f\ 1)$, $((country\ state)\ county)$, and $((a\ b)\ (c\ d))$ are terms. The entire set T can be called the Herbrand universe of the EP data model.

Given $m, n, q \in T$, we introduce the following notations:

Notation 2.2
1. Terms m and n are called sub-terms of the application $(m\ n)$. A term is also a sub-term of itself.
2. If q is a sub-term of m or n, then q is also a sub-term of the application $(m\ n)$.
3. Term m is called the left sub-term and a leftmost sub-term of the application $(m\ n)$, and n the right sub-term and a rightmost sub-term of the application $(m\ n)$.
4. If q is a leftmost sub-term of m, then q is also a leftmost sub-term of the application $(m\ n)$. Similarly, if q is a rightmost sub-term of n, then q is also a rightmost sub-term of the application $(m\ n)$.
5. The parentheses surrounding an application can be omitted when the right sub-term is not another application.

   For

   Example, $(f\ 3)$, $((country\ state)\ county)$, and $((a\ b)\ (c\ d))$ can be rewritten the following way: *f 3, country state county*, and $a\ b\ (c\ d)$ correspondingly.
6. Given an expression $m \equiv n$, the symbol $\equiv$ indicates that the two symbols *m* and *n* are identical.

Many notations from the lambda calculus have been adopted in the EP data model. Unlike the lambda calculus, however, the EP database has identifiers and doesn't have variables. (Froglingo does have variables as discussed in Section 3.)

A term can be assigned with another term.

Definition 2.3
Given $m, n \in T$, the form $m = n$ is an assignment. Here, m is called the assignee and n the assigner. All the assignments in a given T make up a set: $A = \{\ m = n \mid m \in T, n \in T\ \}$.

Now we are ready to introduce the definition of an EP database:

Definition 2.4
An EP database D is the union of a set of terms $T \subset T$ and a set of assignments $A \subset A$, i.e., $D = T \cup A$, such that the following things are true:

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

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

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

$$(m = n) \in D \Rightarrow \forall t \in T, m\ t \notin D$$

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

The above restrictions force users to enter those and only those business data that are semantically equivalent to a class of total recursive functions [25].

The following example is an EP database for the "Social Security Department" in the United States and for a central administration office in a college. In the "Social Security Department" (SSD), each resident has a social security number (SSN), a name, and a birth date. In the college, a resident registers as a student, the college has departments, each department offers classes, and each class has students.

Example 2.5
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";

Note that *college.edu* and *SSD.gov* are also identifiers, and an assigner can be an image file.

According to Definition 2.4, the subterms in the assignee and assigner of a database, e.g., *SSD.gov, John,* and *SSD.gov John* must be in the database too. We don't show them here because it is clear and implied by the following propositions:

Proposition 2.6
1. If an application is in a database, so are its left sub-term and its right sub-term, i.e.
$$m\ n \in D \Rightarrow m \in D \cap n \in D$$
2. If an assignment is in a database, so are its assignee and assigner, i.e.
$$m = n \in D \Rightarrow m \in D \cap n \in D$$

With the EP data model, one may express directed graphs with circles. Here is an example:

Example 2.7
A directed graph with a cycle:

v1 v2 = v2;
v2 v1 = v1;
v2 v3 = v3;

2.2 Normal Forms and Reduction
Given a database, each term in the Herbrand universe T can be reduced to a normal form.

Definition 2.8
Given a database D, the set of normal forms NF is defined as follows:

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

For example, terms *"F", SSD.gov*, and *SSD.gov John* are normal forms, but not *SSD.gov John birth* in Example 2.5.

Definition 2.9
Given a database D, we have the one-step evaluation rules, denoted as →

1. An identifier not in D is reduced to `null`, i.e.,
$$p \in P \cap p \notin D \Rightarrow p \rightarrow \texttt{null}$$
2. An assignee in D is reduced to its assigner, i.e.,
$$(m = n) \in D \Rightarrow m \rightarrow n$$
3. If $m, n \in NF$, and $m\ n \notin D$, then $m\ n$ is reduced to `null`, i.e.,
$$m, n \in NF, m\ n \notin D \Rightarrow m\ n \rightarrow \texttt{null}$$
4. The application of two terms is reduced to the application of their normal forms, i.e.,
$$m, n \in T, m \rightarrow m', n \rightarrow n' \Rightarrow m\ n \rightarrow m'\ n'.$$

Definition 2.10
Let $m, n \in T$, and D a database. If there is a finite sequence $l_0, ..., l_q \in T$, where $q \geq 0$, such that $m \equiv l_0, l_0 \rightarrow l_1, ..., l_{q-1} \rightarrow l_q, l_q \equiv n$, then

1. m is effectively, i.e., in finite steps, reduced to n, written as $m \rightarrow_{EP} n$.
2. If $m_1 \rightarrow_{EP} n$ and $m_2 \rightarrow_{EP} n$, 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 2.11
Below are a few equations from the databases in Examples 2.5 and 2.7:

SSD.gov John SSN == 123456789;
(College.edu Admin (SSD.gov John) Major) == College.edu CS;
v1 v2 v1 == v1;
v1 v2 v1 v2 v1 == v1 v2 v1;

With the concepts defined so far, the EP data model was proved as a consistent, sound, and complete language that takes a class of total recursive functions as its semantics [25]. Therefore, each term in T can be effectively reduced to one and only one normal form with a given database. Intuitively, an EP database is interpreted as a set of higher-order functions, and the entire class of higher-order and total recursive functions can be represented by an EP database provided that the space for a database was unlimited. The proof in [25] further showed that each term or assignee in an EP database has a corresponding higher-order function. Mathematically, this correspondence is called a homomorphism. This will be further discussed in Section 5.1.

2.3 Ordering Relations
There is a rich set of ordering relations among higher-order functions, and therefore also among the terms in a database. Before introducing the individual ordering relations, we provide in Figure 1 below an alternative presentation of the database in Example 2.5.

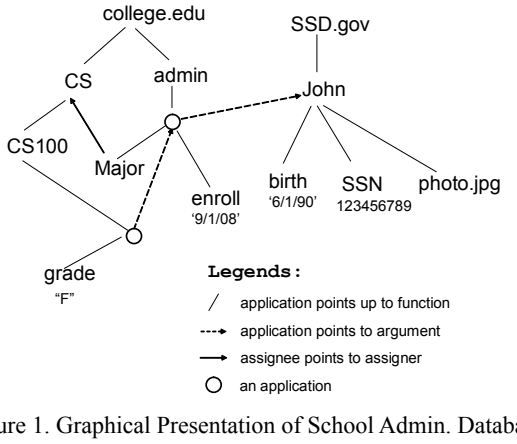

Figure 1. Graphical Presentation of School Admin. Database

In Figure 1 above, we started to use the words "function" and "argument" interchangeably in place of EP terms. We will see in Section 5 that EP terms are homomorphic to higher-order functions.

An application depends both on its function and on its argument as well as the sub-terms of the function and the argument. This leads to the development of the following relations:

Definition 2.12
1. Given a term $(m\ n) \in T$, we define the relation "has the left sub-term", denoted as $m\ n\ \{+\ m$; and the relation "has the right sub-term", denoted as $m\ n\ \{-\ n$, i.e.,

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

2. Given a term m in a database D, let l, s, r are a leftmost sub-term, a sub-term, and a rightmost sub-term of m accordingly, then the operators {=+, {=-, and {= in the following expressions are defined such that the expressions are evaluated to be true: $m\ \{=+\ l$, $m\ \{=-\ r$, $m\ \{=\ s$. In other words:

$\{=+ = \{<m, l> \mid m \in D$ and l is a leftmost sub-term of $m\}$

$\{=- = \{<m, r> \mid m \in D$ and r is a rightmost sub-term of $m\}$

$\{= = \{<m, s> \mid m \in D$ and s is a sub-term of $m\}$

Here are the sample expressions with the value of true:

SSD.gov John birth {+ SSD.gov John;
SSD.gov John birth {=+ SSD.gov;
SSD.gov John birth {=- birth;
College.edu CS CS100 (College admin (SSD.gov John)) {=- John;
birth {= SSD.gov John birth;
John {= SSD.gov John birth.

Because it exists independently, the normal form, resulting from an application, doesn't have to depend on the function and the argument of the application. However, a normal form is derivable from an application; therefore, it is derivable from the function, from the argument, and from the sub-terms of the function and the argument. This leads to the development of a few pre-ordering relations.

Definition 2.13
1. Let $m, n, q \in T$ and D a database, if $m\ n == q$, then the operators (+ and (- in the following expressions are defined such that the expressions q (+ m and q (- n are evaluated to be true. In other words,

$$(+ = \{<q, m> \mid \forall m \in T, \exists n, q \in T \text{ such that } m\ n == q \text{ under } D\}$$

$$(- = \{<q, n> \mid \forall n \in T, \exists m, q \in T \text{ such that } m\ n == q \text{ under } D\}$$

2. Let $m, q, l, s, r \in T$, D a database, $m == q$, l is a leftmost sub-term of m, s is a sub-term of m, and r is a rightmost sub-term of m. Then the operators (=+,(=-, and (= in the following expressions are defined such that the expressions are evaluated to be true: q (=+ l, q (=- r , and q (= s. In the other words:

$(=+ = \{<q, l> \mid \forall l \in T, \exists m, q \in T$ such that
l is a leftmost sub-term of m, and $m == q$ under $D\}$

$(=- = \{<q, r> \mid \forall r \in T, \exists m, q \in T$ such that
r is a rightmost sub-term of m, and $m == q$ under $D\}$

$(= = \{<q, s> \mid \forall s \in T, \exists m, q \in T$ such that
s is a sub-term of m, and $m == q$ under $D\}$

Note that the relations were defined over the entire Herbrand universe. The implementation of Froglingo, however, only considers database D as the domain of the relations.

Example 2.14
Given the databases in Example 2.5 and 2.7, here are a few Boolean expressions with true values:

"F" (+ College CS CS100 (College admin (SSD.gov John));
"F" (=+ College CS CS100;
"F" (=- SSD.gov John;
"F" (= College.edu admin;
v2 (=+ v1; v1 (=+ v2;
v1 (= v2;

The pre-ordering relations appear irrelevant to the queries supported in traditional database technologies. However, they were found useful in the following examples.

Example 2.15
Given the data presentation in Example 2.7 for a directed graph,

1. The query "if there is a path from *v1* to *v3*" is expressed as: *v3 (=+ v1*. It is evaluated to be true since *v3 == v2 v3 == (v1 v2) v3*.
2. The query "if there is a circle between *v1* and *v2*" is expressed as: *v1 (=+ v2* and *v2 (=+ v1*. It is evaluated to be true because *v1 == v2 v1* and *v2 == v1 v2*.
3. The query "find all the vertexes that has a path from *v1*" is expressed as: *select $v where $v (=+ v1*. It is evaluated to be *v1, v2*, and *v3*. Here the identifier *select...where...* is a built-in facility structure for a set-oriented operation.

3. Froglingo

With the EP data model that is equivalent to a class of total recursive functions, we minimize our dependency on a programming language. But a programming language 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 2.3 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 2.3, there are still an infinitely many total recursive functions that are potentially demanded by applications. (The notion of the equivalence of the EP data model to a class of total recursive functions implies that all the total recursive functions could be expressed in the EP data model provided that there were infinite time and space [25]. For practicality, however, the EP data model always expresses finite data. Therefore, a programming language is still needed to express infinite data with finite expressions. The practicality of the EP data model is comparable to the practicality of a programming language that theoretically expresses a class of partial recursive functions with the hypothesis of infinite time and space, but practically expresses a finite set of partial recursive functions.)

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 [19].

In this section, we introduce the rest of the concepts beyond the EP data model of Froglingo.

3.1 Variables

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 3.1

1. Let V be a set of variables. A variable is a term, i.e., $v \in V \Rightarrow v \in T$.
2. If a variable is in the assigner of an assignment in a database, it must be in the assignee.
3. If a variable is in a term in a database, it cannot be the left-term of a sub-term in the given term.

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

Example 3.2

```
fac 0 = 1;
fac $n = ($n * (fac ($n - 1)));
fun $x 1 $y = ($x + $y);
fun $x 2 $y = ($x * $y);
```

Below are a few query expressions and reduction results:

```
fac 4 → 24;
fun 3 2 4 → 12;
```

Note that the reduction rules in Definition 2.9 are enhanced with variables and not discussed here.

Semantically, the expressions above are equivalent to a database having infinite assignments: *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).

A variable can be restricted within a domain 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 3.3

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

The data represented by the terms containing variables obeys the ordering relations; therefore, the corresponding built-in operators are applicable to variables [24].

3.2 Sequential Terms

The EP data model and variables are mathematically sufficient to make Froglingo semantically equivalent to a class of partial recursive functions. To express multiple actions triggered by single events, and to perform user input checking, the procedural statements in imperative programming languages are adopted, and called sequential terms.

A sequential term is a sequence of terms separated by commas '*,*'. Sequential terms can only serve as assigners.

Definition 3.4

1. The set of sequential terms S is formed by the following rules: $t \in T \Rightarrow t \in S$; $t \in T \wedge s \in S \Rightarrow s$ ',' $t \in S$.
2. An assignment in a database is extended to allow its assigner to be a sequential term.

A money transfer between two bank accounts is expressed as follows:

Example 3.5

```
transfer $money =
        (update account2 = (account2 - $money)),
        (update account1 = (account1 + $money));
```

In the example above, we assumed two pre-defined accounts, such as *account1 = 100* and *account2 = 300*. In addition, the identifier *update* is a built-in operator that updates assignees' assigners. The entire expression associated with an update operation is also called a term. There are other built-in operators, *create* and *delete*, that are not introduced but are assumed here. Note that sequential terms, update operations, and set-oriented operations like the one in Example 2.15.3 are implemented to return values too. The returned values are a set of pre-defined constants [28].

The administrator of the college *college.edu*, as another example, can construct a function *register* that accepts class registration requests from students. Given a requester and a course as the inputs of a request, a sample constraint is that the request is accepted only if the requester, through her "signature" (a built-in term to be converted to her own user account), hasn't yet registered for the course:

Example 3.6

```
register $usr:[$usr isa signature] $class =
    register_validate
        ($class (college.edu admin $usr) == null)
        $usr
        $class;
```

The term *SSD.gov john* is a sample instance of the variable *$usr*. The term *college.edu CS CS210*, if *CS210* is another class offered by the college, is a sample instance of the variable *$class*. The Boolean expression with the operator == is to test if the requester has already registered the class. The function *register_validate* is further defined as

Example 3.7

```
register_validate false $usr $class =
        "The registration was not successful.
        You must have registered the class already.";
register_validate true $usr $class =
        "You have successfully registered the class",
        (create $class (college.edu admin $usr));
```

3.3 Built-in facilities

An application program in a traditional programming language, where a relational DBMS is used, needs to have application-dependent data access control (also called user entitlement) against the relational data. This is necessary because the data access control, in the correspondence of total recursive functions, cannot be expressed by the relational data model, but only by the programming language. This is not an issue in Froglingo with the EP data model. Froglingo has a built-in facility

addusr to create user accounts. A user account itself is a term, and it is more than a term. A user cannot access a database unless he/she is logged into a user account by providing a password. For example, the term *college.edu, SSD.gov,* and *SSD.gov John* can be added to the database in Example .2.5 as user accounts.

An EP database can be viewed as a hierarchical structure under the relation {+; therefore, the data in an EP database is divided into private spaces for individual users according to {+ as if it was a file system. The owner of a user account, e.g., a term *h*, is the administrator for the entire space under the user account *h*, i.e., all the assignees (terms) *t* such that *t* {+ *h*.

Analogous to the concept of paths in a file system, the EP term for an assignee in an EP database serves as the name of the assignee; therefore, the concept of a current working directory in a file system is adopted to allow users to navigate through data with the guide of the ordering relation {+. Consequently, an assignee can be named differently, depending on where a user stands. In the school administration database in Example 2.5, for example, the assignee *SSD.gov John* has actually a distinguished name *//SSD.gov John*. When a person logs into it as a user account, the system will prompt the distinguished name in the coming command line: [*//SSD.gov John*].

This means that the system takes the user account as the home working stand by default, and the assignee *SSD.gov John birth,* having the distinguished name *//SSD.gov John birth,* will have a relative name *birth*.

The owner of a user account can freely create new data, remove, or change data under the user account. The owner has the freedom to construct as many functions as he/she wants, as if a user had the freedom to manage as many files as he/she wanted.

Users share their data by granting an access privilege, the only privilege after the administrator privilege assigned to an owner originally. Assuming that the function *register* in Example 3.6 is created under the user account *college.edu,* then the owner of the account can issue the following command:

[*//college.edu*] *grtacc register anyone;*

With the built-in operator *grtacc*, it grants the user account *anyone* (a built-in account in Froglingo for any user) the access privilege to the assignee with the distinguished name *//college.edu register.* Note that the access privilege doesn't necessarily mean a read-only permission. Instead, the user account that is granted the access privilege to a function impersonates the owner of the function in executing the application of the function to a user provided the argument. Therefore, a call to the function *register* from any student may cause the system to create new data in database.

The system interaction with users in Froglingo is extended to use web browsers across a network. For example, a user can call the function *register* through a HTTP request message via a web browser:

http://college.edu/register signature (*%2F%2Fcollege.edu CS cs201*)

Here the entire string is a URI embedding the previous command. In the URI, "%2F" is the hex code for the character '/' required by the URI syntax.

Note that web pages, i.e., HTML documents, possibly embedding Froglingo expressions, are stored as data in EPdatabase..

4. Case Study

Data structures in traditional programming languages help to organize business data and help to detect errors. The classes and subclasses in object-oriented programming languages further improve the productivity of software development by code reuse through the concept of inheritance. Instead of types, the EP data model is untyped, i.e., every thing is modeled as higher-order functions without a user-defined type. See Section 5.2 for more discussion about types.

In this section, we introduce a food recipe advisor written in Froglingo, which was the aforementioned award recipient in the ICCBR 2010 Computer Cooking Contest [22]. Through the recipe advisor, Froglingo demonstrated the following characteristics: 1) a high feature scalability, i.e., easy to adapt to new business requirements, 2) a search ability based on both key words and phrases against ordered data in the database, 3) a flexible user interface indiscriminately accepting diverse application objects, 4) a consistent way of preserving similarities.

4.1 Knowledge Representation

User-defined data types, once established in traditional technologies, become the infrastructures of application systems, and are difficult to change when it is necessary to adapt to new business requirements. It becomes more evident in emerging application areas such as knowledge management and artificial intelligence where initial understandings of a problem are often imperfect and refined knowledge may render existing understanding obsolete [16]. In the CBR (Case-Based Reasoning) domain of cooking [15], for example, we may simply represent cooking knowledge as a list of ingredients. But this is just the beginning of cooking knowledge acquisition process. Beyond ingredients, we may have to consider cooking equipment, ingredient volume, food preparation steps and sub steps, and heating factors as they relate to the changes in flavor, texture, aroma, color, and nutritional content.

In Froglingo, every object is represented in higher order function regardless of whether it is a data type definition, a simple object, or a complex object. We demonstrate in this subsection that the uniformed presentation of data in the EP data model helps application upgrades and new feature enhancement (and therefore increases feature scalability).

An ingredient may have an ingredient type, and an ingredient type may have its parent type. Sometimes, an ingredient may belong to multiple types. Here are a few examples in EP terms:

meat beef (*short loin*)*;*
meat chicken;
vegetable artichoke;
broth (*meat chicken*)*;*
broth vegetable;

The preparation method is also important in cooking. The methods *bake*, *grill*, *steam*, and *stir fry* are typical. Preparing a dish normally involves multiple methods and therefore a sequence of preparation steps, and each step may have its sub-steps. All of these can be represented in Froglingo. But just for demonstration, only one preparation method is considered for a recipe in the case study.

Many ingredients and dishes have different origins. For example, curry is an Indian ingredient, a wok is an East Asian cooking utensil, and a Fajita beef is a Mexican dish. To support this sort of diversity, we inventory a list of food origins. Here are a few examples:

Asian Chinese Szechuan;

French Dijon;

Irish Kilkenny;

Ingredients may have multiple names. Here are a few samples:

lichee = lichi;

chile = hot pepper;

Now let's talk about recipes. A recipe is represented by an assignment in Froglingo. Here are a few examples:

(*flour* (*all purpose*)) *sugar yeast salt milk egg bake*
= world class waffle;

(*flour* (*all purpose*)) *sugar yeast salt shortening bake*
= batter white bread;

(*meat chicken*) *cheese seasoning* (*sauce salsa*) *bake Italian*
= spicy parmesan chicken;

In the first example, the recipe "World Class Waffle" is a baked food and has the ingredients in the order that appears in the original recipe book: all-purpose flour, sugar, yeast, salt, milk, and egg. We simply take the sequence and place them in the order as an EP term. If we know its cooking method and (or) its origin, we just simply append them to the previous EP term.

Because the first two recipes above have the same ingredient sequence at the beginning: all-purpose flour, sugar, yeast, and salt, the sequence is stored only once, and therefore is called a shared ingredient sequence. The shared ingredient sequence is a factor used to determine if two recipes are similar. (This similarity is to be discussed further in Section 4.3.)

All the information is represented so far in a consistent manner, i.e., in higher-order functions. See Figure 2 for a graphical view. The consistency helps to add new functionality to the top of existing data with minimum effort. Here is an example that represents a dinner consisting of an appetizer, a main dish, and a dessert, which would normally be represented by different classes in object-oriented programming language:

dinner (salad potato) (spicy parmesan chicken) (cake lemon);

To express a set of dinners with alternative starters, main dishes, and desserts, at a restaurant owner's discretion, it may be convenient to use variables instead of enumerating the permutations. The following sample expression is equivalent to a set of dinners including the one defined earlier:

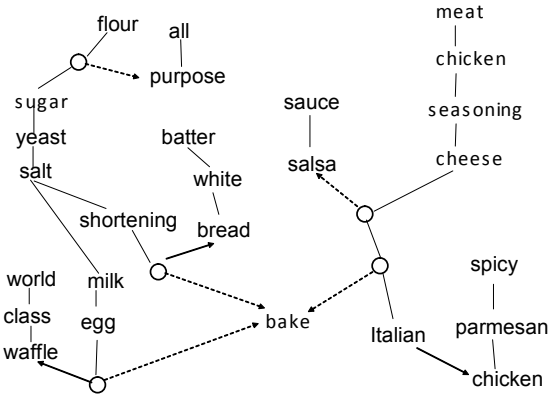

Figure 2. Graphical Presentation of Cooking Knowledge Database

dinner $s:[*$s* {=+ *salad or $s* {=+ *soup*]

$m: [*$m* {=+ *chicken or $m* {=+ *marinate*]

$d: [*$m* {=+ *cake*];

Despite the variables, the above expression preserves the ordering relations discussed in Section 2.3.

4.2 User Interface and Queries

Categorizing and representing business data in different data types may require end users to enter query inputs in different entry fields through a graphical user interface (GUI). For example, a customer may want to find all the Italian foods with baked chicken. In such a case, a recipe system in traditional technologies may provide a GUI with three entry fields corresponding to the three internal data types: ingredients including chicken, origins including Italian, and cooking methods including bake. The separation at user interface level may not be desired in many circumstances, especially when end users have limited knowledge about business domains.

Froglingo supports a user interface quite differently. In the sample screenshot of Figure 3, the interface allows users to enter a list of phrases or words delimited by a comma, and each phrase or word can be preceded with the word "no" for negation. Phrases and words for different types of objects can be entered in a single field without regard to any particular order.

Comparable to the search ability of traditional technologies based on key words against textual documents, Froglingo supports searches based on both key words and phrases against a structured database. A phrase, e.g., "all-purpose flour" from the user interface discussed earlier, is initially translated into a combinatory term in Froglingo, e.g., *flour* (*all purpose*). This translation process is the same one used for knowledge collection discussed in Section 4.1 (and it is automated via a parser).

Figure 3. A user interface indiscriminately taking diverse inputs

This interface is primarily supported by a single built-in operator {= internally. The following expression, for example, can be automatically converted from the query string in Figure 3:

select $dish where $dish {= *flour* (*all purpose*) *and*

$dish {= *bake and $dish* {= *Italian and*

$dish {= *vegetable and not $dish* {= *sauce tomato;*

The expression above will retrieve all the terms (assignees) in a database that have sub terms *flour* (*all-purpose*), *bake*, *Italian*, *vegetable* (and therefore *vegetable artichoke*), but not *sauce tomato*.

The same interface is equally applicable to dinners, more complex objects than recipes. When customers enter a string:

"*dinner, no soup, chicken, lemon cake*", for example, the system, having the function *dinner* defined in Section 4.2, will retrieve all the "dinners" that are generated dynamically to have three components: a starter, a main dish, and a dessert. The system doesn't have the concept of dinner, but it has the term *dinner* from the key word "*dinner*" in the inquiry string. In each selected dinner, soup must not be any of the three components (therefore the starter must be a salad), chicken must be an ingredient of at least one component (since chicken is never a part of cake, it obviously cannot be a part of dessert), and lemon cake must be the dessert.

4.3 Similarity

In many business applications, we are required to compare similarities among business objects. For example, if a restaurant doesn't have an exact dish that the customer wants, he/she may want to order a similar dish. Ranking objects by weights, e.g., a calculated weight for a dish based on an aggregate function over the pre-assigned weights of its ingredients, is a common approach. This works for many applications. The weighted approach is a quick solution in traditional technologies when managed objects are not easily arranged in a way that allows them to be related to each other structurally. Taking the recipes that are represented by sequences of ingredients as an example, we are able to arrange ingredients in classes and subclasses through which one can tell their similarities. For example, salmon and flounder are two subclasses of the super-class fish, and therefore they are similar. But there is not an easy way in traditional technologies to relate two recipes so that their similarities can be revealed structurally, although individual ingredients may be indexed separately.

Recipes in Froglingo are stored structurally. Given the recipe database in Figure 2, for example, the three recipes are similar because all of them are related by *bake*; and the recipe "world class waffle" is similar to "batter white bread" because they share the same ingredient sequence: all-purpose flour, sugar, yeast, and salt. If the recipe "batter white bread" is desired but the ingredient shortening is not available, then the system would choose the recipe "world class waffle" as the substitute.

The structure, i.e., ordering relations among higher-order functions, offers a novel approach to preserve naturally the similarities among recipes. Since the EP data model is untyped, the structural view on similarity can be consistently applied to other business applications where revealing similarities are required.

5. Related Work

At the beginning of this paper, we said that decidability, set-oriented operations, and homomorphism were the three essential features of data models. In Section 5.1, we will use an algebraic approach to give a mathematical definition for data models. The purpose is to understand better the related work in data models and database management, and further to support the concept of ease of use proposed in [27 and 23].

In sections 5.2 and 5.3, we discuss the related work in programming languages.

5.1 Data Models

There is no commonly accepted definition for the concept of data models; however, it is certain that a data model is a data type. Integers, arrays, records, and linked lists are the common data types in programming languages. Viewing a data type as an algebra, i.e., a set of values and a set of operations on the values, in the context of semantics [8], one cannot ignore the fact that a computer language must exist to expresses the data type. Because of this, we promote another view that a data type is a language in which each syntactical expression of a data type has a correspondence in its semantics. Mathematically, this correspondence is called a homomorphism. Because of the homomorphism, the syntactical aspect of a data type is also an algebra. In the rest of this subsection we call the syntactical aspect of a data type the *syntactical* algebra, and the semantics the *semantic* algebra.

A clear motivation for studying data types by promoting homomorphism is to ease the development and maintenance of business applications because data types support better business data organization, error checking, and built-in operator utilizations [8]. Note that the concept of homomorphism is not applicable to a generic programming language because of errors, side-effects, and non-determinism [8]. In addition, local variables as a core constructor of an imperative programming language don't have immediate correspondences to the semantics, i.e., those outlined in the business requirements of business applications.

It is always desirable that a program written in a language terminates. Put differently, it is always desirable for each operation in semantic algebra to halt on arbitrary input. Because of this, equivalently, it is desirable for the homomorphism of a data type to be decidable, i.e., it can be determined effectively (in a finite number of steps) if an entity in the semantic algebra corresponds to a given element in the syntactical algebra. In the work [8], it equivalently says that each element in the semantic algebra is reachable from the syntactic algebra of a given data type. This is true for primitive types like integers and for aggregate types like arrays in generic programming languages. Considering the broad scope of the set of data types defined in [8], including functions, however, it is clear that not every data type has a decidable homomorphism.

To distinguish a data model from a data type, we say that a data model has decidable homomorphism.

Offering set-oriented operations is another unique feature of a data model. To satisfy this requirement, we further differentiate a data mode from a data type by saying that the semantic algebra of a data model includes at least one relation. In summary,

Definition 5.1.1

A data model is a language with a decidable homomorphism to an algebra including at least one relation.

Which systems are data models? First, the relational data model is a data model. The relational algebra is both the syntactical and semantic algebra of the relational data model. (The homomorphism becomes an isomorphism.). Referencing the relational algebra alone to define the relational data model is sufficient because alternative languages such as the algebra calculus can be syntactically converted to the relational algebra.

We can say formally here that the hierarchical data model is a data model. There are many hierarchical structure systems, such as file systems in operating systems, X.500 [21], and XML (Extensible Markup Language). Essentially these systems offer hierarchical structures. A hierarchical structure was clearly described in [12] as a parent-child relationship, denoted as *PCR*. To define the concept of the *PCR* relationships in the language set by Definition 5.1.1, we introduce an alternative definition:

Definition 5.1.2
A relation R over a set S is a parent-child relation (PCR) if, for each pair $<x, y>$ in R, the following conditions are satisfied:

1. There is not another pair $<z, y>$ in R, where x and z are distinguished elements in S.
2. If there is a chain of pairs $<x, e_1>, < e_1, e_2>, \ldots, < e_{n-1}, e_n>$ in R, then it is impossible for y to be identical to any one of the elements $x, e_1, e_2, \ldots, e_{n-1}$, and e_n in S.

The first condition says that a child entity can only have one parent. The second condition says that a child cannot be its ancestor. A hierarchical structure is a PCR relation. Equivalently, a hierarchical structure is a dependent relation:

Definition 5.1.3
Given a PCR relation over a set S, the dependent relation DEP is defined as:

1. For each x in S, $<x, x>$ is in DEP,
2. If $<x, y>$ is in PCR, then $<x, y>$ is in DEP, and
3. If $<x, y>$ and $<y, z>$ are in DEP, then $<x, z>$ is in DEP.

We can now simply define that the hierarchical data model as a hierarchical structure system homomorphic (or even isomorphic) to an algebra including a PCR relation or a DEP relation.

An alternative and intuitive semantic algebra for the hierarchical data model can be an algebra constructed purely on the inclusion operation of the set theory. Given a hierarchical structure: $C \rightarrow B$, $D \rightarrow B$, $B \rightarrow A$, $E \rightarrow A$, where A is the root, and each arrow "$\rightarrow$" is the link from a child to a parent, for example, there is an isomorphic structure: $\{o_1, o_2, o_3, \{ o_1, o_2\}, \{\{ o_1, o_2\}, o_3\}\}$.

The EP data model is a data model. The entire set of EP-terms defined in Definition 2.1 are the syntactical (word) algebra. A database defined in Definition 2.4 and the reduction rules defined in Definition 2.9 are the axioms. The normal forms are mapped to the elements in the applicative structure for the corresponding class of total recursive functions, and the EP data model exactly expresses a class of total recursive functions [25]. In addition, the EP data model has the built-in operators such as {+ which are in correspondence with a relation among the higher-order functions in the applicative structure.

The definition for the concept of data models emphasizes the aspects of both syntax and semantics because not all the operations are allowed to be in the algebras of data models. For example, the function: "find all the paths between A and B in a graph" cannot be a part of an algebra because this function doesn't terminate for a graph having a cycle including vertices A and B.

Given an algebra having its set of values closed on its operations, on the other hand, we cannot be certain that there is a language exactly expressing the algebra. We use Datalog to express transitive closure, e.g., the entire pairs of vertices in a (cyclic) graph such that each pair represents a path from one vertex to another. However, a transitive closure may not be completely reachable by Datalog. (The article [2] shows that the existence of a path whose length is a perfect square between two nodes is not expressible in Datalog.) This says that Datalog cannot serve as the syntactical aspect of a data model that takes transitive closure as the semantic algebra. Many of research interests in network-based (also called graph-based) structures, such as the work in [5, 20, 13, 7, and 14], seek a language as well that has a decidable homomorphism to transitive closure.

5.2 Types
Type (also called data type) is one of the most important terminologies in the field of programming languages. To highlight the differences of Froglingo from traditional programming languages, we claimed in Section 4 that Froglingo is untyped. The phrase "untyped" delivers mixed messages in the field. First, it promotes declarativeness. On the other hand, it is a tag of a naked machine with bit streams in memory [6], that is practically too vulnerable and too tedious to be accessed directly from developers, and thereafter that is protected by a typed programming language. As a matter of fact, the phrase "untyped" is more often connected with the lambda-calculus where self-application is allowed, i.e., applying a function to itself [4]. Self-application demonstrates a flexibility for developers, and at the same time a danger of a non-termination process.

In sections 5.2.1 and 5.2.2, we intuitively discuss what typed programming languages are and what untyped programming languages are, and we further distinguish the differences between those untyped systems for partial recursive functions and the untyped Froglingo that has a type equivalent to a class of total recursive functions. In Section 5.2.3, we revisit the concepts discussed earlier by proposing a precise definition for types.

5.2.1 Typed Systems
Being typed is an essential characteristics of traditional programming language in practice. A variable (or an object in an object-oriented programming language) is typed by assigning a type explicitly. A procedure (or a method in an object-oriented programming language) is typed by assigning types to its parameters and its return value. A type in a typed system can be a built-in data type such as integers and strings, or can be a user-defined data type. (Note that some programming languages support the procedures that take other procedures as values through parameters. In such a case, the entire set of procedures can be viewed as a type; still there is currently no programming language yet that allows a user-defined type with members of procedures.)

The first role of types in a programming language is error-checking. For example, a compiler reports an error after detecting an attempt to add a string with an integer, and an application system rejects an update operation if there is an attempt to add a student record into a set of employee records. Error-checking through types helps developers in debugging and maintains data integrity.

Secondly, types play a role of abstraction, i.e., segregating their lower-level implementation details from their signatures (i.e., abbreviated syntactical forms designating the types). For example, a developer would not worry about how an integer is represented in a machine, but would simply declare a variable tagged with the built-in data type integer. A developer wouldn't worry about how a method in a Java class was implemented by his co-workers, but would simply call the method for his own task.

Another implication of type abstraction is that the instances of types always have names, i.e., abbreviated syntactical forms designating the instances. A variable has its variable name, and a procedure has its signature. (Again, the names are the abbreviations, and not the complete expressions of the instances.)

The third role types play is to improve system performance by static typing during compiling time.

Now let's see what a typed system has missed. We will see in Section 5.2.3 that a type in a typed system always represents a strict subset of the semantics (a class of total recursive

functions) of a typed system (a programming language), and a typed system potentially can have infinitely many types that are user-definable. Because of this, there is no type in a typed system that possesses the complete properties of the class of total recursive functions. In other words, some valuable properties in the class of total recursive functions cannot be carried by the types that can be implemented practically. (For example, so far, there hasn't been a type yet in a typed system that has the ordering relations discussed in Section 2.3). Lacking powerful built-in operators that are derivable from the properties of the class of partial recursive functions makes the sharing of some application-independent code difficult. As demonstrated in Section 4, the lack of such powerful built-in operators may hinder typed systems' ability to address many challenges in the fields of knowledge management and artificial intelligence.

5.2.2 Untyped Systems

Given the understanding that a typed system is one in which individual elements need to be assigned (tagged) with types and potentially there are infinite many user-definable types, we say that an untyped system is one in which there is only one type. Because of this, there is no need to tag a type to individual elements in an untyped system [6]. The pure lambda calculus is a well-known untyped system. The EP data model is untyped. The primitive (built-in) types in programming language, e.g., integers and strings, are untyped by themselves. An untyped system, e.g., the lambda calculus, remains to be untyped after it incorporates another primitive untyped system, e.g., integers [6]. (In this case, one untyped system serves as constants. The syntax of one untyped systems is different from the syntax of the other, and therefore tagging types to elements is not necessary unless a static typing during compile time was required for performance purpose.)

In this paper, we are interested in two kinds of untyped systems: the untyped systems which take a class of partial recursive functions as the semantics, and the untyped system, i.e., the EP data model (and therefore Froglingo), which takes a class of total recursive functions as its semantics. In the following subsections, we discuss why the former ones need to be adapted with typed systems for programming practice and the latter doesn't need to.

5.2.2.1 Partial Recursive Functions

The lambda calculus is a untyped system that take a class of partial recursive functions as semantics. Functions are the only type in the semantic algebra, and lambda expressions are the only type in the syntactic algebra. An implemented system for the lambda calculus would do error-checking by reporting errors when developers enter sequences of symbols that don't assemble lambda expressions. It would provide abstractions by automatically calculating the normal form when developers provide valid lambda expressions even the developers had no knowledge about the beta reduction rule. (This is why we conclude that the lambda calculus is a type.)

Due to the fact that the homomorphism from lambda expressions to partial recursive functions is not decidable, however, there is not an effective algorithm that arranges the lambda expressions in orders according to the properties of the class of partial recursive functions. (In contrast, integers can be divided into odd numbers and even numbers, and a class of total recursive functions can be arranged in orders in the EP data model as discussed in Section 2.3). As a result, it is not possible to semantically arrange lambda expressions in orders. The orders, in the place of types in typed systems, would help to manage business objects with different properties.

5.2.2.2 Total Recursive Functions

Instead of partial recursive functions, the EP data model focuses only on total recursive functions. The EP data model is untyped because the EP terms are the only form of the syntactical algebra and the total recursive functions are the only form of the semantic algebra.

Analogous to integers, the EP terms in a given database are arranged in orders. The ordering relations discussed in Section 2.3, in the role of the types of typed programming languages, can be used to specify the constraints for business objects. Example 3.3, Example 3.6, and the expression for *dinner* at the end of Section 4.1 are sample expressions embedding constraints for business objects. Note that the constraints are not imposed by the EP data model, but by Froglingo which utilizes the ordering relations. The way of carrying business constraints in Froglingo is similar to the data schema in a database management system.

Froglingo is untyped too. Mathematically, there are still infinitely many types in Froglingo for those non-terminating functions beyond the EP data model. But it is not in the interest of practicality to develop a typed system in which each user-defined type is involved with non-termination process.

5.2.3 What types are

we say that a type is a language homomorphic to an algebra; therefore a type has a syntactical aspect and a semantic aspect. Similar to the relational algebra discussed in 5.1, primitive data types like integers and user-defined data types in programming languages are those whose syntactical algebras are seen as identical to the semantic algebras. Here, a type can be simply viewed as an algebra, i.e., a set of values and a set of operations on the values [8]. This is not the case in the EP data model where the homomorphism function depends on a specific database. Separating the syntactical aspect from the semantic aspect is also important for the lambda calculus, because its homomorphism is not decidable.

In this paper, a type is viewed actually not more than a language.

In a typed programming language, in which a class of partial recursive functions is expressible, we observe that a type, such as integers or a user-defined type, always represents a strict subset of the class. (The development of types is intended to avoid those values from the class that may cause non-termination, and to facilitate the management of those business objects with different properties.) Therefore, there are infinitely many types potentially definable and desirable by users [6]. User-defined types are the signals of typed systems.

5.3 Higher-Order Functions

Given an initial set, a function is defined as a binary relation, i.e., a set of pairs, on the set such that no two distinct pairs have the same first coordinate. The first coordinate of a pair is called an *argument*, and the second a *value*. When a function takes other functions as its arguments and (or) values, it is called a higher-order function. The lambda-calculus defines a class of higher-order and partial recursive functions. The EP data model defines a class of higher-order and total recursive functions. A class of higher-order functions is called an applicative structure [4 and 25].

Instead of primitive types such as integers only, taking a functions as a parameter and (or) a return value of another function provides an additional flexibility for developers in software development and maintenance. Analogously in English, for example, it would

be easier to say “Whom am I speaking with?” than to say “What is the name of the person with whom I am speaking?”.

Typed systems (e.g., functional and some imperative programming languages) have already supported the feature at a limited scope. To maximize the flexibility higher-order functions can provides, however, a language system shall define that higher-order functions is untyped. In the lambda calculus, for example, any lambda expression can be applied to another lambda expression. In the EP data model, we can have the following assignments an EP database:

$D\ A = 1;$
$F\ D = D;$
$F\ F = 3;$
$F\ 2 = 5;$

Here, F is a higher-order function because applying it to the function D returns function D. The untyped EP data model also allows self-application, e.g., $F\ F$ which yields the return value *3* in the above expressions.

In the Haskell expression: *compose f g x= f (g (x))*, the function *compose* appears to be untyped, but actually typed [9]. Here, f and g are not functions but variables for functions. For example, applying the function *compose* to an integer, e.g., *compose 4*, or to itself is certainly not a defined operation. As another example, applying *compose* to a function always returns another function, but not an integer.

6. Conclusion

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. As the result, the new monolith is expected to improve productivity while it doesn’t lose functionality.

The EP data model is untyped. It is untyped with unique characteristics such that it is a novel approach to many challenges facing traditional technologies. The untyped system consistently arranges business data in higher-order functions; therefore, an application system in Froglingo is easy to be adapted to new requirements. The high feature scalability should be particularly helpful in application areas such as knowledge management and artificial intelligence, where the initial understanding of a problem is often imperfect. The untyped system indiscriminately treats diverse application objects that would otherwise be defined as different data types in traditional technologies, and therefore allows users to specify their queries through a single entry field. The untyped system supports searches based on key words and phrases against a database and therefore allows users or software applications to approach precisely their final query results by starting with a few key words and phrases. The flexible user interface is expected to be particularly useful in the application areas of knowledge management and artificial intelligence, where there are great numbers of objects, and each object has hundreds and perhaps even thousands of attributes. The untyped system consistently stores common attributes of multiple objects only once in database, and therefore preserves the similarities between the multiple objects. The ability to preserve (rather than mine) similarities is expected to be particularly helpful in the application areas of knowledge management and artificial intelligence, where similarity is an important concept.

A data model is defined to be a homomorphism from a language to an algebra such that the algebra includes at lease one relation and each operation of the algebra terminates on every input. This definition preserves the essence of the concept of a data model that database management systems started with in the 1970s. It is the essence of keeping a programming language easy to use in software development and maintenance.

One may view a linked list as the easiest if he/she only needs to represent a sequence of objects; and a relational DBMS as the easiest if tables are the only concern. But to construct and to maintain arbitrary applications, and to communicate between applications, it has been assessed that Froglingo, incorporated with a total-recursive-equivalent data model, achieves the greatest possible ease.

In the paper, the notion of programming language was used always for a language that is Turing-complete. Therefore a language is not categorized as a programming language in this paper if it is not Turing-complete. Many strongly typed languages, such as Nominal System T [15], express strict subsets of total recursive functions. In the authors’ best knowledge, the EP data model is the first language exactly expressing a complete class of total recursive functions.

7. Acknowledgment

We thank Harold Boley for his hints about typed/untyped functional programming and combinatory logic, Rong Hu for her comments about work related to the cooking recipe advisor, Simon Peyton-Jones for his encouragement during the development of this paper. This paper also incorporated many comments from anonymous reviewers.

References

[1] Abiteboul, S., Hull, R., ianu, V (1995). Foundations of Databases. Addison-Wesley Publishing Company.

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

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

[4] Barendregt, H. P. (1984). The Lambda Calculus - its Syntax and Semantics. North-Holland.

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

[6] Cardelli, L., Wegner, P. (1985). On Understanding Types, Data Abstraction, and Polymorphism, *Computing Surveys*, 17 (4) 471-522, December 1985.

[7] Chen, P. (1976). The Entity-Relationship Management Model – Toward a Unified View of Data, ACM Transactions on Database Systems. 1 (1) March 1976, 9 – 36.

[8] Cleaveland, J. C (1986). An Introduction to Data Types. Addison-Wesley Publishing Company.

[9] Davie, A. J. T. (1992). An Introduction to Functional Programming Systems Using Haskell. Cambridge University Press.

[10] Dikovsky, A. J. (1993). On the Computational Complexity of Prolog Programs. Theoretical Computer Science 119 (1) 63-102.

[11] Doets, K. (1994). From Logic to Logic Programming. The MIT Press.

[12] Elmasri, R., Navathe, S. B. (1994). Fundamentals of Database Systems, Second Editions. The Benjamin/Cummings Publishing Company, Inc.,

[13] Gyssens, M., Paredaens, J., Bussche, J. V., Gucht, D. V. (1994). A Graph-Oriented Object Database Model, *IEEE Transactions on Knowledge and Data Engineering* 6 (4) 572 - 586.

[14] Hammer, M., McLeod, D.(1981). Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems* 6 (3) September 1981, p. 351 – 386.

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

[16] Leake, D. (1996).Case-Based Reasoning: Experience, Lessons, and Future Directions. Menlo Park: AAAI Press/MIT Press.

[17] Lloyd, J.E. (1994). Practical Advantages of Declarative Programming, *In*: Joint Conference on Declarative Programming.

[18] Pitts, A. M. (2010). Nominal System T. POPL'10, January 17-23, 2010, Madrid, Spain.

[19] Revesz, P. (2002). Introduction to Constraint Databases. Springer-Verlag New York, Inc.

[20] Shipman, D. W. (1981). The Functional Data Model and the Data Language DAPLEX, *ACM Transactions on Database Systems*, 6 (1) 140 – 173.

[21] Steedman, D. (1993). X.500 - The Directory Standard and its Application, Technology Appraisals.

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

[23] Xu, K. H., Zhang, J., Gao, S (2010). An Assessment on the Easiness of Computer Languages. *The Journal of Information Technology Review*, p. 67 - 71.

[24] Xu, K. H., Zhang, J., Gao, S. (2010). Higher-order Functions and their Ordering Relations, *In*: The Fifth International Conference on Digital Information Management (ICDIM 2010).

[25] Xu, K. H., Zhang, J., Gao, S., McKeown, R. R.. "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.

[26] Xu, K. H., Zhang, J., Gao, S. (2010). 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.

[27] Xu, K. H., Zhang, J. Gao, S. (2009). Assessing Easiness with Froglingo, The Second International Conference on the Application of Digital Information and Web Technologies, 2009, p. 847 - 849.

[28] Xu, K. H., Zhang, J. 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

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