High-Order Functions and their Ordering Relations

Kevin H. Xu, Jingsong Zhang, Shelby Gao
Bigravity Business Software LLC
(kevin, jingsong, shelby)@froglingo.com

Abstract

High-order functions are the sole elements in a class of recursive functions. The functions are related to each other through application, i.e., applying a function to an argument yields a value where the argument and the value are also functions. Through Froglingo, a language that exactly takes advantage of high-order functions and their properties, we introduce the method of representing business applications including knowledge management in high-order functions and reasoning according to the ordering relations among the high-order functions.

1. Introduction

Given an initial 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 functions are allowed to be arguments and (or) values, the resulting function is called a high-order function. The collection of all the (high-order) functions including the initial set is called a class of partial recursive functions. It is interpreted as the semantics of a Turing-machine equivalent language.

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. Therefore, all the software programs with a reasonable quality are 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 take more than one arguments at a time before producing a value. In this paper, however, we stick with the form of unary (high-order) functions, i.e., a function strictly takes one argument at a time before producing a value. A none-unary function has an equivalent presentation in the form of unary functions. The process of converting none-unary functions to unary functions is called "currying" [1].

Now, we turn our attention to languages. We have programming languages, including Froglingo, with the semantics equivalent to a class of partial recursive functions. On the other hands, we have data models 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 [4]. A data model 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.

By introducing Froglingo in this paper, we suggest a methodology of representing software applications including knowledge management in high-order functions and reasoning on the applications according to the relationships among the functions. In Section 2, we discuss the concepts in which the software applications are represented as high-order functions. From Section 3 to Section 5, we introduce the built-in operators that reflect the ordering relations among the business data. The concepts up to Section 5 constitute the EP data model. In Section 6, the concept of variable in Froglingo is introduced.

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 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 as in article [9], one dependent entity is called enterprise

(such as organization and party), the other is called participant (such as employee and party participant), and the dependent entity is called participation. An enterprise consists of multiple participations. Determined by its enterprise and its participant, a participation yields a value, and this value is in turn another enterprise.

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.

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 called 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 left-most sub-term of the application $(m\ n)$, and n the right sub-term and a right-most sub-term of the application $(m\ n)$.
4. If q is a left-most sub-term of m, then q is also a left-most sub-term of the application $(m\ n)$. Similarly, if q is a right-most sub-term of n, then q is also a right-most 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 re-written the following way: f 3, country state and 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 6.)

A term can be assigned with a value:

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 followings 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 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 applications that are semantically equivalent to a class of total recursive functions [4].

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), 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 John birth =: '6/1/90';
SSD John SSN =:123456789;
College admin (*SSD John*) *enroll =: '9/1/08'; College admin* (*SSD John*) *Major =: College CS;*
College CS100 (*College admin* (*SSD John*)) *grade =: "F";*

According to Definition 2.4, the sub-terms appeared in the assignee and assigner of a database, e.g., *SSD*, *John*, and *SSD John* in the sample database above, are separated entities. We don't show them normally 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:

Example 2.7 A directed graph with a circle:

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

Representing and reasoning on knowledge has been a challenge to traditional technologies [2]. The example below of knowledge representation and the corresponding query expressions in Sections 3 and 5

for reasoning show that the EP data model is a solution.

Example 2.8 A database approximating two recipes:

1. *grill (marinate (rub (beef short loin 8 oz) garlic salt)) =: dish1;*
2. *grill (plate (grill (rub (beef short loin 8 oz) garlic salt)) (sauce chile) (cheese cheddar)) =: dish2;*

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

The database above doesn't 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 indiscriminatingly embedded in high-order functions.

Another benefit of doing this is that all the recipes are stored together with the attributes shared among them. For example, the attribute "Rub beef short loin, 8 ounces, with garlic and salt", which appeared in both sample recipes given earlier, is stored once in the database as:

rub (beef short loin 8 oz) garlic salt;

3. Dependent Relations

An application depends both on its function and on its argument; and thereafter it depends on the sub-terms of the function and the argument. This leads to the development of the dependent relations.

Definition 3.1 1. Given an application $m\ n$ in a database D, the operators $\{+$ and $\{-$ in the following expressions are defined such that the expressions are evaluated to be true:

$m\ n\ \{+\ m$
$m\ n\ \{-\ n$

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

$m\ \{=+\ l$
$m\ \{=-\ r$
$m\ \{=\ s$

Here are the sample expressions with the value of true:

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.

We will define a type of ordering relations – dependent relations and show that the operators above are dependent relations.

Definition 3.2 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 ρ is a dependent relation in B.

Proposition 3.3 The relations $\{+$, $\{-$, $\{=+$, $\{=-$, and $\{=$ are dependent relations in a database D.

The first 4 operators above have been introduced and proved as dependent relations in [8] except for the last one $\{=$. However the proof for $\{=$ is clear from its own definition.

Notation 3.4 The operators $\{+$, $\{-$, $\{=+$, $\{=-$, and $\{=$ are called functional dependency, argumentative dependency, recursively functional dependency, recursively argumentative dependency, and recursively neutral dependency correspondingly.

The EP data model supports set-oriented operations in conjunctions with ordering relations. We give two examples to demonstrate it in the rest of the section.

Example 3.5 Select all the participations of *John* in the database of Example 2.5:

select $x where $x {=- John;

The output would be *John, SSD John, College admin (SSD John), College CS CS100 (College admin (SSD John))*.

Note that a variable, e.g., *$x* in the example above, can appear in the *where* clause of a *select* operation. Syntactically, it is the same as those introduced in Section 6. Semantically, however, it plays no role at all to the construction of a database since it is local to the *select* expression.

Given the recipes representation in Example 2.8, the recursively neutral dependency, i.e., $\{=$, is found useful in supporting the queries from the customers in a restaurant who know nothing about knowledge representation but remembering a few attributes about dishes. For example, she may express: "I want a marinated beef short loin. I hate cheese". A system having the EP data model may simply translate her query into a list of key words with possible negations: "beef, short, loin, marinate, no cheese". Further, the system is able to translate the query to an EP expression:

Example 3.6 *select $dish where $dish {= beef and $dish {= short and $dish {= loin and $dish {= marinate and not ($dish {= cheese);*

The system would include *dish1*, but not *dish2* of Example 2.8 as the result.

When a database stores thousands of recipes, the query above may bring up hundreds of satisfied answers. To narrow down to fewer answers that customers really want, the customers can describe more attributes as a part of queries.

Before ending this section, we give the relationships among the dependent relations.

Lemma 3.7 1. $m \{+ n \Rightarrow m \{=+ n$

2. $m \{- n \Rightarrow m \{=- n$

3. $m \{=+ n$, *or* $m \{=- n \Rightarrow m \{= n$

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 $\{=$.

Dependent relations imply partial orderings. The first four relations in Notation 3.4 are further tree-structured orderings [8]. The operator $\{=$ is not tree-structured, but only partial ordering. 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*.

4. Equation

The assignments in a database allow terms semantically equal. Driven by a set of reduction rules, assignments lead to a set of equations in a given database. The equations, along with the dependent relations discussed in Section 3, lead to the pre-ordering relations that are to be discussed in Section 5. In this section, we introduce the normal forms, the reduction rules, and the resulting equations.

Definition 4.1 Given a database *D*, the set of normal forms *NF* is defined as follows:

1. All the constants are normal forms, i.e.,

$c \in C \Rightarrow c \in NF$

2. All the terms in *D* that don't have assigners are normal forms by themselves, i.e.,

$t \in D - A \Rightarrow t \in NF$

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

Definition 4.2 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.,

$p \in P \cap p \notin D \Rightarrow p \rightarrow$ `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* ∈ *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 4.3 Let *m, n* ∈ *T* with the environment of a given database *D*. If there is a finite sequence $l_0, \ldots, l_q \in T$, where $q \geq 0$, such that $m \equiv l_0, l_0 \rightarrow l_1, \ldots, l_{q-1} \rightarrow l_q, l_q \equiv n$, then

1. *m* is effectively, i.e., in finite steps, reduced to *n*, written as $m \rightarrow_{EP} n$.

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 derived from the environment of *D*.

Example 4.4 The followings are a few equations from the databases in Example 2.5, 2.7, and 2.8:

SSD John SSN == 123456789;
(*College Admin (SSD John) Major*) == *College CS;*
v1 v2 v1 == v1;
v1 v2 v1 v2 v1 == v1 v2 v1;

5. Pre-Ordering Relations

The normal form, resulting from an application, doesn't have to depend on the function and the argument of the application because it exists independently. However, the normal form is derivable from the application; and 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 the pre-ordering relations.

Definition 5.1 1. Let *m, n, q* ∈ *T* and *D* 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*
q `(-` *n*

2. Let *m, q, l, s, r* ∈ *T*, *D* a database, *m* == *q*, *l* is a left-most sub-term of *m*, *s* is a sub-term of *m*, and *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*
q `(=-` *r*
q `(=` *s*

Example 5.2 Given the databases in Example 2.5 2.7, and 2.8, here are a few Boolean expressions with true values:

"F" `(+` *College CS CS100 (College admin (SSD John));*
"F" `(=+` *College CS CS100;*
"F" `(=-` *SSD John;*
"F" `(=` *College admin;*
v2 `(=+` *v1; v1* `(=+` *v2;*
v1 `(=` *v2;*

Notation 5.3 1. A relation ρ in a set X is reflexive iff $x \rho\ x$ for each x in X. ρ 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. ρ is antisymmetric iff whenever $x\ \rho\ y$ and $y\ \rho\ x$ imply $x = y$.
3. A relation ρ is called partial ordering in X iff ρ is reflexive, antisymmetric, and transitive.
4. A relation ρ is called pre-ordering in X if ρ is reflexive and transitive.
Proposition 5.4 (=+, (=- and (= are pre-ordering. Here is the proof 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 $n1$.
Let $n1$ replace n in $m1$, and we denote the resulting term as $m2$ in which q is a sub term. According to the theorem 3.5 of [4], we have: $m == m1 == m2$. Therefore, m (= q.
The proofs for (=+ and (=- were given in [8], which were similar to the proof for (=.

The pre-ordering relations appear loose and irrelevant to the queries supported in traditional database technologies. However, they were found useful in the following examples.
Example 5.5 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*.

In the real life, an object can have many different names. For example New York steak is actually referring to beef short loin, which can be represented as an assignment in the EP data model:
New York steak =: beef short loin;
Then the query: "I want a marinated New York steak. I hate cheese" would be translated to the following expression:
Example 5.6 *select $dish where $dish (= New and $dish (= York and $dish (= steak and $dish (= marinate and not ($dish (= cheese);*
This expression may not exactly produce the result that Example 5.5 did, but it guarantees that the dish *dish1* of Example 2.8 will be included as a part of the outcom.
Notation 6.7 The operators (+, (-, (=+, (=-, and (= are called functional derivative, argumentative derivative, recursively functional derivative, recursively argumentative derivative, and recursively neutral derivative correspondingly.
Lemma 5.8 $m \{=+\ n \Rightarrow m$ (=+ n
$m \{+\ n \Rightarrow m$ (+ n
$m \{=-\ n \Rightarrow m$ (=- n
$m \{-\ n \Rightarrow m$ (- n
$m \{=\ n \Rightarrow m$ (= n
m (=+ n, or m (=- $n \Rightarrow m$ (= 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.

6. Variable

Using variable in Froglingo is a method of carrying (possibly infinite) data with finite expressions, and it makes Froglingo Turing-machine equivalent.

A variable in Froglingo is represented by an identifier preceded with the symbol $. For example, $*$a_var* and $*student*. A variable is a term too. The definitions 2.1 and 2.4 are extended with the following additions:
Definition 6.1 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 database:
Example 6.2 The factorial function
fac 0 = 1;
*fac $n = ($n * (fac ($n - 1)));*

Semantically, the expressions above are equivalent to a database having infinite assignments: *fac 0 = 1; fac 1 = 1; fac 2 = 2; fac 3 = 6; ….* It demonstrates that variables semantically add nothing to the EP data model, but syntactically the finite expressions for possible infinite entities (semantics).

A variable can be restricted to prevent unwanted data from being its instances. As a result, the entities represented by the terms containing variables obey the ordering relations discussed in this paper. To demonstrate this, we modify the recipe of Example 2.8: 1) Rub beef short loin, 8 ounces, with garlic (or pepper) and salt; 2) Marinate it; and 3) grill the short loin. The corresponding Froglingo expression is:
Example 6.3 A recipe with optional ingredients
grill (marinate (rub (beef short loin 8 oz)
($x: [$x == garlic or $x == pepper])
salt)) =: dish3;

This assignment is semantically equivalent to the two separated assignments:

grill (marinate (rub (beef short loin 8 oz) garlic salt)) =: dish3;
grill (marinate (rub (beef short loin 8 oz) pepper salt)) =: dish3;
Therefore, the query expressions in Example 3.6 and 5.6 will produce the same outcomes with the addition of *dish3* if all the assignments for the recipes are in a single database.

When a variable allows one to express possibly infinite data in finite representation, it may also introduce a non-termination process. It is users' responsibility to avoid it.

6. Summary

By organizing software application in high-order functions, many queries can be expressed by using a set of built-in operators, which are semantically nothing but the ordering relations among high-order functions. While applicable to the finite data in the EP data model, the operators are equally applicable to the infinite data in variables.

The way of thinking in high-order functions offers immediate solutions to many applications that appear to be challenges to traditional technologies. It was demonstrated through a few examples in business applications, especially in knowledge management.

Representing business applications solely in high-order functions is aimed to increase the productivity of software development and maintenance [5 and 3].

7. References

1. H. P. Barendregt. "The Lambda Calculus - its Syntax and Semantics". North-Holland, 1984.
2. K. Trentelman. "Survey of Knowledge Representation and Reasoning Systems", Defence Science and Technology Organization, Australia, DSTO-TR-2324, July 2009.
3. K. H. Xu, J. Zhang, S. Gao. "Froglingo, an Monolithic alternative to DBMS, Programming Language, Web Server, and File System". The Fifth International Conference on Evaluation of Novel Approaches to Software Engineering, 2010.
4. K. H. Xu, J. Zhang, S. Gao, R. R. McKeown. "Let a Data Model be a Class of Total Recursive Functions". The International Conference on Theoretical and Mathematical Foundations of Computer Science (TMFCS-10), 2010.
5. K. H. Xu, J. Zhang, S. Gao. "An Assessment on the Easiness of Computer Languages". The Journal of Information Technology Review, 2010.
6. 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.
7. 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.
8. K. H. Xu. "EP Data Model, a Language for Higher-Order Functions". Manuscript unpublished, March 1999. http://www.froglingo.com/ep99.pdf.
9. 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.