FEDELE: A Mechanism for Exending the Syntax and Semantics for the Hybrid Functional-Object-Oriented Scripting Language FOBS

James Gil de Lamadrid

Bowie State University, 14000 Jericho Pk Rd, Bowie, MD. 20715
jgildelamadrid@bowiestate.edu

Abstract

A language FOBS-X (Extensible FOBS) is described. This language is an interpreted language, intended as a universal scripting language. An interesting feature of the language is its ability to be extended, allowing it to be adapted to new scripting environments. The interpretation process is structured as a core-language parser back-end, and a macro processor front-end. The macro processor allows the language syntax to be modified. A configurable library is used to help modify the semantics of the language, allowing the addition of the required capabilities for interacting in a new scripting environment. This paper focuses on the semantic extension of the language. A tool called FEDELE has been developed, allowing the user to add library modules to the FOBS-X library. In this way the semantics of the language can be enhanced, and the language can be adapted to new scripting environments.

Keywords functional, object-oriented, programming language

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. FOBS-X is a single language with the expressive power of both paradigms allowing the user to tackle both types of problems, with fluency in only one language. FOBS-X is a modification to the FOBS language described in Gil de Lamadrid & Zimmerman [4]. The modification involves simplifications to the pointers used in the scoping rules.

FOBS-X has a distinctly functional flavor. In particular, it is characterized by the following features:

- A single, simple, elegant data type called a FOB, that functions both as a function and an object.
- Stateless programming. In the runtime environment, mutable objects are not allowed. Mutation is accomplished, as in functional languages, by the creation of new objects with the required changes.
- 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 scoping that supports attribute overriding in inheritance. This allows a sub-FOB to replace data or behaviors inherited from a super-FOB.
- A macro expansion capability, enabling the user to introduce new syntax.
- A tool for easily writing new library modules, allowing the semantics of FOBS-X to be modified to fit differing scripting requirements.

As with many scripting languages FOBS is weakly typed, a condition necessitated by the fact that it only has one data type. However, with interpreted languages the line between parsing and execution is more blurred than with compiled languages, and the necessity to perform extensive type checking before execution becomes less important.

Several researchers have built hybrid language systems, in an attempt to combine the functional and object-oriented paradigms, but have sacrificed referential transparency in the process. Yau et al. [11] present a language called PROOF. PROOF tries to fit objects into the functional paradigm with little modification to take into account the functional programming style. The language D by Alexandrescu [1] is a rework of the language C transforming it into a more natural scripting language similar to Ruby and Javascript.

Scala by Odersky et al. [12] is a language compiled to the Java Virtual Machine, which claims to implement a hybrid of functional and object-oriented paradigms, but tends toward the imperative language end of the spectrum. A class based language that is proposed as a tool to write web-servers, Scala is implemented as a small core language, and many of its capabilities are implemented in the library. FOBS has this same structure, allowing the capabilities of the language to be easily extended.

Two languages that seek to preserve functional features are FLC by Beaven et al. [2], and FOOPS by Goguen and Mesegner [6]. FOOPS is built around the addition of ADTs to functional features. We feel that the approach of FLC is conceptually simpler. In FLC, classes are represented as functions. This is the basis for FOBS also. In FOBS we have, however, removed the concept of the class. In a stateless environment, the job of the class as a "factory" of individual objects, each with their own state, is not applicable. In stateless systems a class of similar objects is better represented as a single prototype object that can be copied with slight modifications to produce variants.

IFL '14, October 1–3, 2014, Boston, MA, USA.
Copyright © ACM [to be supplied]. . . $15.00.
http://dx.doi.org/10.1145/

Another language that implements object-orientation while maintaining a mostly functional approach is OCAML[8]. Built around ML, OCAML has added elements enabling imperative and object orient programming. A record structure supports the creation of objects, and mutable objects support stateful programming. Later in the paper we discuss the importance of mutation in object-orientation. And, although important, we felt that mutation should be isolated and controlled, to help preserve the overriding computation model of FOBS, which prominently features referential transparency. In OCAML, mutable objects are tightly integrated into the computational model, giving it a distinct non-declarative nature.

Scripting languages have tended to shy away from the functional paradigm. Several object-oriented scripting languages such as Python [3] are available. Although mostly object-oriented, its support for functional programming is decent, and includes LISP characteristics such as anonymous functions and dynamic typing. However, Python lacks referential transparency. We consider this as one of the important advantages of FOBS. In the design of FOBS, we also felt that a simpler data structure could be used to implement objects and the inheritance concept, than was used in this popular language. FOBS combines object orientation and functional programming into one elegant hybrid, making both tools available to the user. Unlike languages like Python or FOOPS, this is not done by adding in features from both paradigms, but rather by searching for a single structure that embodies both paradigms, and unifies them.

2. Language Description

FOBS-X is built around a core language, core-FOBS-X. Core-FOBS-X has only one type of data: the FOB. A simple FOB is a quadruplet,

```
[m i -> e ^ ρ]
```

The FOB has two tasks. Its first task is to bind an identifier, i, to an expression, e. The e-expression is unevaluated until the identifier is accessed. Its second task is to supply a return value when invoked as a function. ρ (the ρ-expression) is an unevaluated expression that is evaluated and returned upon invocation.

The FOB also includes a modifier, m. This modifier indicates the visibility of the identifier. The possible values are: "'+", indicating public access, "'~", indicating protected access, and "'$", indicating argument access. Identifiers that are protected are visible only in the FOB, or any FOB inheriting from it. An argument identifier is one that will be used as a formal argument, when the FOB is invoked as a function. All argument identifiers are also accessible as public.

As an example, the FOB

```
[‘+x -> 3 ^ 6]
```

is a FOB that binds the variable x to the value 3. The variable x is considered to be public, and if the FOB is used as a function, it will return the value 6.

Primitive data is defined in the FOBS library. The types *Boolean*, *Char*, *Real*, and *String* have constants with forms close to their equivalent *C* types. The *Vector* type is a container type, with constants of a form close to that of the ML list. For example, the vector

```
["abc", 3, true]
```

represents an ordered list of a string, an integer, and a Boolean value. Semantically, a vector is more like the Java type of the same name. It can be accessed as a standard list, using the usual car, cdr, and cons operations, or as an array using indexes. It is implemented as a Perl list structure. Unlike the Java type, the FOBS-X type is immutable. The best approximation to the mutate operation is the creation of a brand new modified vector.

There are three operations that can be performed on any FOB. These are called *access*, *invoke*, and *combine*. An access operation accesses a variable inside a FOB, provided that the variable has been given a public or argument modifier. As an example, in the expression

```
[‘+x -> 3 ^ 6].x
```

the operator "." indicates an access, and is followed by the identifier being accessed. The expression would evaluate to the value of x, which is 3.

An invoke operation invokes a FOB as a function, and is indicated by writing two adjacent FOBs. In the following example

```
[‘$y -> _ ^ y.+[1]] [3]
```

a FOB is defined that binds the variable y to the empty FOB and returns the result of the expression $y + 1$, when used as a function. When the example is used as a function, since y is an argument variable, the binding of the variable y to the empty FOB is considered only a default binding. This binding is replaced by a binding to the actual argument, 3. To do the addition, y is accessed for the FOB bound to the identifier +, and this FOB is invoked with 1 as its actual argument. The result of the invocation is 4.

In an invocation, it is assumed that the second operand is a vector. This explains why the second operand in the above example is enclosed in square braces. Invocation involves binding the actual argument to the argument variable in the FOB, and then evaluating the ρ-expression, giving the return value.

A combine operation is indicated with the operator ";". It is used to implement inheritance. In the following example

```
[‘+x -> 3 ^ _] ; [‘$y -> _ ^ x.+[y]]                (1)
```

two FOBs are combined. The super-FOB defines a public variable x. The sub-FOB defines an argument variable y, and a ρ-expression. Notice that the sub-FOB has unrestricted access to the super-FOB, and is allowed access to the variable x, whether modified as public, argument or protected.

The FOB resulting from Expression (1) can be accessed, invoked, or further combined. For example the code

```
([‘+x -> 3 ^ _] ; [‘$y -> _ ^ x.+[y]]).x
```

evaluates to 3, and the code

```
([‘+x -> 3 ^ _] ; [‘$y -> _ ^ x.+[y]]) [5]
```

evaluates to 8.

Multiple combine operations result in FOB stacks, which are compound FOBs. For example, the following code creates a FOB with an attribute x and a two argument function that multiplies its arguments together. The code then uses the FOB to multiply 9 by 2.

```
([‘+x -> 5 ^ _] ; [‘$a -> _ ^ _] ;
 [‘$b -> _ ^ a.*[b]]) [9, 2]
```

In the invocation, the arguments are substituted in the order from top to bottom of the FOB stack, so that the formal argument a would be bound to the actual argument 2, and the formal argument b would be bound to 9.

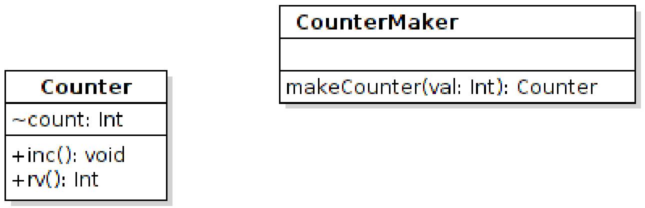

Figure 1. Class structure of Example (2)

In addition to the three FOBS operations, many operations on primitive data are defined in the FOBS library. These operations include the usual arithmetic, logic, and string manipulation operations. In addition, conversion functions provide conversion from one primitive type to another, when appropriate.

Example (2) presents a larger example to demonstrate how FOBS code might be used to solve more complex programming problems. In this example we define a FOB that implements a standard up-counter. The FOB structure is shown in Figure 1, using UML. The outermost FOB implements the UML class called *CounterMaker*, that copies a prototype to create new counters. The counters are known as the class *Counter* in Figure 1. *CounterMaker* creates a new *Counter* when its function *makeCounter* is called. The argument to *makeCounter*, *val*, becomes the initial value of the counter. The counter contains an instance variable, *count*, that contains the current count value, and a function *inc* that ”increments” the counter. Since FOBS is stateless, what *inc* actually does is create a new *Counter* object with the incremented *count* variable.

```
## Implementation of a standard up-counter
([‘+makeCounter ->
     [‘$val -> 0 ^
          [‘~count -> val ^_];
          [‘+inc ->
               [‘~_ -> _^ makeCounter[
                    count.+[1]]]
           ^_];                                    (2)
          [‘~_ -> _ ^ count]
     ]
^_]
## test it
     .makeCounter[6].inc[].inc[])[]
#.
#!
```

Since UML is designed to model object-oriented systems, it is no surprise that using it to model a FOB requires extra notation to handle the ability to invoke a FOB as a function. In Figure 1. the notation *rv* is used to represent the operation of invoking the FOB as a function. The use of *rv* (return value) in the diagram indicates that, when the FOB Counter is invoked, it returns the current value of the variable *count*.

Larger examples, and a more complete definition of the FOBS language are given by Gil de Lamadrid and Zimmerman [4].

3. Core-FOBS Design Topics

Expression evaluation in FOBS-X is fairly straight forward. Three issues, however, need some clarification. These issues are: the semantics of the redefinition of a variable, the semantics of a FOB invocation, and the interaction between dynamic and static scoping.

3.1 Variable overriding

A FOB stack may contain several definitions of the same identifier, resulting in overriding. For example, in the following FOB

```
[‘$m -> ’a’ ^ m.toInt[]] ; [‘+m -> 3 ^ m]
```

the variable m has two definitions; in the super-FOB it is defined as an argument variable, and in the sub-FOB another definition is stacked on top with m defined as a public variable. The consequence of stacking on a new variable definition is that it completely overrides any definition of the same variable already in the FOB stack, including the modifier. In addition, the new return value becomes the return value at the top of the full FOB stack.

3.2 Argument substitution

As mentioned earlier, the invoke operator creates bindings between formal and actual arguments, and then evaluates the ρ-expression of the FOB being invoked. At this point we give a more detailed description of the process.

Consider the following FOB that adds together two arguments, and is being invoked with values 10 and 6.

```
([‘$r -> 5 ^ _] ; [‘$s -> 3 ^ r.+[s]]) [10, 6]
```

The result of this invocation is the creation of the following FOB stack

```
[‘$r -> 5 ^ _] ;
[‘$s -> 3 ^ r.+[s]] ;
[‘+r -> 6 ^ r.+[s]] ;
[‘+s -> 10 ^ r.+[s]]
```

In this new FOB the formal arguments are now public variables bound to the actual arguments, and the return value of the invoked FOB has been copied up to the top of the FOB stack. The return value of the original FOB can now be computed easily with this new FOB by doing a standard evaluation of its ρ-expression, yielding a value of 16.

3.3 Variable scope, and expression evaluation

Scoping rules in FOBS-X are, by nature, more complex than scoping used in most functional languages. Newer functional languages, such as Haskell and ML, typically use lexical scoping. Dynamic scoping is often associated with older dialects of LISP.

Pure lexical scoping does not cope well with variable overriding, as understood in the object-oriented sense, which typically involves dynamic message binding. To address this issue, FOBS-X uses a hybrid scoping system which combines lexical and dynamic scoping. Consider the following FOB expression.

```
[‘~y -> 1^_] ;
[‘~x ->
     [‘+n -> y.+[m] ^ n] ;
     [‘~m -> 2 ^_]                       (3)
^_] ;
[‘~z -> 3 ^x.n]
```

This expression defines a FOB stack that is three deep, containing declarations for a protected variable y, with value 1, a protected variable x with a FOB stack as its value, and a protected variable z with the value 3 as its value. The stack that is the value of x consists of two FOBs, one defining a public variable n, and one defining a protected variable m.

We are currently mostly interested in the FOB stack structure of Expression (3), and can represent it graphically with the stack graph, given in Figure 2. In the stack graph each node represents a simple FOB, and is labeled with the variable defined in the FOB.

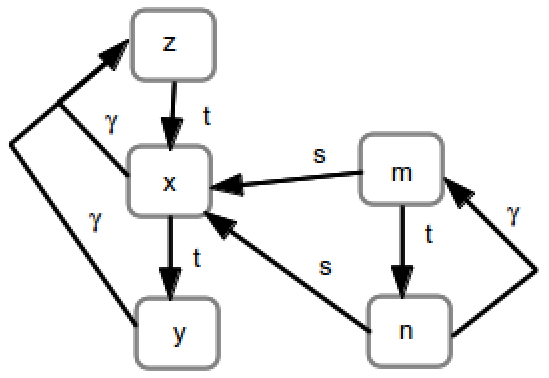

Figure 2. Stack graph of Example (3)

Three types of edges are used to connect nodes: the s-pointer, the t-pointer, and the γ-pointer. The s-pointer describes the lexical nested block structure of one FOB defined inside of another. The s-pointer for each node points to the FOB in which it is defined. For example m is defined inside of the FOB x.

The t-pointer for each node points to the super-FOB of a FOB. It describes the FOB stack structure of the graph. In Figure 2 there are basically two stacks: the top level stack consisting of nodes z, x, and y, and the nested stack consisting of nodes m, and n.

The γ-pointer is a back pointer, that points up the FOB stack to the top. This provides an easy efficient mechanism for finding the top of a stack from any of the nodes in the stack.

If the FOB z were invoked, it would access the FOB n for the value of n. This would cause the expression $y+m$ to be evaluated, a process that demonstrates the use of all three pointers. The process of resolving a reference in FOBS-X first examines the current FOB stack. The top of the current stack is reached by following the γ-pointer. Then the t-pointers are used to search the stack from top to bottom. If the reference is still unresolved, the s-pointer is used to find the FOB stack enclosing the current stack. This enclosing stack now becomes the current stack, and is now searched in the same fashion, from top to bottom, using the γ-pointer to find the top of the stack, and the t-pointers to descend to the bottom.

To summarize this procedure for the example, to locate the definition of the variable y, referenced in the FOB n, the γ-pointer for n is followed up to the FOB m, this FOB is examined, and then its t-pointer is followed down to the FOB n, which is also examined. Not having found a definition for the variable y, the s-point for FOB n is followed out to the FOB x, and then the γ-pointer is followed up to the FOB z. FOB z is examined, and its t-pointer is traversed to FOB x, which is also examined. Then the t-point for FOB x is finally followed down to the FOB y, which supplies the definition of y needed in the FOB n.

As mentioned above, the scoping for FOBS-X is a combination of lexical and dynamic scoping. S-pointers are lexical in nature, since the nesting of FOBs is a static property. T-pointers and γ-pointers are dynamic. These pointers must be created as new FOB stacks are created during execution.

4. The FOBS Library

As FOBS-X can be extended by adding new primitive FOBs to the library, we use the term *native primitive FOBs* to denote the primitive FOBs that are part of core-FOBS. The FOBS library contains

Libary FOB	Operation	Description
`Boolean`	`b.if[x, y]`	If Boolean value b is `true`, return x, otherwise return y
	`b.&[x]`	Return the boolean value of the expression $b \bigwedge x$
	`b.|[x]`	Return the boolean value of the expression $b \bigvee x$
	`b.![]`	Return the boolean value of the expression $\neg b$

Table 1. Operations for the *Boolean* FOB

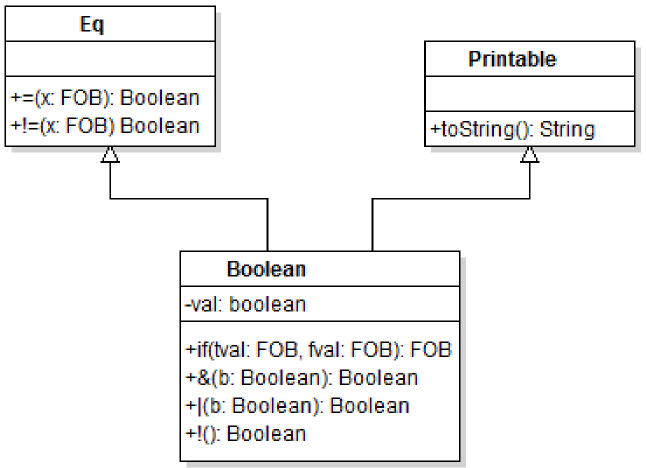

Figure 3. Interface for the *Boolean* FOB

definitions of all native primitive FOBs. The native primitive FOBs are *Int*, *Char*, *Real*, *Boolean*, *Vector*, *String*, and *FOBS*. In addition a set of "mix-in" FOBs are contained in the library, that serve the same purpose as mix-in classes described by Page-Jones [9], providing general capabilities to primitive FOBs. For example the *Boolean* FOB uses the mix-in FOBs *Eq*, and *Printable* to supply operations to compare Boolean values for equality, and the ability to be printed, respectively.

The native primitive FOBs mostly implement the native data types of the FOBS language. Each data type provides the wrapper for the data, along with a set of operations, used to manipulate the data. As an example, Table 1. shows the operations provided by the *Boolean* FOB. This operation structure is shown in the UML diagram of Figure 3. The operations for the *Boolean* FOB are implication, logical *and*, logical *or*, and logical *not*. The *Boolean* FOB inherits the operations of *equals*, and *not-equals* form the mix-in FOB *Eq*, and it inherits the *toString* function, that generates a print-string, from the FOB *Printable*.

The primitive FOB *FOBS* is the one primitive FOB that does not implement a data type. This FOB is, initially, largely empty. It, however, provides the mechanism for extending the FOBS-X language, allowing it to be adapted to differing scripting environments. The user of the FOBS-X language extends the language by adding modules to the *FOBS* FOB, one for each extension to the language.

5. Extensions

FOBS is a language that is designed to be extensible, both in terms of syntax, and semantics. To extend the language the user designs an *extension*. An extension is defined by an *extension module*, which is composed of two pieces: a macro file, and a collection of library modules.

5.1 Macro files

FOBS-X allows the syntax of the language to be changed in a limited fashion. The mechanism used to modify the syntax is macro expansion. Before a FOBS expression is parsed, a macro processor is used to expand macros used in the code. In this way, the user can alter the syntax of FOBS expression by writing and loading the appropriate macros to handle the changes.

Many programming languages have macro capabilities. These range from the fairly simple mechanisms in the programming language *C*, to the the relatively more sophisticated mechanisms of LISP. It was felt that these simple systems were inadequate for FOBS. In particular, to implement a fair degree of flexibility, we felt that the ability to modify syntax should be more extensive than these types of systems offer, including a limited ability to change delimiter symbols. The language MetaML [13] provides much more sophisticated macro capabilities. It is built for the manipulation of macro type code, and implements multi-stage meta-programming. The macro capability of FOBS is much lighter weight than that of MetaML, but ideas from MetaML have found their way into FOBS-X. In particular, we found the staging of macro expansion useful. The staging in our case is used to implement macro operator precedence.

Macro definitions are quadruples, which are described in detail by Gil de Lamadrd [5]. Example (4) gives a simple demonstration the form of macro definitions.

```
## the array mutate operation
#defleft
     #?x [ #*i ] <- #?y
#as                                           (4)
     ( #?x ) . -+ [ #*i , #*y ]
#level
     3
#end
```

The macro quadruple, $< S_1 \rightarrow S_2 : P, d >$, is composed of the following parts.

- S_1: the search string, which includes wild-card tokens
- S_2: the replacement string, which includes wild-card tokens.
- P: the priority of the macro, with priority 19 being highest priority, and priority 0 being the lowest.
- d: the direction of the scan, with *right* indicating right-to-left, and *left* indicating left-to-right.

In the FOBS notation of Example (4) the parts of the quadruple are specified using either the `#defleft`, or the `#defright` directive. Firstly, the directive specifies the direction d, depending on whether `#defleft` or `#defright` is used. Then the search string S_1, the replacement string S_2, and the priority P are specified, in order, separated by the two delimiters `#as`, and `#level`, and terminated by the `#end` directive.

The strings S_1, and S_2 are strings of FOBS lexicons, and wild-card tokens. Wild card tokens are all tokens that begin withe either the sequence "#?" or "#*", indicating a *single* wild card token, or a *mutiple* wild card token. A single wild card matches a single *atom*, and a multiple wild card matches a string of atoms. An atom is either a single FOBS token, or a balanced bracketed string, using one of the usual bracketing characters such as parentheses or braces.

Wild cards are named, so that the match in S_1 can be referred to in S_2. In Example (4), for example, the wild cards `#?x`, `#?y`, and `#*i` are matched in S_1, and their values are used in S_2.

The direction, d, and the priority of a macro, P, are used to control the associativity of the operator defined by the macro, and the precedence of the operator, respectively. To control associativity, macros defined with direction *left* are expanded left-to-right, resulting in the definition of a left-associative operator, and macros defined with a direction of *right* are expanded right-to-left, resulting in a right-associative operator. To control precedence, macros with higher priority are expanded before macros with lower priority, resulting in operators with different precedences.

5.2 The standard extension

The syntax in core-FOBS-X is a little cumbersome. It has been designed with minimalistic notation, allowing a concise formal description, given by Gil de Lamadrid & Zimmerman [4]. It is not necessarily attractive to the programmer. Standard extension (SE) FOBS-X attempts to rectify this situation. In particular, SE-FOBS-X includes constructs to enable the following.

- Allow infix notation for most operators.
- Eliminate the cumbersome syntax associated with declaring a FOB.
- Introduce English keywords to replace some of the more cryptic notation.
- Allow some parts of the syntax to be optionally omitted.

SE-FOBS-X is a language defined entirely using the macro processor. It demonstrates the flexibility of the FOBS-X macro capability to almost entirely rework the syntax of the language, without touching the back-end of the interpreter.

To help demonstrate the changes in syntax allowed by SE-FOBS, we rewrite the counter of Example (2) to in SE-FOBS.

```
#use #SE
## Implementation of a standard up-counter
(fob{
   public makeCounter
   val{
      fob{
         argument val
         ret{
            fob{
               count val{val} \
               public inc
               val{
                  fob{
                     ret{makeCounter                  (5)
                        [count + 1]}
                  \}
               } \
               ret{count}
            \}
         }
      \}
   }
\}
## test it
   .makeCounter[6].inc[].inc[])[]
#.
#!
```

The `#use` directive loads the standard extension macro file. This file makes available the syntax used in the remainder of the code. The most salient syntax feature of the code is the *fob* structure, used to define FOB-stacks. Each FOB in the stack is listed in the *fob* construct, and terminated by the "\" delimiter.

A FOB declaration contains a modifier, the identifier, a *val* structure, and a *ret* structure. The *val* structure defines the e-expression for the FOB, and the *ret* structure gives the ρ-expression for the FOB. Any of the parts of the FOB declaration may be omitted, resulting in the use of appropriate default values. Modifiers in SE-FOBS are the keywords *public*, *private*, and *argument*, instead of the cryptic symbols "'+", "'~" , and "'$".

A final feature present in Example (5) is the use of the infix version of the addition operator. All common binary operators in SE-FOBS are available in their infix version, relieving the user from using the normal core-FOBS access and invoke notation.

5.3 Extension library modules

Macro files extend the syntax of the FOBS language. To extend the semantics, you must add modules to the FOBS library. The FOBS library is written in Perl, and so to add modules you simply write Perl modules, and add them into the appropriate library directory structure. This process initially may sound simple. On further reflection, it becomes obvious that to do this

- One needs to be fairly familiar with the structure of the FOBS library.
- One must be familiar with how to manipulate FOBs in Perl.

While it is reasonable to expect a user requiring complex semantic changes to learn the required material to develop library modules from scratch, it is an unreasonable burden to impose on the user that desires to make only minor changes to the semantics of FOBS. To make small changes it is more appropriate for the user to do so using a tool that simplifies the process. The tool that we have developed is the FOBS Extension Definition Language Extension (FEDELE).

When designing FEDELE, we first thought of a meta-language that was implemented as an external tool. However, since FOBS is a scripting language, and designed for just such work, we rapidly realized that it made sense to implement FEDELE as a FOBS extension. FEDELE is, therefor, a FOBS extension that helps the user create other FOBS extensions.

6. FEDELE Operating Environment

The standard extension is unusual in that it is an extension with only one component: the macro file. No semantic changes are made to FOBS; only syntactic changes. Most extensions contain both a macro file and library modules. FEDELE is a more usual extension. Library modules provide the capabilities of the package, and a macro file provides more convenient syntax for using it.

The FEDELE extension provides a simpler way of writing the library modules necessary for implementing an extension. The FEDELE language allows the user to specify the structure of the extension much in the same way that YACC (see Johnson [7]) allows a programming language designer to specify the structure of a new programming language. The specification is translated into a set of Perl modules implementing the extension. The modules are then placed in a directory, and the directory path is placed on the Perl include path *@INC*, extending the directories searched for library modules. This summarizes the process of extending the library, but to continue our discussion of FEDELE, we will need to examine the structure of the FOBS library in more detail.

6.1 The FOBS library implementation

The FOBS library is composed of a collection of primitive and utility FOBS. As explained previously primitive FOBs use utility FOBs to mix-in general capabilities. However, from the standpoint of structure, there is no difference between a primitive and a utility FOB. In this discussion we will therefor consider only the structure of a primitive FOB.

To illustrate the structure of a primitive FOB, we take as example the FOB *Boolean*. The Boolean FOB can be represented in UML as shown in Figure 3. It contains an instance variable *val* that contains the actual Boolean value, represented as a character string. It also contains the common Boolean operations of *and*, "&", *or*, "|", and *not*, "!". In addition it contains the operator *if* that implements the implication operator. The FOB *Boolean* inherits operations from the FOBs *Eq*, and *Printable*. From *Eq* it inherits the operations *equals*, "=", and *not-equals*, "!=". From *Printable* it inherits the operation *toString*, that converts a Boolean value into a printable string.

It should be noted that the term "inhertitance" for primitive FOBs is only loosely applied. In fact, the mechanism is more of a message-forwarding mechanism. That is to say that, for example, if a *Boolean* FOB receives an *equals* access request, the request is forwarded to its parent *Eq* FOB.

Implementing the *Boolean* FOB in Perl is done with two structures: a hash table, containing the data of the primitive FOB, and a Perl module, *Boolean*, that contains code for all of the operations in the primitive FOB.

6.2 Primitive FOB hash table structure

The hash-table representing the data in a primitive FOB stores information in attribute-value pairs. The attributes of interest are the following.

- *type* - This attribute gives the type of the FOB. A primitive FOB is of type "omega", using the notation described by Gil de Lamadrid & Zimmerman [4].
- *code* - This attribute stores the name of the primitive FOB. For the FOB *Boolean*, the code attribute would have the value "Boolean".
- Super-FOBs - This is a collection of attributes, one per parent FOB. Each of these attributes stores an instance of one of the parent FOBs. For the FOB *Boolean* there are two such attributes. *superEq* stores an instance of the primitive FOB *Eq*, and *superPrintable* stores an instance of the primitive FOB *Printable*.

In adition to the above standard attributes, the primitive FOB hash-table contains attributes that are specific to the particular primitive FOB. For the *Boolean* primitive FOB, there is only one more attribute: the attribute *val*, that holds the boolean value of the FOB, stored as a character string.

6.3 Primitive FOB module structure

The main library module for a primitive FOB has the same name as the primitive FOB. For example, for the primitive FOB *Boolean* there is a Perl module called *Boolean*. This module has four standard functions in it.

- *construct* - This function constructs the hash table representing an instance of the primitive FOB.
- *constructConstant* - This function is an extension of the function *construct*. It constructs the instance, using *construct*, and then initializes it by filling in any instance variables.

- *alpha* - This is the function α that is described by Gil de Lamadrid & Zimmerman [4]. It takes a single argument, a character string, and accesses the primitive FOB for the value of the identifier specified by the argument.
- *iota* - This is the function ι described by Gil de Lamadrid & Zimmerman [4]. It takes a single argument, a *Vector* FOB, and invokes the primitive FOB using the vector to supply its actual arguments.

6.4 Operation modules

The main module of a primitive FOB is not the only module needed to define the FOB. To understand why this is so, consider the following FOBS code, and the semantics of invocation.

```
false.&[true]                                    (6)
```

In this expression, the Boolean FOB *false* is being accessed for its *and* operation. The operation is then being invoked, with the argument *true*. However, the question arises, when we say that the operation is invoked, what is an operation? The simple answer is that if an operation is invoked, then it must be a FOB, because only FOBs are invoked. This observation becomes trivially clear when we look at an example that does not involve a primitive FOB.

```
[‘+ & -> [‘~_ -> _ ^ false] ^ _] . & [true]   (7)
```

In this example, as in Example (6), a FOB is accessed for an ”&” operation, and the operation is invoked with the Boolean FOB *true*. The difference is that in Example(7) the FOB being accessed is not a primitive FOB. What is produced by the access operation is a FOB, in this case, that always returns the value *false*. We observe that the same must be true of Example (6). That is to say that an access operation always produces a FOB, whether the FOB being accessed is a primitive FOB or not.

What the above discussion points out is that when we access an operator in a primitive FOB, what is produced is a FOB. That FOB, when invoked would perform the particular operation. Every operator in a primitive FOB must have defined a FOB that will perform the given operation. For a library FOB such as *Boolean* each of its operators is defined as a primitive FOB. For example the *and* operator for the FOB *Boolean* is defined as a primitive library FOB called *Boolean_and*. We refer to library modules for the operations of a primitive FOB, as *primitive operation modules*.

To summarize, a primitive FOB is represented as a set of library modules. These consist of the main library module, described above, and a set of operation modules, one per operation. An operation module contains the same functions as the main module. That is to say that the operation module will have a *construct* function, a *constructConstant* function, an *alpha* function, and an *iota* function, each with the same role as in he main module. Each of these functions would perform actions appropriate to the operator. That is to say that the *alpha* function would always return an empty FOB, and the *iota* function would perform the operation of the operation module.

6.5 Extension access

Once the user has defined an extension, the language FOBS must be able to allow the user to use the extension. This section describes the mechanism used to allow FOBS code to use an extension.

The modules of the extension can be placed at any location in the directory hierarchy of the operating system. The author of the extension then must inform FOBS where the extension is located. As discussed previously, this is done by ensuring that the extension directory is on the list of include directories for Perl, *@INC*. This is easily done by seting the environment variable PERL5LIB to the extension path.

Recall that the two components of an extension are the macro file, and the library modules. We discuss how the FOBS interpreter locates both of these components in this section. We begin with how the macro file is located. A macro file is loaded with the *#use* directive. An example might be

```
#use Count
```

This directive tells the FOBS interpreter to look for a file called *Count.fobs* containing the macros of the extension. What the FOBS interpreter does then is to search Perl include directories, listed in the array *@INC*. There are two exceptions to the procedure, as illustrated in the following *#use* invocations.

```
#use #SE
#use #FEDELE
```

The extensions *#SE*, the standard extension, and *#FEDELE* are considered part of the FOBS language, and as such are located in a separate default FOBS include directory.

We now turn to the location of library modules. The standard mechanism for accessing the library in FOBS is a reference to a constant. For example, if a FOBS expression contains a reference to the constant `true`, the FOBS interpreter observes that this is a *Boolean* constant. The interpreter then goes to the default library directory, locates the main *Boolean* module, and invokes its *constructConstant* constructor function to create the hash-table. *ConstructConstant* also links the main module to the hash-table, a Perl mechanism called *blessing*, effectively making the hash table an *object*, in the object-oriented sense, which is to say that the hash table can be sent messages corresponding to any of the functions defined in the main *Boolean* module.

When the user defines their own library module, the above procedure cannot be used, because there is no FOBS constant for the new primitive FOB that would trigger the FOB construction. Instead, the construction of a FOB is triggered using the FOB *FOBS*. This is illustrated in the following FOBS expression.

```
FOBS.Count.new[5]                                (8)
```

The FOB *FOBS* is a primitive FOB in the FOBS library used to present links to extensions to the user. In Example (8), the user is attempting to access the identifier *Count*, which is the name of an extension. This identifier is not explicitly defined in the FOB *FOBS*. However, the FOBS interpreter will consider it implicitly defined, and, when referenced, will attempt to load the main module for the extension from the list of Perl include directories.

If we assume that the *Count* FOB is defined along the lines of the UML diagram in Figure 1, the *Count* FOB has one operation, *inc*, explicitly defined. For every extension, generated by FEDELE or not, the primitive FOB must also contain a *new* operation. This operation, when called, generates a new instance of the FOB, and calls the *constructConstant* constructor for the FOB. In Example (8), the *new* operator is called to construct a primitive *Count* FOB, initialized to the value 5.

7. The FEDELE Extension

This section describes the components of the FEDELE extension. FEDELE has both a macro file, extending the syntax of FOBS to more easily specify extension components, and library modules, providing the operators required to specify the contents of the library modules of the new extension, and write the module out. We begin by describing the FEDELE operations.

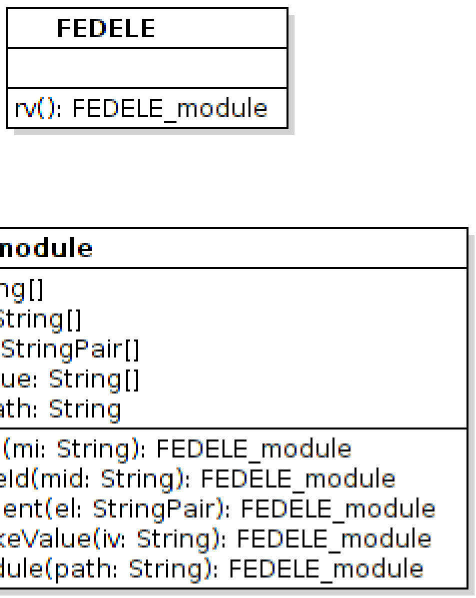
FEDELE
rv(): FEDELE_module
FEDELE_module
-mixIn:String[]
-makeID: String[]
-element: StringPair[]
-invokeValue: String[]
-modulePath: String
+addMixIn(mi: String): FEDELE_module
+addMakeId(mid: String): FEDELE_module
+addElement(el: StringPair): FEDELE_module
+addInvokeValue(iv: String): FEDELE_module
+writeModule(path: String): FEDELE_module

Figure 4. Interface for FEDELE

7.1 The FEDELE primitive FOB

The primitive FOB `FOBS.FEDELE` is a very uncomplicated FOB that has no accessible identifiers in it, and can only be invoked. The result of an invocation is a *FEDELE_module* FOB. The *FEDELE_module* is a data structure used to collect information on the new FOB being described. Figure 4 is the UML diagram showing the two FOBS: *FEDELE*, and *FEDELE_module*.

The *FEDELE_module* FOB contains variables for storing the following items

- *mixIn*: a list of primitive mix-in FOBs.
- *makeID*: a list of identifiers that will be included in the hash-table representing the FOB.
- *element*: a list of operations that will be included in the FOB. Each operation is represented as a pair consisting of the operation name, and a snippet of Perl code that will become the body of the *iota* function for the operation.
- *invokeValue*: a snippet of Perl code that will become the body of the *iota* function for the new FOB itself.
- *modulePath*: the directory on to which the files of the library module will be written.

In addition to the above variables, the *FEDELE_module* also contains operators for adding items to its data structures. Each operation adds an item and returns the modified *FEDELE_module*.

7.2 FEDELE macros

The second part of the FEDELE extension is a macro file that defines the FEDELE language, and allows easier specification of a primitive FOB. This FEDELE language is a structured language. The structures of the language are listed in Table 2.

A FEDELE specification, at the outer level is an *extension* structure. This structure would contain clauses; each clause being either a *mixIn*, a *make*, an *element*, or an *invoke* structure. This is illustrated more clearly in the next section. FEDELE translates the *extension* structure into FOBS code that creates a *FEDELE_module*. The clause structures are translated into *FEDELE_module* operations that add the appropriate elements to the module. For example, the *make* clause would translate into an invocation of the *addMakeId* operator shown in Figure 4.

Structure	**Description**
`extension "` *xtnd* `" {` *clauses* `} to` *path*	Defines an extension with name *xtnd*, to be written to the given directory path. It contains clauses giving the content of the extension *xtnd*.
`mixIn` *mixinFOB*	A clause indicating that the FOB *mixinFOB* from the library is a super-FOB for this FOB. This clause can be repeated to include several super-FOBs.
`make {` *idList* `}`	Describes the constant constructor for the FOB. The constructor will be available as the function `FOBS.`*xtnd*`.new`. *New* takes an argument for each identifier listed, and stores the argument as the value of the identifier. The *idList* is given as a list of strings, separated by commas.
`element "` *id* `" as "` *perlScript* `"`	Gives the name of an element, or operator, of the FOB available through the access operator. The included Perl script gives the value returned if the operator is invoked.
`invoke "` *perlScript* `"`	The Perl script gives the result of an invoke operation on the FOB itself.

Table 2. FEDELE macro operations

8. A FEDELE Example

We now present an example to illustrate how FEDELE is used. Suppose that the user wished to add a primitive FOB to the library that is similar to the counter FOB of Example (2). Remember that this example is illustrated in UML in Figure 1. The new library FOB, however, will be mutable. That is to say that a counter will have state, and each time the counter is incremented it will change its state, rather than produce a new counter with the modified state. This new counter will also support two new syntactic constructs: one to easily construct a counter, and one to increment the counter. The syntax of these operations is illustrated in the following example.

```
++(%C(5))
```

This example uses the ”%C” operator to create a counter initialized to 5. The second operator illustrated, ”++”, is used to increment a counter.

Our new counter will also allow the user to increment it by any value, as opposed to just an increment of one. An increment of more than one will not be supported by the macros, but can still be accomplished by using the *inc* function itself, as in

```
c.inc[3]
```

that increases the value of the counter *c* by 3. Figure 5 shows the new FOB structure in UML.

8.1 The counter FEDELE specification

The extension specification for our new counter consists of a FEDELE specification describing the library modules, and a macro file defining the syntax of the constructor operator, and the increment operator. We begin by presenting the FEDELE code to generate the library modules.

```
## FEDELE specification to generate
##  the example counter
#use #FEDELE

extension "Count" {
     mixIn "Printable"
     make {"count"}
     element "inc" as "
          $args = lib::PrimitiveFobs->
             thunkDown($args->[0]);
          if($args eq $undef){
               return(lib::PrimitiveFobs::
                  getEmpty())
          };
          if($args->{type} eq \"omega\" &&
             $args->{code} eq \"Int\"){
               $it->{count}->{val} +=
                  $args->{val};                    (9)
               return($it);
          }
          return(lib::PrimitiveFobs::
             getEmpty());
     "
     element "toString" as "
          my $v = $it->{count}->{val};
          return(lib::FEDELE::
             evalString(\"\\\"%C:\\\"
             .+[$v .toString[]]\"));
     "
     invoke "
          return($it->{count});
     "
} to "e:/fobs-x/code/Count"
#.
#!
```

Considering the overall structure of Example (9), it is , in fact, faithful to the UML description of Figure 5. It specifies a mix-in FOB *Printable*, an identifier *count*, two operations, *inc*, and *toString*, and a return value when invoked.

The code sections illustrate several issues concerned with the interface between FOBS and Perl. The first issue is how to enable a Perl segment to access the arguments of the function call. This is accomplished through the use of several special variables.

- `$it` - The FOB being operated on. That is to say that `$it` is the target of the invoke operation.
- `$args` - A *Vector* FOB containing the arguments of the invoke operation.

The variable `$it` contains all the identifiers declared in the FEDELE declaration as hash attributes. For instance, in the *Count* FOB, the sequence `$it->{count}` is the *count* identifier.

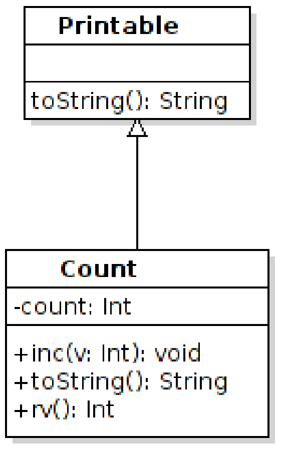

Figure 5. The mutable counter FOB

To access the arguments in the variable `$arg` a helper function is necessary. The arguments to FOBs are stored in thunks. To be used, the FOB inside the argument thunk must be unwrapped and evaluated. The function `lib::PrimitiveFobs->thunkdown` can be used for this purpose, as demonstrated in the definition of the operator *inc*.

There are a couple of other useful Perl functions used in Example (9). The function `lib::PrimitiveFobs::getEmpty` can be used to create an instance of the empty FOB, a FOB often used to signal an exception. Another function `lib::FEDELE::evalString` is used to evaluate FOBS expressions within the Perl code. This is a useful feature. Often it is easier to perform certain actions in FOBS, than in Perl. *EvalString* provides the ability to mix Perl with FOBS code, allowing the user to choose the more efficient implementation.

8.2 The counter macro file

The second component of the extension is the macro file that introduces more compact syntactic notation for the new counter operations. The contents of the file are shown in Example (10).

```
## macros for the Counter example FOB
#defleft
     % C ( #?op )
#as
     ( FOBS . Count . new [ #?op ] )
#level
     9
#end
                                                  (10)
#defleft
     ++ #?op
#as
     #?op . inc [ 1 ]
#level
     8
#end
#!
```

The macro file contains the definitions of two macros. The first one implements the constructor structure with the ”%C” notation. It matches a string beginning with the character sequence ”% C”, and followed by a single atom enclosed in parentheses. This sequence is replaced by an invocation of `FOBS.Count.new` with the matched atom as an argument.

The second macro is for the increment operator. It matches the operator ”++” followed by an atom, and replaces it with an invocation of the *inc* operation of the atom with argument 1.

8.3 Stateful and stateless programing

Example (10) demonstrates rather graphically one of the issues concerning the hybrid paradigm of FOBS. There is a dichotomy between functional programming and object-oriented programming. The object-oriented paradigm clearly involves the explicit maintenance of state. In fact we often refer to the bindings of instance variables as the state of the object. On the other hand, although state does exist in functional languages, and is usually maintained by the system stack, it is not manipulated explicitly, in the sense that the program does not change the state directly as is the case in imperative and object-oriented programs, but rather indirectly by invoking functions. But, this difference between the two paradigms often becomes significant, and produces awkward situations in FOBS.

One of the defining characteristics of FOBS is referential transparency. This puts FOBS squarely in the camp of stateless programming. This is seen when we observe, for example, that identifiers can be bound to a value only once. Mutable objects are not an option in this style of programming.

On the other hand, mutable objects are a staple of object-oriented programming. Also, state is often an integral part of scripting environments. For example, operating systems scripting often involves manipulating the state, represented as environment variables. To accommodate these situations in a language that advertises itself as an universal scripting language, it is not unreasonable for the user to wish to introduce state into FOBS. This is not difficult; the library can be extended to include mutable FOBS, as for example the *Count* FOB. However, it is still a stretch to use the language FOBS to manipulate these new mutable FOBS. In particular, what is needed to handle mutable FOBs is the ability to define operators whose return values are not used, but rather they are invoked for their side-effects on the state.

The problem of doing this type of stateful programming in a functional paradigm has been well researched, and has resulted in a body of literature on monadic programming (see Peyton Jones & Wadler [10], for example). Related to these results is a technique that has long been used in the object-oriented paradigm, called *method chaining*. This technique is used to pass multiple messages to the same object, as in the example

```
recipient.doX(xArg).doY(yArg)
```

in which the mutable object *recipient* is being first sent the message *doX* with the argument *xArg*, followed by the message *doY* with the argument *yArg*. Although the operations *doX*, and *doY*,naturally, might be thought of as returning no value, with the chaining technique they would instead return the object being operated on, *recipient*. In this way the next message in the chain is sent to the same recipient. One can think of the operators as passing the state along the chain from one operator to the next.

The technique of chaining is used in FOBS to handle mutable objects. Its effects can be observed in the code snippets of Example (9). The operator *inc* is defined to return the variable *$it*, which is the target FOB, and this allows FOBS expressions such as `++(++ %C(5))`, with a chain of increment operation being applied to the same FOB.

9. FOBS and Scripting

The intended use of FOBS is as a universal scripting language. Scripting languages are used to automate processes in a variety of environments. One of the most prevalent uses is in operating system interface. Scripting languages have also become very useful in creating dynamic web pages, and handling the collection of data using forms. They are also used in application programs, such as spreadsheets to automate calculations or procedures. In each of these applications the runtime system has two major components: an interpreter to execute scripts, and an interface that allows the script to interact with the environment. In FOBS-X, the library FOB *FOBS* is the interface to the environment. To adapt FOBS-X to a particular environment, an extension is created in the FOB *FOBS*. This extension contains all operations required for the interface, defined as FOBs.

As seen in the previous section, we have somewhat automated the process of creating these extensions. The user supplies a FEDELE description of an extension, and it is translated into a Perl definition. We have commenced the construction of a UNIX extension. We present an example of how this extension might be used in scripting.

A simple UNIX C-shell script follows that takes a command line argument, and prints out all files in the current directory containing that string.

```
#!/bin/csh
     if ( $#argv == 0 ) then
          echo Usage: $0 name
          exit 1
     else
          set user_input = $argv[1]
          ls | grep i $user_input
     endif
     exit 0
```

Assuming that an extension UNIX has been created, the above code could be translated into SE-FOBS-X as follows.

```
#use #SE
#use UNIX
     if {unix.args.length[] = 0} then {
          ## execute echo and exit in sequence,
          ## using the UNIX extension operation
          ## =>, (sequence)
          unix.echo["Usage: " + unix.args[0] +
             " name"] =>
          unix.exit[1]
     } else {
          fob {
               userInput
               val { unix.args[1] }
               ret {
                    ## use the UNIX package
                    ## operator || to perform
                    ## the UNIX pipe operation
                    ## on ls and grep, and use
                    ## the sequence operator to
                    ## follow this with an exit.
                    unix.ls[] || unix.grep["i",
                       userInput]
                       =>
                    unix.exit[0]
               } \
          }[]
     }
#.
#!
```

This script begins with two directives that inform the FOBS preprocessor that the standard and UNIX extensions are being used. The UNIX extension makes available the keyword `unix`, that is a convenience definition that allows the user to use this simple keyword, rather than the full specification, `FOBS.UNIX`.

Another notation defined in the UNIX extension is the operator ”=>”, which might be called the *sequence* operation. This operator is used to interact with UNIX, which is stateful, using the FOBS computational model, which is stateless. In UNIX, operations are performed in sequence, and although they return values, they are usually performed for their side effects. The sequence operator takes as operands two FOBs representing UNIX commands, performs them in sequence, alters the UNIX environment, and returns the return value of the last command as a FOB. The operator implements the chaining technique, discussed in Section 8.3.

A last notation used in the example is the operation ”||”. This is also part of the UNIX extension, and implements the UNIX pipe operation.

As a universal scripting language, FOBS will often be required to interact with stateful environments. The FOBS library gives FOBS that ability, although such interaction diminishes the referential transparency of the language. To ameliorate the situation, the library is structured to isolate all operations with side effects in the FOB *FOBS*.

10. Conclusion

We have briefly described a core FOBS-X language. This language is designed as the basis of a universal scripting language. It has a simple syntax and semantics.

FOBS-X is a hybrid language, which combines the tools and features of object oriented languages with the tools and features of functional languages. In fact, the defining data structure of FOBS is a combination of an object and a function. The language provides the advantages of referential transparency, as well as the ability to easily build structures that encapsulate data and behavior. This provides the user the choice of paradigms.

Core-FOBS-X is the core of an extended language, SE-FOBS-X, in which programs are translated into the core by a macro processor. This allows for a language with syntactic sugar, that still has the simple semantics of our core-FOBS-X language.

Because of the ability to be extended, which is utilized by SE-FOBS-X, the FOBS-X language gains the flexibility that enables it to be a universal scripting language. The language can be adapted syntactically, using the macro capability, to new scripting applications. The Extension FEDELE allows the semantics of the language to be adapted to new applications. FEDELE makes the process of extending the library easier, by automatically generating new library modules from a high-level specification language.

We are currently working on developing extensions for various scripting environments. Our next project is to produce a UNIX extension. Further in the future we plan to investigate using FOBS for web scripting applications.

References

[1] A. Alexandrescu *The D Programming Language*. Adison Wesley. 2010

[2] M. Beaven, R. Stansifer, D. Wetlow, : *A Functional Language with Classes*. In: Lecture Notices in Computer Science, Springer Verlag, 507. 1991

[3] D. Beazley, G. Van Rossum: *Python; Essential Reference*. New Riders Publishing, Thousand Oaks, CA. 1999

[4] J. Gil de Lamadrid, J. Zimmerman. *Core FOBS: A Hybrid Functional and Object-Oriented Language*. In: Computer Languages, Systems & Structures, 38. 2012

[5] J. Gil de Lamadrid. *Combining the Functional and Object-Oriented Paradigms in the FOBS-X Scripting Language*. In: International Journal of Programming Languages and Applications, AIRCC, Vol. 3, No. 2, Oct. 2013

[6] J. A. Goguen, J. Mesegner. *Unifying Functional, Object-Oriented, and Relational Programming with Logical Semantics*. In: Research Directions in Object-Oriented Programming, MIT Press, pp. 417-478. 1987

[7] S. C, Johnson. *Yacc: Yet Another Compiler-Compiler*. AT&T Bell Laboratories. 2014

[8] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Remy, J. Vouillon. *The OCaml System Release 4.00: Documentation and Users Manual*. Institut National de Recherche en Informatique et en Automatique. 2012

[9] M. Page-Jones. *Fundamentals of Object-Oriented Design in UML*. Addison Wesley, pp. 327-336. 2000

[10] S. L. Peyton Jones, P. Wadler. *Imperative Functional Programming*. POPL, Charleston, Jan, 1993

[11] S. S. Yau, X. Jia, D. H. Bae. *Proof: A Parallel Object-Oriented Functional Computation Model*. In: Journal of Parallel Distributed Computing, 12. 1991

[12] M. Odersky, L. Spoon, B. Venners. *Programming in Scala* , Artima, Inc. 2008

[13] T. Walid, T Sheard. *MetML and Multi-stage Programming with Explicit Annotations*, In: Proceedings of ACM SIGPLAN Symposium on Partial Evaluation and Semantic Based Program Manipulation, pp. 203-217, Amsterdam, NL. 1997