FROGLINGO
A Monolithic Alternative to DBMS, Programming Language, Web Server and File System

Kevin H. Xu, Jingsong Zhang and Shelby Gao
Bigravity Business Software LLC, U.S.A.

Keywords: Total recursive functions, Computability, Productivity, Data model, Programming language, DBMS, Data exchange, Access control, Web server, File system.

Abstract: Application software started with a monolithic architecture in the 1960s, i.e., a single executable file for the entire application. For better productivity in software development, software application in a typical corporate environment today consists of multiple components including off-the-shelf products. Froglingo is a unified solution for database management and programming language. It is an alternative to the combination of software technologies including DBMS, programming language, web server, and file system. The Enterprise-Participant (EP) data model, Froglingo without variables, is a computer language equivalent to a class of total recursive functions. It brings the monolith back to application software. In this paper, we show that Froglingo is a monolith and demonstrate that this monolith with the EP data model improves the productivity in both software development and software maintenance.

1 INTRODUCTION

A typical database application system in a corporate environment today needs DBMS (such as Oracle and MySQL), programming language (such as Java and C#), and middleware components including web servers (such as Websphere and WebLogic), data exchange tools (such as Hibernate and LINQ), and centralized authentication tools for multiple web applications (such as IBM TAM and RSA ClearTrust). With the combination of the current technologies, we have made limited progress in effectively and efficiently developing and maintaining software applications (Loucopoulos et al., 2006).

Froglingo (Xu and Zhang, 2010) is a unified solution for software development and maintenance, and an alternative to DBMS, programming language, file system, and web server. It is a “database management system (DBMS)” to store and to query business data; a "programming language" to support business logic; a "file system" to store and to share files; and a "web server" to host multiple applications and to interact with users across network. It does more than combine existing technologies. It is a single language that uniformly expresses both data and application logic, and it is a system supporting integrated applications without using application-based data exchange component and data access control mechanism.

The EP (Enterprise-Participant) data model is at the centre of Froglingo. It is semantically equivalent to a class of total recursive functions (Xu et al., 2010). The equivalence for a data model dictates that the EP data model is nothing but high-order functions and the ordering relations among the functions (Xu et al., 2010).

Representing software applications in high-order functions and their ordering relations is not only applied to business data, i.e., finite data, by using the EP data model, but also applied to business logic, i.e., infinite data, by using Froglingo, the extended system having variables beyond the EP data model (Xu, 1999).

It is not surprising that Froglingo is a programming language, i.e., a Turing-machine equivalent system reaching the full capacity of what a computer can do (Xu, 1999). What makes Froglingo unique is the high-order functions as the sole objects in representing software applications. The uniformness of the managed objects leads to Froglingo’s opportunity of being a monolith in software architecture. Being claimed as a monolith, Froglingo is an off-the-shelf product, i.e., a single executable file, and is self-sufficient in software development and maintenance. Being worth as an alternative, Froglingo is expected to be more productive than the traditional technologies.

In this paper, we analyze the individual components of the traditional technologies, identify how the equivalent functions of the components are supported in Frolingo, and conclude with the feasibility of the monolithic architecture of Froglingo.

To facilitate the discussion in this paper, we briefly introduce Froglingo in Section 2. From Section 3 to Section 7, we discuss the components of the traditional technologies and identify where the equivalent functions of the components go in Froglingo. Through the discussion, we demonstrate that Froglingo is monolithic in system architecture and suggest that it is more productive in software development and maintenance. In the conclusion, we reiterate an objective view on the easiness of computer language to strengthen the suggestion that the monolith Froglingo is more productive.

2 FROGLINGO

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 hierarchical data model are the typical examples. This restriction, however, doesn't reflect the complexities of the real world that are manageable using a computer. The EP (Enterprise-Participant) data model suggests that if an entity is dependent on others, it precisely depends on two other entities. Drawing the terminologies from the structure of an organization or a party, one depended entity was called enterprise (such as organization and party), the other called participant (such as employee and party participant), and the dependent entity called participation. An enterprise consists of multiple participations. Determined by its enterprise and its participant, a participation yields a value, and this value in turn is another enterprise.

The EP data model is the core of Froglingo. It establishes the entire semantic space for practical software applications. Variable is a way of using finite expressions for the (infinite) semantics established in the EP data model. It is intended to be a supplement semantically to the EP data model although it unfortunately brings non-termination processes into Froglingo (Xu et al., 2010).

2.1 EP Data Model

The core concepts are terms, assignment, database, normal form, and reduction.

A *term* is a constant, an identifier, or a pair of parenthesized terms, i.e.,

- If **`T`** is a constant, then **`T`** is a term,
- If **`T`** is an identifier, then **`T`** is a term,
- If **`T1`** and **`T2`** are terms, then (**`T1 T2`**) is a term.

Integers, real numbers, timestamps, and strings are *constants*. In addition, files, as long as not embedding Froglingo expressions, are also constants. For example, `3.14`, `'5/2/2009'`, `"any strings"`, and a file content at operating system level are all constants. *Identifiers* are the tokens to represent high-order functions. The examples are `an_id`, `salary`, `Mike`, and `www.aclient.com`.

A term is used to express data, to embed relationships between data, and to serve as a name in data communications. The examples are `3.14`, `Mike`, `(Mike Salary)`, `((country state) county)`, and `(tax (Mike salary))`.

When a term consists of an ordered pair of two other terms, it is called a combinatory term, abbreviated as comb-term. The first term of a comb-term is called the left-term; and the second term the right-term. For example, the comb-term `(Mike salary)` has `Mike` as the left-term and `salary` as the right-term.

If the right-term of a comb-term is not another comb-term, the parentheses surrounding the term don't have to be written. For example, `((country state) county)` is equivalent to `(country state county)`; and `((a b) (c d))` is equivalent to `(a b (c d))`.

A term can be assigned with a value. An *assignment* is a state that a term takes another term as its value. For example, `(Mike salary) = 2000`, `2 = 3`, and `a = b`. Given an assignment, the term at the left side of the symbol '=' is the assignee; and the term at the right side the assigner (also called value).

A *database* is a finite set of terms and assignments. To make a database meaningful, the terms and the assignments in a database must satisfy the following conditions:

- A constant cannot be an assignee, and cannot be a left-term,
- The right-term of a comb-term appeared in an assignee must not have an assigner; and

- Assignments cannot form a circle, i.e., if there is a sequence of assignments: M_0 = M_1, M_1 = M_2, …, M_{n-1} = M_n, M_n must not be identical to M_0.

As an example, we may have the following database for a school administration:

```
SSD John SSN = 123456789;
SSD John birth = '6/1/1990';
SSD John photo.jpg = …;
```
/* a file is a binary stream*/;
```
College admin (SSD John) enroll =
    '9/1/2008';
College admin (SSD John) Major =
    College CS;
College CS CS100 (College admin (SSD
John)) grade = "F";
```

The *normal form* of a term is the final form, i.e., the value, reducible from the term. An arbitrary term can be *reduced* to its normal form. Here are a few examples of the reduction process:

```
SSD John SSN → 123456789;
College CS CS100 (College admin (SSD
John)) grade → "F";
College CS CS100 → College CS CS100;
College admin (SSD John) Major CS100 →
College CS CS100;
```

In EP data model, we say that a comb-term *functionally depends* on its left-term because the existence of the comb-term depends on the existence of its left-term in a database. Similarly, we say that a comb-term *argumentatively depends* on its right-term. A database is *ordered* as a tree structure either under the functional dependency or under the argumentative dependency.

The EP data model has built-in operators for the dependencies. For example, we have `SSD {=+ SDD`, `SSD John {+ SSD` (or equivalently `SSD John {=+ SSD`), and `SSD John birth {=+ SSD`.

Pre-ordering relations (Xu et al., 2010) are the additional relations existing among high-order functions and lead to the corresponding built-in operators in the EP data model, e.g., `(=+`, `(=-`, and `(=`, which are not further explained here.

2.2 Variables

A variable in Froglingo is represented by an identifier preceded with the symbol $. For example, $a_variable, and $student. A variable is a term too. To be in a database, a variable must satisfy the two conditions:

- 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 is for finite data and the second for infinite data. However, they are managed together, i.e., both are stored physically together in the same data structure; and both can be updated with the same manner. For example, one can issue the expression: `delete fac;`, which removes both assignments from database.

Semantically, the two assignments represent the factorial function and they are equivalent to a database having infinite assignments in the EP data model: `fac 0 = 1; fac 1 = 1; fac 2 = 2; fac 3 = 6; …;`.

A variable can have a range to prevent unwanted data from being the instances of the variable. For example, the factorial function can be redefined to allow integers only being the instances of the variable `$n`:

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

Data, having variables or not, is equally applicable to the built-in operators in Froglingo, and produces the meaningful values according to the semantics of the data. Here is an example: select all the integers that are less than `7` and applicable to the factorial function `fac`:

```
select $x where fac $x != null
              and $x < 7;
```

Please note that variables bring non-termination process to Froglingo. It is users' responsibility to avoid it.

To express the multiple actions triggered by a single event, Froglingo adopts the sequential order of the statements in a traditional programming language, i.e., a statement is not executed until its preceding statement is executed in a procedure of an application program. For example, transferring money between bank accounts is expressed in Froglingo as:

```
transfer $m =
 (update acnt2 = (acnt2 - $m)),
 (update acnt1 = (acnt1 + $m));
```

provided that the two accounts were established earlier: `acnt1 = 100;`, and `acnt2 = 300;`.

3 DBMS AND DATA MODELING

The relational data model and the hierarchical data model, and the conceptual Entity-Relationship model, from which the traditional commercial database management systems (DBMS) are established, are the special cases of the EP data model. In this section, we use examples to demonstrate it.

The employees table in the relational data model:

Employees

ID	Name	salary
1	“Jone”	50000
2	“Mary”	60000

can be presented in Froglingo as:

```
employees 1 name = “Jone”;
employees 1 salary = 50000;
employees 2 name = “Mary”;
employees 2 salary = 60000;
```

The query of finding all the employees whose salary is greater than 55000 is expressed in Froglingo as:

```
select $e name, $e salary where
where $e salary > 55000;
```

A car consisting for its body and its engine (and the engine further consisting for its piston and its cylinder), that is traditonaly managed by using the hierarchical data model, can be expressed in Frogingo as:

```
car body;
car engine piston;
car engine cylinder;
```

The query: retrieve all the parts and assembles under the car is expressed in Frogling:

```
select $p where $p {=+ car;
```

The network-oriented data is the favourite of the conceptual model: Entity-Relationship model. It can also be presented in Froglingo. Given the directed graph: G = {A->B, B->A, B->C, C->D}, as an example, one has a Froglingo presentation:

```
A B = B;
B A = A;
B C = C;
C D = D;
```

A database in the EP data model is a high-order function, and all the total recursive (high-level) functions can be expressed by the EP data model. This is not the case for the relational and the hierarchical models. The relational can only express functions of level 3 (Hillebrand and Kanellakis, 1994); and the hierarchy data model can express only those functions expressed in the EP data model where the right-terms are not comb-term. For example, the term `College admin (SSD John)` cannot be expressed in the hierarchical data model.

4 APPLICATION PROGRAM

The main function of application programs, i.e., application-oriented executable files in a traditional programming language, is to express infinite data in finite expressions. Froglingo has its variables to counter this function as discussed in Section 2.2.

A traditional programming language is also used to express finite data and the queries on the finite data. A typical example is to express the following query: Is there a path from vertices A to vertices D in the directed graph given in Section 3. The EP data model has the following expression for it: `D <=+ A;`.

Placing data access controls and generating web page contents are also application-specific. We will discuss them in Sections 7 and 5 correspondingly and conclude that all of them are managed as data in Froglingo.

Historically, many efforts have been made to couple DBMS and programming together as discussed in Section 2. However, it was concluded that there was a difficulty, called “impedance mismatch”, when one wanted to manage relational data model and programming language together (Ohori et al, 1989). In other words, the clear obstacle is the lower expressive power of traditional data models and the lower productivity of traditional programming languages in representing finite data (Xu et al., 2010). A stored procedure in relational DBMSs appears to be a “monolith” physically. But it retains two exclusive languages, i.e., a data model and a programming language. There are many other attempts for a monolith with high productivity. Without a data model equivalent to a class of total recursive functions, however, none of them can get rid of the issues raised from the traditional technologies. Please reference (Xu et al., 2009) for more discussion on this topic.

5 WEB SERVER

In addition to DBMSs, application programs also need web servers to communicate with web browsers on networks. A web server is to perform the common task of software applications: parsing

HTTP requests and generating HTTP responses. A web server as a off-the-shelf product is at the front-end facing the networks while a DBMS as another off-the-shelf product is at the back-end.

When the functions of the application programs are expressed as data and the physical executable files for the application programs disappear from the architecture of Frogingo, the function of web servers is further supported by Froglingo itself, and therefore the web servers as off-the-shelf products are also eliminated from the architecture of Froglingo . See a graphical view of the evolution in the diagram of Section 8 that compares the software architectures of the traditional technologies and Froglingo.

When we conclude that the application programs and the web servers are eliminated from the architecture of Froglingo, we assume that the function of generating web page contents, that is application-specific, is expressed as data in Froglingo as well. Here is an example to show how Froglingo stores and generates web pages. Froglingo accepts a HTML file (named as `store.html`) embedded with Froglingo expressions:

```
<html>
<body>
   Welcome to my store <br>
   The price of apple is
   <frog>storage apple price</frog>
   <br> The price of milk is
   <frog>storage milk price</frog>
   <br>
</body>
</html>;
```

The contents between the pairs of tags `<frog>` and `</frog>` are Froglingo expressions. When the file is uploaded to Froglingo database, it is stored as a set of assignments:

```
store.html 1 = "<html>
<body>
   Welcome to my store <br>
   The price of apple is ";
store.html 2 = storage apple price;
store.html 3 ="
   <br> The price of milk is ";
store.html 4 = storage milk price;
store.html 5 ="
   <br>
</body>
</html>";
```

When the HTML file is sent to the web clients as the responses, the terms originally between the pairs of tags `<frog>` and `</frog>` are evaluated and replaced with their normal forms.

A HTML file embedded with Froglingo expression in database is uniformly stored as data and can be as complex as a business application needs. For other relevant features including recursively set-oriented query expressions and parameters passing with HTML files, please see the reference (Xu and Zhang, 2010).

6 DATA EXCHANGE AGENT

A typical application software in the traditional technologies stores data in a relational DBMS, processes data with the objects (user-defined data structures) of programming language, and transfers data in a third format (data communication protocol, such as XML). Years ago, the data parsing and conversions between the different data formats were done by writing code as a part of application program. Now, we utilize off-the-shelf data exchange agents (such as Hibernate) to do the common task: message parsing and data conversion based on application-specific conversion rules defined by developers.

Froglingo uses the single format – the EP data model for data storage, data process, and data transformation. Therefore, there is no need to perform data conversion using an agent. Here is a sample data communication in Froglingo by using the built-in operator `print`:

```
Print College →
   College admin (SSD John) enroll =
   '9/1/2008';
   College admin (SSD John) Major =
   College CS;
   College CS CS100 (College admin (SSD
   John)) grade = "F";
```

7 ACCESS CONTROL

An application program, where a relational DBMS is used, also needs to specify data access controls (or called user entitlements) 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 by programming language. This becomes no issue in Froglingo due to its equivalence to a class of total recursive functions. In addition, Froglingo offers a set of built-in operators, stemming from the dependent relationships, to specify data access controls as if a file system specified file access controls.

In an integrated environment such as at a corporate level with multiple applications, an off-the-shelf security product, such as Microsoft Active Directory for Windows-based applications, IBM Tivoli Access Manager or RSA ClearTrust for web-based applications, is utilized to coordinate corporate level security policies. Given the traditional technologies, it simplifies the management of data access controls among the multiple applications by utilizing a centralized database. When the applications in an integrated environment establish a trusted relationship and communicate with each other in Froglingo, the off-the-shelf security products don't have a place in the architecture of Frolingo either.

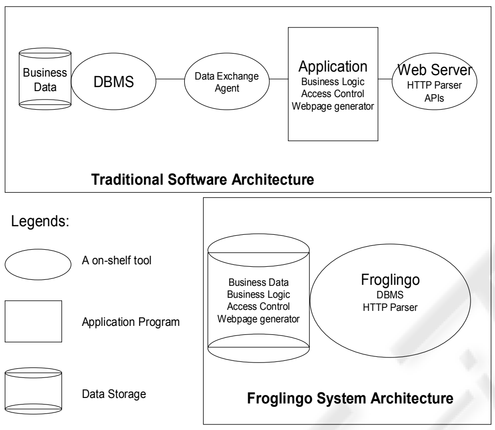

Figure 1.

8 CONCLUSIONS

The functions of the multiple software components in the architecture of the traditional technologies have never been segregated. In addition to the primary copy in DBMS, for example, an employee ID discussed in Section 3 may need to be duplicated and re-processed in almost every other component, i.e., application program, data exchange agent, and data access controls.

Froglingo is a monolith powered by nothing but high-order functions. It is a monolith for every software application. It is a monolith consolidating the multiple components of the traditional technologies.

The consolidation itself improves the productivity of software development and maintenance. In addition, we have concluded in the article (Xu et al., 2010) that Froglingo reaches the greatest possible ease when we assumed 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 those applications where a programming language is involved. The easiness further suggests that Froglingo improves the productivity.

REFERENCES

G. Hillebrand, P. C. Kanellakis, "Functional Database Query Languages as Typed Lambda Calcluli of Fixed Order", ACM SIGMOD/PODS 94.

P. Loucopoulos, K. Lyytinen, K. Liu, T. Gilb, L.A. Maciaszek. "Project Failures: Continuing Challenges for Sustainable Information Systems", Enterprise Information Systems VI, 1-8, 2006 Springer.

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

K. H. Xu, J. Zhang, S. Gao. "High-Ordering Functions and their Ordering relations". The Fifth International Conference on Digital Information Management, 2010.

K. H. Xu, J. Zhang, S. Gao. "An Assessment on the Easiness of Computer Languages". The Journal of Information Technology Review, 2010.

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

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.

K. H. Xu, J. Zhang. "A User's Guide to Froglingo, An alternative to DBMS, Programming language, Web Server, and File System". http://www.froglingo.com/froglingoguide10.pdf, January 2010.

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

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