# High-Level Functions and their Ordering Relations for Knowledge Representation and Mechanized Reasoning

Kevin Xu, Jingsong Zhang, Shelby Gao

Froglingo Development Association,
2306 Johnson Circle, Bridgewater, New Jersey, United States
{kevin, jingson, shelby}@froglingo.com

**Abstract.** High-level functions are the sole elements in a class of recursive functions. The functions are related to each other through application, i.e., applying an argument to a function yields a value while the argument and the value are also functions. Representing knowledge as high-level functions and reasoning through the relationships between the functions are significant. By doing so exclusively, we achieve both the maximum expressive power and "the greatest possible ease" in knowledge representation and reasoning. By introducing Froglingo in this paper, we demonstrate how knowledge is represented in high-level functions and reasoning is accomplished through the relationships between the functions. Froglingo is a language that exactly takes advantage of high-level functions and the relationships between the functions. We also use Froglingo to approach a sample application that appears to be a challenge to traditional technologies.

## 1 Introduction

Given a set, a function is defined to be 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 is allowed to be in the set, i.e., arguments and values are also can be functions, the resulting function is called a high-level function. Given an initial set (with at least two elements), the collection of all the (high-level) functions including the initial set is called a class of partial recursive functions. It is interpreted as the semantics of a programming language, i.e., all what a computer can do.

A partial recursive function may not terminate on certain arguments. For example, a software program for the query: "print out all the paths from vertices A to vertices B in a directed graph", is not terminating on a directed graph in which there is a cycle between A and B. (There would be a path having infinite vertex ABABA… if there is a edge from A to B, and another edge from B to A). Therefore, all the software programs with good quality are well tuned during the development and maintenance such that they terminate on all the possible arguments. The set of all the software programs that are terminating unconditionally has a corresponding semantics: a class of total recursive functions, a well-restricted subset of the partial recursive functions.

A function may strictly take one argument at a time before producing a value. This is the original form of a function when the concept of function was mathematically introduced (see the sample text book [4]). It was used to interpret the semantics of many formal languages such as Lambda Calculus and Combinatory Logic [1]. However, a function taking one or more than one arguments at a time before producing a value is the most commonly used form in almost every other field including the semantic interpretations of imperative programming languages. Fortunately, we don't have to worry about the functions with multiple arguments as far as computability is concerned because the functions having multiple arguments can be expressed by semantically equivalent unary functions, i.e., those taking one argument at a time. The process of converting the functions with multiple arguments into unary functions is called "currying" [1].

In this paper, we stick with the form of unary (high-level) functions. The high-level functions we refer to in this paper are all in this form.

Now, we turn our attention to languages. We have programming languages, including Froglingo, with the semantics equivalent to a class of partial recursive functions. We have other languages such as the relational algebra that are bound by and restrictively within a class of total recursive function. The EP data model, Froglingo without variables, is a language semantically equivalent to a class of total recursive functions [5 and 8]. A language like the EP data model is significant. It enforces the software applications to be constructed to terminate unconditionally. It can express all the possible software applications that are practically meaningful, provided that the space was unlimited. Because it always terminates on all the possible arguments, it along with a programming language was objectively assessed as "the greatest possible ease" in software development and maintenance in the articles [9, 8, and 6]. In other words, it is a consistent tool to construct as much business (finite) data as a software application desires without a possibility of exceptions.

Regardless of the assessment, the EP data model has a set of built-in operators. They are semantically nothing but the ordering relations among high-level functions, i.e., those derivable from the relationships between function, arguments, and values. They are more expressive than those in traditional data models such as the containment relationships in the hierarchal data model and the relational algebra in the relational data model. In addition, many applications that challenge the traditional software technologies can be uniformly expressed in Froglingo.

In this paper, we introduce Froglingo. It is aimed for readers to see the roles of high-level functions and their relationships in data presentation and query.

In the title and the abstract of this paper, the terminologies "knowledge" and "reasoning" were used. They are nothing more than the software applications we construct for business needs. The intention of mentioning them in the paper is to share the work on Froglingo with the research community in the field of artificial intelligence.

In Section 2, we give the definitions of term, assignment, and database. They are the sufficient concepts for finite data construction. We further discuss normal forms and reduction rules that allow arbitrary terms to be evaluated against a given database. In Section 3, we introduce the built-in operators stemming from the ordering relations among high-level functions. A few examples are given to show that they can be used

to express many queries that are the challenges to traditional software technologies. A recipe advisor is given separately in Section 4 due to the lengthy discussion. The concepts up to Section 3 constitute the EP data model. In Section 5, we add variables on the top of the EP data model. The concepts up to Section 5 constitute Froglingo, a Turing-machine equivalent language.

Through all the sections, we primarily focus on the concepts from the view of software development. For the formal aspects, i.e., theorems and mathematical proofs, readers may reference [5, 8, and 10] for more information.

## 2 Finite Data Presentation

In traditional data models, an entity is either dependent on one and only one other entity, or independent from the rest of the world. The functional dependency in relational data model and the child-parent relationships in the hierarchical data model are typical examples. This restriction, however, doesn't reflect the complexities of the real world that can be managed by using a computer. The logic of the EP data model is that if one entity is dependent on entities, then those entities are precisely two in number. Drawing terminology from the structure of an organization or a party in article [11], one dependent entity was called enterprise (such as organization and party), the other was called participant (such as employee and party participant), and the dependent entity was called participation. An enterprise consists of multiple participations. Determined by its enterprise and its participant, a participation yields a value, and this value is in turn another enterprise.

The EP data model is described as a formal language. The core concepts are terms, assignment, database, normal form, and reduction.

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

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

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

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

**Notation 2.2** 1: Given an application $(m\ n) \in T$; $m$ is called the left sub-term, and $n$ the right sub-term. In this paper, an application is also called a comb-term.

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

Unlike Lambda Calculus, the EP database doesn't have variables. Note that Froglingo does have variables as discussed in Section 4. EP carries high-level functions by using identifiers. Lambda Calculus doesn't have identifiers. Assignment is the starting point in the EP database that carries high-level functions.

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

**Notation 2.4** 1. Given an expression $a \equiv b$, the symbol $\equiv$ indicates that the two symbols $a$ and $b$ are identical.
2. Given an expression $a \not\equiv b$, the symbol $\not\equiv$ indicates that the two symbols $a$ and $b$ are not identical.
3. Let $\mathbb{N}$ be the set of natural numbers {0, 1, 2, …}.
4. Let $m, n_0, ..., n_i \in T$, here $i \in \mathbb{N}$. We write:

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

We further write: $\bar{n} \cong n_0, ..., n_i \in T$; and $\| \bar{n} \| = i$.
5. Given a term $m\ \bar{n}$, $m$ is called a left-most term of $m\ \bar{n}$, here $\| \bar{n} \| \geq 0$. Note that the terms $m\ n_0$, $m\ n_0\ n_1$, …, and $m\ n_0 ... n_i$ are also called the left-most terms of $m\ n_0 ... n_i$.
6. Let $n_0, ..., n_i, m \in T$, here $i \in \mathbb{N}$. We call $m$ in the term $((n_0 ... (n_i\ m)\ldots))$ a right-most term. Similarly, the terms $(n_i\ m)$, $((n_{i-1}\ (n_i\ m))$, and $((n_0 ... (n_i\ m)\ldots))$ are also right-most terms.

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

**Definition 2.5** An EP database $D$ is the union of a set of terms $T \subset T$ and a set of assignments $A \subset A$, i.e., $D = T \cup A$, such that the following are true:
1. If an application $m\ n$ is in $D$, the left sub-term $m$ must not be a constant and the right sub-term $n$ must not have an assigner, i.e.,

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

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

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

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

The above restrictions force users to enter those and only those business applications that are semantically equivalent to a class of total recursive functions [5].

We adduce a few EP database examples:

**Example 2.6** 1. The function $s(x) = x^2$ can be expressed in EP database $S$ = {*s, s 1 =: 1, s 2 =: 4, …, 1, 2, 3, …*}.
2. The function f(x, y) = x + y can be expressed in EP database $F =: \{f, f\ 1, f\ 1\ 1 =: 2, f\ 1\ 2 =: 3, \ldots f\ 2, f\ 2\ 1 =: 3, f\ 2\ 2 =: 4, \ldots 1, 2, 3, \ldots\}$.
3. A school administration database application can be expressed as: $D$ = {*SSD, SSD John, SSD John birth =: '6/1/90', SSD John SSN =:123456789, College, College admin, College admin* (*SSD John*), *College admin* (*SSD John*) *enroll =*: '*9/1/08*', *College admin* (*SSD John*) *Major =*: *College CS, College, College CS, College CS100, College CS100* (*College admin* (*SSD John*)), *College CS100* (*College*

*admin* (*SSD John*)) *grade =*: "*F*", '*6/1/90*', *123456789*, '*9/1/08*', "*F*"}. It is a database for the "Social Security Department in the United States"; and for a central administration office in a college or university. In the "Social Security Department" (SSD), each resident has his/her social security number (SSN), name, and birth date, etc. In the college or university, a student registers; the college has departments; each department offers classes; and each class has students who attend.

4. $G$ = *{v1 v2 =: v2; v2 v1 =: v1; v2 v3 =: v3}*. (A directed graph with a circle)
5. The union of the sets above $S \cup F \cup D \cup G$ form an EP database.

The normal form and reduction rules are adduced below.

**Definition 2.7** Given a database $D$, the set of normal forms $NF$ is defined as follows:

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

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

**Definition 2.8** Given a database $D$, we have the one-step evaluation rules, denoted as $\rightarrow$:

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

**Definition 2.9** Let $m, n \in D$. If there is a finite sequence $l_0, \ldots, l_n \in D$ , where $n \geq 0$, such that $m \equiv l_0,, l_0 \rightarrow l_1, \ldots, l_{n-1} \rightarrow l_n, l_n \equiv n$, then

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

Given the databases in Example 2.6.5, the following examples are effective reductions:

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

It has been concluded that an arbitrary term under a database can be effectively reduced to one and only one normal form; the EP data model, as a formal language, is semantically equivalent to a class of total recursive functions; and it is consistent, sound, and complete. Please reference [5] for the detail.

## 3 Ordering Relation

The occurrence of applying an argument to a function depends on both the function and the argument; and thereafter it depends on a sub-term of the functions and the argument. This leads to the development of the dependent relations.

The value, which is different from the occurrence of applying an argument, doesn't depend on the function and the argument because it exists independently. However, the value is derivable from the occurrence; and therefore it is derivable from the function, from the argument, and from sub-terms of the function and the argument. This leads to the development of the derivative relations.

We discuss dependent and derivative functions in this section.

**Definition 3.1: 1.** Given a comb-term term *m n*, the operators {+ and {- in the following expressions are defined such that the expressions are evaluated to be true:

*m n* {+ *m*, (called functional dependency)

*m n* {- *n*, (called argumentative dependency)

**2.** Given a term *m*, let *l*, *s*, *r* are a left-most sub-term, a sub-term, and a right-most sub-term correspondingly, then the operators {=+, {=-, and {= in the following expressions are defined such that the expressions are evaluated to be true:

*m* {=+ *l*, (called recursively functional dependency)

*m* {=- *r*, (called recursively argumentative dependency)

*m* {= *s*. (called recursively neutral dependency, either functional or argumentative)

All the operators above have been formally introduced in [10] except for the neutral dependency relation {=. Here are the sample expressions having `true` as the evaluation results:

*SSD John birth* {+ *SSD John*,

*SSD John birth* {=+ *SSD*,

*SSD John birth* {=- *birth*,

*College CS CS100 (College admin (SSD John))* {=- *John*,

*birth* {= *SSD John birth*,

*John* {= *SSD John birth*.

**Lemma 3.2** 1. *m* {+ *n* ⇒ *m* {=+ *n*

2. *m* {- *n* ⇒ *m* {=- *n*

3. *m* {=+ *n, or m* {=- *n* ⇒ *m* {= *n*

The above lemmas say that if a term is dependent on another, so is it recursively, and if a term is functionally or argumentatively dependent on another, so is it neutrally. In other words, {+ and {- are stronger than {=+ and {=-; and {=+ and {=- are stronger than{=. Let's formally define the dependent relationships.

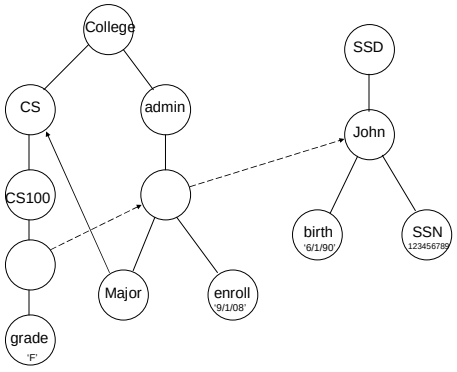
**Definition 3.3**: Given a non-empty set *A*, a strict subset $B \subset A$, and $x, y \in A$, $x$ is dependent on $y$ in $B$, denoted as $x \rho y$, if and only if $x \in B$ implies that $y \in B$, i.e., $\rho = \{<x, y> \mid x, y \in A$; if $x \in B$, then $y \in B\}$. Here we say that $\rho$ is a dependent relation in *B*.

**Proposition 3.4** The relations {+, {-, {=+, {=-, and {= are dependent relations in a database *D*.

The proofs have been given in [5 and 10] for all except for {=. However the proof for {= is clear from its own definition. It has been further proved that {+ and {- are tree-structured. These are essential in set-oriented data updates in addition to its usage on set-oriented queries. The operator {= is not tree-structured, but a partial ordering relation. For example, a term A B (A C) neutrally dependent on both A B and A C; and the terms A B and A C dependent on term A.

The purpose of introducing the dependent relations in this paper is to introduce the pre-ordering relations because the later is based on the former. It is the pre-orderings that are found surprisingly useful in some applications challenging the traditional technologies. Before we start discussing the pre-ordering relations, we further provide a graphical presentation of the school administration database. It will help us to better understand the roles of high-level functions and the ordering relations and therefore to effectively map software applications to high-level functions according to business needs.

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



**Definition 3.5:** 1. Given term *m n* and *q* in a database, if *m n* == *q*, then the operators (+ and (- in the following expressions are defined such that the expressions are evaluated to be true:

*q* (+ *m*, (called functional derivative)

*q* (- *n*. (called argumentative derivative)

**2.** Given terms *m, q, l, s,* and *r* in a database such that:

*m* == *q*,

*l* is a left-most sub-term of *m*,

*s* is a sub-term of *m*,

*r* is a right-most 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*, (called recursively functional derivative)

*q* (=- *r*, (called recursively argumentative derivative)

*q* (= *s*. (called recursively neutral derivative)

Here are a few examples:

*"F"* (+ *College CS CS100 (College admin (SSD John)),*

*"F"* (=+ *College CS CS100,*

*"F"* (=- *SSD John,*

*"F"* (= *College admin,*

*v2* (=+ *v1,*

*v1* (= *v2.*

**Lemma 3.6:** *m* {=+ *n* ⇒ *m* <=+ *n*

*m* {+ *n* ⇒ *m* <+ *n*

*m* {=- *n* ⇒ *m* <=- *n*

*m* {- *n* ⇒ *m* <- *n*

*m* {= *n* ⇒ *m* <= *n*

*m* <=+ *n*, or *m* <=- *n* ⇒ *m* <= *n*

The lemmas say that if a relation is dependent, it is also derivative. In other words, the dependent relations are stronger than derivative relations. The proofs are clear from their definitions.

**Notation 3.7** 1. A relation $\rho$ in a set *X* is reflexive iff *x*$\rho$ *x* for each *x* in *X*. $\rho$ is symmetric if *x*$\rho$ *y* implies *y* $\rho$ *x*, and it is transitive iff *x*$\rho$ *y* and *y* $\rho$ *z* imply *x*$\rho$ *z*.

2. $\rho$ is antisymmetric iff whenever *x* $\rho$ *y* and *y* $\rho$ *x* imply *x* = *y*.

3. A relation $\rho$ is called partial ordering in *X* iff $\rho$ is reflexive, antisymmetric, and transitive.

4. A relation $\rho$ is called pre-ordering in *X* if $\rho$ is reflexive and transitive.

**Proposition 3.8**: (=+, (=- and (= are pre-ordering.

The proofs for (=:

1. It is reflexive because each term is a sub-term of itself, and equal to itself.

2. It is transitive. Given *m* (= *n* and *n* (= *q*, prove that *m* (= *q*.

Since *m* (= *n*, there is a *m1* such that *m* == *m1* and *n* is a sub term in *m1*.

Since *n* (= *q*, there is a *n1* such that *n* == *n1* and *q* is a sub term in *n1*.

We let *n1* to replace *n* in *m1*, denote the resulting term as *m2* in which then *q* is a sub term. According to the theorem 3.5 of [5], we have: *m* == *m1* == *m2*. Therefore, *m* (= *q*.

The proofs for (=+ and (=- were given in [10], which were similar to the proof for (=.

The pre-ordering relations appear loose and irrelevant to the queries supported in traditional database technologies. However, we found some meaningful applications for the pre-orderings. The authors found that the following example plus the one in Section 4 are interesting.

**Example 3.9** The operator (=+ or (=- can be used to express the query: if there is a path from one vertices to another in a directed graph.

Given the data presentation in Example 2.6.4 for a directed graph, 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*.

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*.

Note that the operator (=- can also be used for the same queries if a directed link is represented in the opposite orientation, e.g., *v1 v2* =: *v1*, rather than *v1 v2* =: *v2* in 2.6.4.

By the way, this query is not expressible in the relational algebra and the containment relationships of the hierarchical data model.

## 4 Case Study: A Recipe Advisor

A computer system is required to approximate the knowledge of a professional chef and to interactive with customers as a valuable advisor.

There are following challenges: 1) A consistent method of representing the knowledge of cooking including ingredients, cooking equipment, dish provenance, synonym, preparation methods, preparation steps, and many others that may even not be realized during the system design; 2) The ability of accepting a format of the inputs producible by the customers who don't know the knowledge representation of system; 3) The accuracy of answering customer queries as if it was a professional chef. See the articles [2 and 3] for the state-of-art in this field.

### 4.1 Knowledge Representation

In response to the 2010 Computer Cooking Contest, a workshop of the 9th International Conference on Case-Based Reasoning – ICCBR 2010, the authors propose to indiscriminately represent all the knowledge of cooking in high-level functions. Here are two sample terms approximating two recipes:

```
grill (marinate (rub (New York steak 8 oz) onion salt)) =:dish1;
grill (plate (grill (rub (New York steak 8 oz) garlic salt))
             (sauce chile)
             (cheese cheddar)
       ) =: dish2;
```

The first term embeds the preparation steps of a recipe: 1) Rub New York steak, 8 ounces, with garlic and salt; 2) Marinate it; and 3) grill the steak. The second term embeds the preparation steps of another recipe: 1) Rub New York steak, 8 ounces, with garlic and salt; 2) grill the steak; 3) place steak in a plate and spread with chile sauce and cheddar cheese; and 4) place the plate back to grill.

The terms above don't and will never exactly reflect the recipes in reality. However, Froglingo allows a recipe to be represented as accurate as a software application desires by accumulating more attributes. It doesn't care if an attribute of a recipe is an ingredient, a piece of cooking equipment, or a process. All the attributes are embedded in high-level functions.

Another benefit of doing this is that all the recipes are stored together and those attributes, such as an ingredient and an intermediate preparation step, are also stored together and shared by the recipes. For example, the attribute "Rub New York steak, 8 ounces, with garlic and salt", which appeared in both sample recipes given earlier, is stored once in the database as:

```
rub (New York steak 8 oz) garlic salt;
```

Sharing attributes among recipes helps the system evaluation process in answering queries, as to be discussed in Section 4.3.

### 4.2 Customer Input Format

To simplify the discussion of the case study, we require customers to provide as an input a sequence of phrases separated by a comma ",". Here, each phrase can be a single or multiple words; and the order of the phrases in the sequence is insignificant. There is a built-in word "no", implying negation. Here are a few examples:

```
"beef, salt, grill",
"steak, American, no pepper",
"New York steak, cheddar cheese, no onion".
```

Actually, the system is implemented to accept a textual string as an input. In this case, the system only picks those words that are inventoried in database, and further converts the built-in word "no" to a negation. We will see more discuss in the next section.

### 4.3 Query Expression

A recipe advisor may need to answer a wide range of queries from customers. But the essential type of queries is to precisely find recipes that match a set of attributes. In other words, an answer should not miss a single recipe that possesses the attributes; and should not contain a single one that doesn't satisfy the attributes. We limit ourselves here to discuss the essential queries only.

A term, representing a recipe, embeds indiscriminately all the relevant information including ingredients, preparation methods, and cultures. The key words appeared in customer inputs can be directly used to reference the terms in the database, and therefore to identify those terms (dishes) that embeds the key words as sub-terms. For example, we have the following expression:

```
select $dish where
     $dish (= steak and
     $dish (= American and
```

```
not ($dish (= pepper);
```

This expression is for the customer input: “steak, American, no pepper”. It answers a request in English like: “I like an American steak. But I hate pepper”.

When a database stores thousands of recipes, the query above may bring up hundreds of satisfied answers. To narrow down to a few answers that customers really want, customers can add more attributes in the input sequence.

Synonym can be handled by assignments in Froglingo. If the database is added with the following assignments:

```
hate =: not;
New York steak =: meat beef short_lion;
```

The following inputs will not bring up any dishes having steaks (including New York teak):

“garlic, cheddar cheese, onion, hate meat”.

Many scenarios, such as recipe adaptation and optional ingredients in a recipe, are not considered in the recipe advisor discussed in this paper. Also, the query expressions above may bring up some intermediate preparation steps that customers may not need. However, what was discussed here is sufficient to demonstrate that modeling knowledge in high-level functions and reasoning through the ordering relations appear to be a solution. In addition to the recipe advisor, we may find more applications in the area of artificial intelligence. See a brief discussion in Summary about a possible use in pattern recognition.

## 5 Variables

Variable expands the EP data model to constitute Froglingo, a full language system semantically equivalent to Turing-machine.

The addition of variable in Froglingo doesn’t semantically alter the structure of high-level functions. Therefore, the ordering relations are applicable to a database having variables.

A variable in Froglingo is represented by an identifier proceeded with the symbol $. For example, `$a_variable` and `$student`. Variables can appear in database with two restrictions:

- If a variable appears in an assigner, it must appear in assignee;
- A variable cannot be a left-term in an assignee.

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

```
fac 0 = 1;
fac $n = ($n * (fac ($n - 1)));
```

Syntactically, the first assignment represents a business data; and the second a business logic. However, they are managed together, i.e., both are stored physically together in the same data structure.

Semantically, the two assignments represent the factorial function and they are equivalent to an EP database having infinite assignments:

```
fac 0 = 1;
fac 1 = 1;
fac 2 = 2;
```

```
fac 3 = 6;
…;
```

The variables signal the ability of expressing infinite data in finite algorithms as demonstrated above. Unfortunately, they also indicate the appearances of non-termination processes from the partial recursive functions expressible by Froglingo. (For example, applying the non-integer value `3.5` to the factorial function `fac` above would cause the calculation process not terminating). For the practical needs and for avoiding the non-termination processes, a variable can be defined with a bound. For example:

```
fac 0 = 1;
fac $n:[$n isa integer] = ($n * (fac ($n - 1)));
```

The modified factorial function above will terminate when an arbitrary input is applied.

A variable can be bound to a finite domain. For example, a book store offers 10% discount only to the students at `College`:

```
sale $price $x: [$x {+ SSD and College admin $x != null] =
              ($price * 0.9);
sale $price $y = $price;
```

For a software application to terminate, however, users have to ensure that variables only bring total recursive functions through function recursion. Further, the variables that are involved with the executions of the ordering relations may be required to have finite domains.

## 6 Summary

In this paper, we discussed high-level functions and their ordering relationships. The discussion was performed through introducing Froglingo, a language that is Turing-machine equivalent; and a language that fully takes advantages of high-level functions and their properties such that an objective view suggested its possession of “the greatest possible ease” in software development and maintenance.

Regardless of the assessment on the Froglingo expressive power and easiness, we gave a case study, a recipe advisor, to demonstrate that the high-level functions, (a type-free system, or the concept of function itself as the sole type), offer a suitable data structure for representing knowledge, which once appeared to be a challenge to traditional technologies. It can uniformly express the knowledge as a whole that was modeled separately as user-defined data types in traditional technologies. In addition, it also can express a process, i.e., a sequence of objects in which sub sequences can be embedded.

With the uniform representation of knowledge, many queries that again appear to be a challenge to traditional technologies are expressible by using the ordering relations of high-level functions.

A precise answer to a query that takes as an input a sample set of attributes in the case study of this paper is meaningful in the field of pattern recognition. Assume that a Froglingo database store thousands of unique images, e.g. fingerprints, as high-level functions as the way we did earlier for a recipe advisor. Given an image, we attempt to match it with one and only one in the database. The ordering relations can be used

to partially search the database by randomly providing a sample set of attributes as input, e.g., a sample set of points or lines with their properties. By doing this, we can obtain a set of candidates. This set is guaranteed to include the one to be matched if the given image is indeed among those in the database. By repeating the same process, but choosing a different set of sample attributes, and querying against the previous candidates, a narrowed set of candidates can be found. This approximation process is eventually able to find the desired image. What we gain from this process is the speed because not all the images in database are searched sequentially.

## References


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

2 F. Badra, J. Cojan, A. Cordier, J. Lieber, T. Meilender, A. Millle, P. Molli, E. Nauer, A. Napoli, H. Skaf-Molli, and Y. Toussaint. "Knowledge Acquisition and Discovery for the Textual Case-Based Cooking System WIKIAAABLE". 8th International Conference on Case-Based Reasoning – ICCBR 2009, Workshop Proceedings 2009, Page 249 - 258.

3 N. Ihle, R. Newo, A. Hanft, K. Bach, M. Reichle. "CookIIS – A Case-Based Recipe Advisor". 8th International Conference on Case-Based Reasoning – ICCBR 2009, Workshop Proceedings 2009.

4 R. R. Stoll. "Set Theory and Logic". Dover Publications, Inc. 1963.

5 K. H. Xu, J. Zhang, S. Gao, R. R. McKeown. "Let a Data Model be a Class of Total Recursive Functions". To appear in the 2010 International Conference on Theoretical and Mathematical Foundations of Computer Science (TMFCS-10)".

6 K. H. Xu, J. Zhang, S. Gao. "Froglingo, When A Data Model is Semantically Equivalent to A Class of Total Recursive Functions". To appear in the Journal of Information Technology Review, 2010.

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

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

9 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.

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

11 K. H. Xu and B. Bhargava, "An Introductioin to Enterprise-Participant Data Model", Seventh International Worshop on Database and Expert Systems Applications, September, 1996, Zurich, Switzerland, page 410 – 417.