Computer Languages, Systems & Structures 38 (2012) 214–241

Contents lists available at SciVerse ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Core FOBS: A hybrid functional and object-oriented language

James Gil de Lamadrid [a,*], Jill Zimmerman [b]

[a] Bowie State University, Bowie, MD 20715, United States
[b] Goucher College, Baltimore, MD 21204, United States

ARTICLE INFO

Article history:
Received 31 January 2011
Received in revised form
12 January 2012
Accepted 22 April 2012
Available online 29 May 2012

Keywords:
Object-oriented
Functional
Hybrid

ABSTRACT

We describe a computer language that is a hybrid between functional and object-oriented languages. The language is based on a simple structure called a FOB (functional-object), capable of being used as a function, or accessed as an object. FOBS is a dynamically typed, referentially transparent language, designed for use as a universal scripting language. An extensive library is integral to the language. The library implements the primitive types and provides an interface to the external environment, allowing scripting actions to be carried out.

FOBS is structured as an extended language, that is reduced to a core language by macro expansion. Our paper provides an introduction to the core language, a brief discussion of the extended language, and formal specifications of syntax and semantics for the core. The formal semantic description for FOBS is somewhat unusual for a scripting language. However, this description ensures that the FOBS semantics is well-specified, allowing programmers to write well understood programs, increasing program reliability.

1. Introduction

The object-oriented programming paradigm and the functional paradigm both offer valuable tools to the programmer. Many problems lend themselves to elegant functional solutions. Others are better expressed in terms of communicating objects.

A single language with the expressive power of both paradigms allows the user to tackle both types of problems, with fluency in only one language. Many languages have both object oriented features, and functional features. For example, a language like Java, which is object-oriented, allows the user to write functions with recursion. If the programmer restricts their programming style, they can produce programs that are in line with programs produced using pure functional languages.

FOBS differs from other languages that support both functional and object-oriented features. These languages, are mostly centered around object-orientation. FOBS has a distinctly functional flavor. This kinship with the functional paradigm is further discussed in Section 3.

In particular, FOBS is characterized by the following features:

- A single, simple, elegant structure called a FOB, that functions both as a function and an object.
- A stateless runtime environment. In this environment, mutable objects are not allowed. Mutation is accomplished, as in functional languages, by the creation of new objects with the required changes.

* Corresponding author. Tel.: +1 3018603968.
E-mail address: jgildelamadrid@bowiestate.edu (J. Gil de Lamadrid).

1477-8424/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cl.2012.04.002

- A simple form of inheritance. A *sub-FOB* is built from another *super-FOB*, inheriting all attributes from the super-FOB in the process.
- A form of dynamic scoping to support attribute overriding in inheritance. This allows a sub-FOB to replace data or behaviors inherited from a super-FOB.
- A single data type; the FOB. All of the usual primitive data types, such as integer, real, and boolean are implemented as FOBs defined in the system library.

2. FOBS design overview

In designing FOBS several factors were taken into consideration. Our goal was to design a language that introduced objects into a functional language without destroying the functional property of referential transparency exhibited by stateless environments. In addition, it was important that the resulting language be concise in syntax and semantics. This requirement ensures that a thorough formal definition of the language can easily be provided, and that a simple interpreter for the language can be constructed. The result is a light-weight language, that can be confidently used, knowing that it adheres unfailingly to a standard definition.

Borrowing from languages like Smalltalk [10], in which everything is an object, we sought a single homogeneous structure expressive enough to represent all necessary data objects. The FOB, described further in Section 5, is this single data type that exhibits behaviors of both an object and a function. This characteristic of FOBS, makes it clear that FOBS is not a language with a strong system of types, and type checking. In fact, the single homogenous type limits the amount of static type checking that can be performed. The result is that FOBS, like Smalltalk, is dynamically typed.

In scripting languages like FOBS, the flexibility of dynamic typing usually outweighs the safety benefits of static typing. In these languages, where there is no separate compilation phase, we feel that performing static error checking is less important. Any information that would have been discovered at compile time can easily be discovered by an operator at run time, and an error message specific to that problem can be issued by the operator. Also, dynamic typing has its advantages. In particular it relieves the user from the type definitions, and the constraints of typed variable use.

Although a simple language is beneficial to the language implementer, it is not necessarily beneficial to the user, who may be more interested in a large set of features to choose from to easily craft their code. Considering this, we decided to allow the user the flexibility to change the syntax of the language, tailoring it to their own preferences. Extensions to the core language can be defined using a fairly sophisticated macro expansion system. This feature is covered in more detail in Section 10.

Macro expansion is one of several options for extending a language. Other common options include template meta-programming (TMP), and staged compilation. TMP and staged compilation both have as an advantage the use of semantic information in the specification of transformations, whereas macros are mostly limited to syntactic transformation. In the case of FOBS, there was a concern, with extension transformation available to the user, that the fundamental semantics of the language might be altered. In this light the limited nature of macro expansion seems a better fit to FOBS.

One of the major problems faced in the implementation of FOBS was scoping rules. Purely lexical scope, which is often used in modern functional languages, does not take into account the dynamic elements needed in object-oriented dynamic message binding. This problem, and the solution are discussed further in Section 7.3. The solution in FOBS is a hybrid dynamic-static scoping rule that searches dynamically through an inheritance hierarchy, and also statically through a set of nested block structures.

3. Language features

Probably the characterizing feature of functional languages is referential transparency. In many respects, the referential transparent character of FOBS places it closer to the functional paradigm, rather than to the procedural aspects of most common object-oriented languages. FOBS partially shares several other features with functional languages.

In Section 15 the formal semantic description of FOBS presents a semantic model that is clearly stateless, and as such is referentially transparent. This is a strength of the FOBS language, and other referentially transparent languages. It has been argued that the transparency of the code produces code that is more reliable. It also leads to code that can be more easily automatically separated into separate processes and distributed, allowing a large degree of parallelism. Referential transparency also implies a smaller dependency on a particular platform, allowing code to be more easily ported from one machine to another.

The ability to define higher-order functions is considered as a strength in functional languages. It allows a programmer to abstract out useful computational patterns. In FOBS the equivalent is to write a high-order FOB, meaning a FOB that takes another FOB as a parameter, or returns a FOB as its result. This is trivially possible in FOBS, since everything is a FOB, and no distinction is made between a primitive FOB, or a user defined FOB.

Automatic currying is another strength of many functional languages. It allows a programmer to create partially applied functions that can be completely customized at a later time. In currying, the under-application of a function results in a return value which is a function with the remaining parameters still unbound. In FOBS the under-application of a FOB also results in a FOB with the remaining parameters unbound. The mechanism used, however differs substantially from