An Assessment on the Easiness of Computer Languages

Kevin Xu, Jingsong Zhang, Shelby Gao
Bigravity Business Software LLC
{kevin, jingsong, shelby}@froglingo.com

Abstract

Expressive power has been well-established as a dimension of measuring the quality of a computer language. Easiness is another dimension. It is the main stream of development in programming language and database management. In this article, we make the following assessment on the easiness of computer languages: 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; and 3) A unification, i.e., a system having both a data model semantically equivalent to a class of total recursive functions and a programming language, achieves the greatest possible ease. We materialize the assessment with the following facts: 1) A data model is semantically within a class of total recursive functions while a programming is semantically within a class of partial recursive functions; 2) A data model arranges the managed objects in orders including at least one dependent order; and 3) The objects represented for both business data and business logic, in the case of a unification, obey the orders enforced by the data model.

1. Introduction

Many database applications were written in programming language in 1960s and 1970s and they are currently still in operation. Database management system (DBMS) came to the field of database application software in 1970s. It significantly improved the productivity in the development and maintenance of database applications. However, programming language and DBMS must be employed together for a database application because of the limitation of DBMS.

Froglingo is a unified solution for information management, an alternative to the combination of programming language, DBMS, file system, and web server. It is a "database management system (DBMS)" to store and to query business data; a "file system" to store and to share files; a "programming language" to support business logic; and a "web server" to interact with users across networks. More than the combination of the existing technologies, it is a single language uniformly representing both data and application logic. See more information about Froglingo in [7].

Assessing a language's easiness is generally considered subjective. Froglingo, however, suggested how one assesses easiness more objectively. The authors in [6] made a case for this view. It first assumed that a data model is easier to use than a programming language in the development and maintenance of those applications expressible in the data model. It had a second assumption that 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. With the two assumptions above, it was concluded that the easiness reached the upper limit mathematically when a unification, i.e., a system having both a data model semantically equivalent to a class of total recursive functions and a programming language.

In this paper, we expand the discussion in the article [6] and emphasize the following facts that materialize the assumptions and the conclusion made in the article [6]:

1. A data model is semantically within a class of total recursive functions. It guarantees that a software application in the data model terminates unconditionally.
2. A data model has a set of ordering relations. It guarantees that set-oriented queries and update operations can be applied to the business data managed in the data model
3. The objects represented for both business data and business logic in a unification obey the ordering relations enforced by the data model. It allows both business data and business logic to be managed equally, and therefore consolidates the multi-component architecture of the software applications in current technologies.

By recalling the history of the progress in the fields programming language and database management through Sections 2 to 5, we identify the essences of a data model and quantitatively differentiate it from a programming language. The outcome of the analysis is the first two facts listed above, which justifies the two assumptions given in the article [6].

With the two justified assumptions, it is clear that a unification achieves the greatest possible ease. In Section 6, we outline what is meant practically by the greatest possible ease.

2. Language

A language has its syntax and its semantics. Semantics characterizes an aspect of the quality of a language, i.e., the effect or outcome of the programs written in language. It has been well studied and is quantitatively measurable by expressive power.

Syntax characterizes another aspect of the quality, i.e., the effort of writing the programs in a language. In this article, we call it easiness. When two languages have the same expressive power, users may prefer to use one over another. There hasn't been a precise measurement on easiness. We raise the issue here because it has been a factor in the evolution of computer languages.

Two languages can be comparable in easiness only if they are equivalent in terms of expressive power. It is meaningless to compare two languages in easiness if they represent two exclusive sets of semantics. However, if the semantics of one language is a superset of the semantics of a second language, the easiness of the first language can be compared with the easiness of the second when that first language is used to express the semantics of the second.

3. Programming Language

A programming language is a language with the semantics equivalent to a class of computable (or partial recursive) functions, which determines the upper limit of what a computer can handle.

Programming languages have evolved from low-level machine/assembly languages to higher-level languages. One aspect of programming languages that remains unchanged is their expressive power, the Turing equivalent.

What, then, has changed in the process of programming language evolution? What has changed is the easiness. Driven by the need for programmers to be ever more productive in software development and maintenance, we have worked continuously to produce easier programming languages.

We note a key difference between a programming language and a data model: A programming language can represent computable functions, and therefore infinite data, in expressions that are themselves finite.

The semantics of programming languages inevitably falls into a class of partial recursive functions. However, we strive hard to design and to use programming languages so that the programs written in the programming languages can eventually fall into a narrower class of total recursive functions. The class of total recursive functions is all those useful in representing software applications given the limitations of a computer.

4. Data Model

A data model is a language. As far as the existing data models and the objective of this article are concerned, we contend that the semantics of a data model is a subset of a class of total recursive functions. A data model is the mathematical abstraction of a database management system (DBMS) for database applications.

A data model normally refers to a data structure that stores a set of data and a set of built-in operators under which that set is closed. (By closed, we mean that an operation always terminates and returns members from the set.) To emphasize the dominant role of data structure, we say that a data model is a data structure which arranges a set of objects in orders including at least one dependent order. The second definition is intended to be equivalent to the first one, while the orders attend the built-in operators. For a precise definition of the concept of data model in mathematical terms, please reference [5].

When the objects in a set are arranged in orders (or called ordering relations), e.g., sequential order, tree-structured order, partial order, and preorder, one can express set-oriented operations against the objects. This is one of the essences of a database over programming language.

A dependent order says that if one object depends on another in a set, and that first object is in the set, then the second must be in the set as well. The dependent order is bi-conditional, so conversely, if the second object is not in the set, then the first object cannot be there either. (e.g., an attribute in a relational table depends on its row; a child object depends on its parent object in hierarchy; and the birth of an infant depends on both its mother and father.) Requiring a data model to have at least one dependent order is to reflect the fact that a set of objects can be added or removed from a database by using a built-in operator.

The orders among the objects in a set must also be decidable. In other words, one is always able to tell, in finite steps, if one object in a set is related to another object in terms of the orders. This requirement is to reflect the other essence of a database over programming language, i.e., the built-in operators in a data model always terminate. Given a dependent order, for example, one can always tell if one object is dependent on another object or not. A dependent order disallows an object to depend on itself (i.e., data that is organized to have a cyclical loop) in the managed sets. Such a requirement disqualifies a computer language as a data model if there is a program in the language that doesn't terminate on an input.

With the definition above, we say that queue in programming language is an example of data models. So are the relational data model and the hierarchical data model with the containment relationship. The traditionally called "network data model" that allows cyclical data and not clearly defining the dependencies among cyclical data is

excluded from being a data model in this paper. Obviously, both Datalog and a programming language are not data models.

Requiring a data model to have decidable orders justified the assumptions used in article [6].

A data model is allowed to have infinite and countable objects. We will see in Section 6 that there is a data model that can "store" an entire class of total recursive functions.

5. Hybrid

Programming languages define functions by coding algorithms; data models define functions by enumerating properties. One might say that although a data model is preferable, a programming language is inevitable.

There are several reasons for this. First of all, a lot of business data falling into a class of total recursive functions may be desired to be, but not expressible in a traditional data model. (By not expressible, we mean that some dependencies would be lost even if the data were placed, that is to say decomposed, into the data structure of a data model.) Hierarchical data, as a typical example, can be folded into a table, but its containment relationships cannot be captured by the relational data model. Another example would be the relationships among the vertices in a directed graph (e.g., is there a path from A to B), which cannot be captured in both relational data model and hierarchical data model.

Secondly, constructing arbitrary queries on the top of a managed data set requires a programming language. Although built-in operators (such as SQL in relational data model) can be used to construct a class of useful queries, they don't exhaust all the queries that are practicality required and within a class of total recursive functions. For example, a query in the relational data model cannot simply return a single attribute or a sequence of attributes out of a relational database. There is no exception to this. This holds true even for a data model equivalent to a class of total recursively functions.

A system having both a programming language and a data model is called a hybrid. Hybrids started with the research efforts into "database programming language" in 1970s to the early 1990s. They offered programming languages on the top of relational data models, hierarchical data models, and network structures (called network data models at that time). Some proposals were Galileo, Functional Object Language, Machiavelli, PFL, BULK, and XML/XQUERY. The approach, the combined relational data model and programming language, is the most popular hybrid today. Due to the lower expressive power of the underlying data models, however, this approach didn't start from a well-established foundation.

In database applications, a hybrid is easier than a stand-alone programming language. A hybrid is easier because a data model is used for a part of database application. A hybrid is easier than another hybrid if the data model of the first hybrid is semantically a superset of the data model of the second hybrid. It is not meaningful to compare the easiness of hybrids based on relational and hierarchical data models because their semantics overlap, and are not inclusive.

6. Unification

A hybrid becomes a unification when its data model is semantically equivalent to a class of total recursive functions. Mathematically, this means that the data model could represent arbitrary software applications without a programming language, provided that space was unlimited.

We conclude, given the understanding of easiness that we have established, that the easiest hybrid to use is a unification.

What are implied practically by the easiest in a unification? First of all, the data model of a unification is a consistent tool to construct as much finite data as an application needs without a possibility of an exception. This is significant because one no longer needs to specify user-defined data types (or object classes) for applications. One uses the data model of a unification for an application as if he or she used SQL for table constructions; or used a queue data structure for message queues.

Secondly, the data model of a unification has a rich set of ordering relations due to the inherent structure of (total) high-level functions. The ordering relations lead to a set of built-in operators that are much more expressive than those in the relational data model and hierarchical data model. Therefore, many queries that challenge traditional technologies can be simply expressed in the data model of a unification. For example, the query: "if there is a path from `A` to `B` in a directed graph" can be simply expressed as `B (=+ A` in the EP data model. For an extensive discussion on the ordering relations, readers may reference the article [2].

Thirdly, a unification is monolithic in software architecture, and therefore it consolidates the software architecture of traditional technologies with multiple components.

In traditional technologies, a data model cannot replace programming language due to its low expressive power; and a programming language cannot replace a data model due to its less easiness. This determines the separation of the business data in DBMS from the business logic in programming language, and this led to the "impedance of mismatch" concluded from the research effort of database programming language [1].

A unification, however, has the unique opportunity to eliminate the separation because its data model has a semantic space, i.e., a class of total recursive functions, to accommodate all the practically meaningful applications. Although the data model itself can not practically fill up the entire (infinite) semantics, additional constructors, e.g., variables from programming language, can do so. As a result, the ordering relations are preserved for the objects representing business logic; and therefore the corresponding built-in (both query and update) operators are equally applicable to both business data and business logic. Froglingo is such a unification. The addition of variables, that constitutes Froglingo as a Turing equivalent language, allows users to express business logic along with business data. It semantically stuffs the (infinite) semantic space of the EP data model by finite expressions.

Note that the addition of variables also brings in non-termination processes. It is users' responsibility to avoid them. For an extensive discussion on the consolidations of software architecture, readers may reference the article [4].

7. Summary

In this article, we justified the rationale that a data model is preferred in software development over a programming language, and concluded that the more expressive power a data model has, the easier 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 the applications, it is concluded in this article that a unification achieves the greatest possible ease. The conclusion has the following implications in practice: 1) A unification is a consistent tool for the data construction of arbitrary applications; 2) A unification has more expressive built-in operators for queries; and 3) A unification consolidates the software architecture of traditional technologies.

Reference:

1 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.

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

3 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.

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

5 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.

6 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.

7 K. H. Xu, J. Zhang. "A User's Guide to Froglingo, Database Application Management System". To appear on the website: http://www.froglingo.com.