Functional-Logic Programming for Smalltalkers:

The FLOOP system

Zeki O. Bayram

Computer Engineering Department
Bogazici University
Bebek 80815/Istanbul-Turkey
internet: bayram@boun.edu.tr
Fax: 90 212 287 2461
Tel: 90 212 263 1500

Barrett R. Bryant

Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35294-1170, USA
internet: bryant@cis.uab.edu
Fax: 1 205 934-5473
Tel: 1 205 934-2213

Hakan Akcalar

Computer Engineering Department
Bogazici University
Bebek 80815/Istanbul-Turkey

Abstract

Motivated by the goal of being able to manipulate complex objects symbolically, we propose a method of integrating functional, logic and object-oriented programming paradigms. Our method assumes the existence of an object-expression evaluator (i.e. the underlying Smalltalk interpreter) and relies on transformations and calls to this object-expression evaluator as its means of computation. Programs of the combined paradigm consist of conditional rewrite rules augmented to incorporate object expressions.

1 Introduction

In this paper, we propose a method of integrating three programming language paradigms, namely *object-oriented, functional* and *logic*, in an intuitive, coherent and practical way. Integrating the three different styles of programming, when successful, has the natural consequence of permitting the programmer to approach different aspects of a problem in ways that are most convenient for those aspects, resulting in maximum program development convenience and productivity. Through the integration, the advantages of the component styles of programming naturally carry over.

The advantages of integrating functional and logic programming have been well demonstrated [1,2,3,4,5,6,7,11,14,15]. We are taking those advantages one step further in FLOOP by incorporating complex objects into the combined functional/logic paradigm. FLOOP is a system implemented in Smalltalk that achieves the integration of functional, logic and object oriented programming paradigms through a functional/logic interpreter based on transformations, an independent "object-evaluator" (i.e. the Smalltalk interpreter [19]) for evaluating object oriented expressions and an interface between the two. A FLOOP program consists of a set of conditional rewrite rules. The incorporation of the object oriented functionality is achieved through allowing *object expressions* (Smalltalk expressions evaluating to objects) to occur anywhere constants can in the rewrite rules. Object expressions can also contain logical variables.

The expressiveness of conditional term rewriting systems is well demonstrated [2,6]. In FLOOP, this expressiveness is taken to greater heights since any object-oriented expression (more specifically, any *Smalltalk* [19] expression) is allowed to appear anywhere in a rule where a constant can, and logical variables can be bound to objects. Naturally, the definition of a term also has to be changed to accommodate object-oriented expressions to be part of terms. Complex objects that are created in Smalltalk can thus be manipulated symbolically just as ordinary constants can. Moreover, since almost everything in Smalltalk is an object of some kind, including *messages* to objects, and a logical variable in a rule can be bound to an object, we get some unexpected higher-order features in the language FLOOP.

The proposed method of integration dictates the use of conditional rewrite rules to describe relationships between objects *externally*, but a different imperative language (namely, Smalltalk) to create classes, class hierarchies, objects, and methods. To justify this separation of the object-oriented component from the functional/logic component, we need to consider that in a language like Smalltalk, the main duty of methods is to *change* the internal state of the objects they *belong to*, and to *inform* the "outside" world of the internal state of the objects. Methods, then, need to be performing *imperative* actions when they are used for changing the internal state of objects, and an imperative style of programming is most convenient for that role. On the other hand, the relationships of objects to one another is *declarative*, and can be concisely expressed using conditional equality in the form of rewrite rules.

The remainder of this paper is organized as follows. Section 2 gives a brief introduction to the syntax and semantics of FLOOP. Section 3 describes the set of transformations used as the operational semantics.

Section 4 gives sample programs of the language, demonstrating the possibilities of being able to manipulate complex objects symbolically. Section 5 compares other approaches to the integration of the three paradigms to the transformations approach developed in this paper, and section 6 is the conclusion and future research directions.

2 FLOOP: A Brief Introduction to the System

In this section we give an informal introduction to the syntax and semantics of FLOOP programs. The language of implementation, as well as the underlying object-oriented component, is Smalltalk (we are assuming that the reader is familiar with Smalltalk, as well as basic term rewriting terminology). The operational semantics, which relies on transformations, is described in detail in the next section.

In order that we may give a declarative reading to FLOOP programs, we need to place certain restrictions on how methods should be written. These restrictions are:

- Let *m* be a method defined for the instances of a class *C*. If *m* changes the the internal state of an object it *belongs to*, then its last statement should be ^ *self*; conversely if it returns some other object, it must NOT change the internal state of the object it *belongs to*.

- If a message has arguments, the method activated by the message must not send state changing messages to any of the arguments. Hence a method can change the state only of the object it *belongs to*.

These restrictions allow us to regard methods as pure (polymorphic) functions without any side effects.

Further restrictions on the object oriented component are that each object must be able to respond to the *deepCopy* message, returning a structurally equivalent copy of itself, as well as the = message upon the receipt of which returns *true* if the argument to the message is structurally equivalent to itself, and *false* otherwise.

In the following, we assume the existence of two disjoint sets of function symbols: the set of *(data) constructors*, and the set of *defined* function symbols (also to be called *function names*).

We define an *(augmented) term* recursively as:

- A variable is a term (variables start with capital letters).

- A constant is a term (constants start with a small letter, and end with a delimiter like white space, comma, or a right paranthesis).

- If f is a function name, and a_i, $1 \leq i \leq n$, are terms, then $f(a_1, a_2, \ldots, a_n)$ is a term (called a *function application*).

- If c is a constructor, and a_i, $1 \le I \le n$, are terms, then $c(a_1, a_2, \ldots, a_n)$ is a term (called a *constructor term*).

- If *anyExpression* is a valid Smalltalk expression, then {*anyExpression*} is a term (called an *object term*).

- If a_i, $1 \le i \le n$, are terms, then $MSG(a_1, a_2, \ldots, a_n)$ is a term (called a *message term*). a_1 should eventually evaluate to an object, a_2 should evaluate to a valid Smalltalk message understood by a_1, and $a_3, \ldots, a_n$ should eventually evaluate to objects to be passed as arguments to the method a_2.

- If a_i, b_i, $1 \le i \le n$, are terms, then $NOT(a_1 = b_1, a_2 = b_2, \ldots, a_n = b_n)$ is a term (called a *negation term*). Informally, if the equations in the argument of the negated term cannot be shown to be true, then the negated term is true.

A *rewrite rule* is of the form

$$f(a_1, a_2, \ldots, a_n) : (b_1 = d_1, b_2 = d_2, \ldots, b_m = d_m) \rightarrow rhs$$

where $a_1, a_2, \ldots, a_n, b_1, d_1, b_2, d_2, \ldots, b_m, d_m$ and *rhs* are all terms, and f is a function name. $a_1, a_2, \ldots, a_n$ are not allowed to contain any function names. $f(a_1, a_2, \ldots, a_n)$ is called the *left hand side* of the rewrite rule, $(b_1 = d_1, b_2 = d_2, \ldots, b_m = d_m)$ is called the *condition*, and *rhs* is called the *right hand side* of the rewrite rule.

The meaning of a rewrite rule such as the above is as follows. Let σ be a substitution that makes the rule ground by substituting terms that contain only constructors, constants, or Smalltalk objects for each variable in the rule. We say that for any such substitution σ, $f(a_1, a_2, \ldots, a_n)\sigma$ is equal to $(rhs)\sigma$ iff $(b_1 = d_1)\sigma, (b_2 = d_2)\sigma, \ldots, (b_m = d_m)\sigma$ can all be shown to be true using the equality axioms and all the other (conditional) rules in the rulebase *after* all message terms have been simplified to single objects through being evaluated by the Smalltalk interpreter. Note that because of the limitations placed on methods, they can be seen in purely functional terms, taking in a set of objects as arguments, including the object to which they were sent, and returning another object as an answer (which may very well be the object to which they were sent, possibly with a different internal state). Thus we can give a fully declarative reading to rewrite rules. The declarative reading of a FLOOP *programs*, then, is the conjunction of the declarative reading of the individual rewrite rules in the program.

3 Operational Semantics Through Transformations

In this section, we give the set of transformations that form the core of the FLOOP interpreter. The method of transformations for solving unification problems both in the first order and higher order cases, either with or without equality, has been developed and advocated strongly by [8,10,12,16,17,18]. Indeed, the method provides "an abstract and mathematically elegant means of analyzing the invariant properties of unification in various settings by providing a clean separation of the logical issues from the specification of procedural information" [18]. The usefulness of transformations is that they can be used (as we have done, though in some restricted form due to efficiency considerations) as the operational semantics of functional/logic languages in a very straightforward manner.

Our transformations, which are based on those in [10], force the binding of variables *only* to terms that denote a *value* (i.e. a term that contains no defined function symbols and no subterm of the form $MSG(\ldots)$). This implements an *eager* strategy for parameter passing. Because of this eager strategy, our transformations are not complete. The alternative of trying to achieve completeness would have resulted in much more nondeterminism and inefficiency. The transformations (excluding the ones that deal with the object oriented extensions to keep the discussion confined to first order equational logic) are sound, however, since any rule that we use is either a restricted version of a rule in [10], or can be simulated by *narrowing* and *reflection* as defined in [10].

3.1 The Transformation Algorithm

A FLOOP program consists of a set of rules of the form

$$f(t_1, t_2, \ldots, t_n) : (r_1 = s_1, r_2 = s_2, \ldots, r_m = s_m) \rightarrow t.$$

The goal of the transformation algorithm is to solve a set of equations of the form $(e_1 = d_1, e_2 = d_2, \ldots, e_p = d_p)$, i.e. find values (terms that consist only of variables, constants, constructors or objects) for variables in the equations such that the equalities can be shown to be true (using the rules in the program) when the values are substituted in the place of variables in the equations. The methodology described below returns answers in the form of *sequential* substitutions. We define a *sequential* substitution as a list of pairs of terms, $< a_1/X_1, a_2/X_2, \ldots, a_n/X_n >$, where a_i is a term that contains no function applications, and X_j is any variable. Let the notation $t[a/X]$ mean the term obtained when all occurrences of the variable X in t are replaced by the term a (if a is an object term, then by a **copy** of a). Then the *application* of a sequential substitution σ (e.g. $< a_1/X_1, a_2/X_2, \ldots, a_n/X_n >$) to a term t, $(t)\sigma$, is the term $(((t)[a_1/X_1])[a_2/X_2]) \ldots [a_n/X_n]$. This is the same as applying γ (in the conventional sense) to t where γ is the composition of the substitutions $\{a_1/X_1\}, \{a_2/X_2\}, \ldots, \{a_n/X_n\}$.

The Algorithm: We start out with a *pair* of ordered lists of equations (F,S). F, which will eventually become the basis for a solution, is empty. S is the original list of equations to be solved (i.e. the *goal*). If S ever becomes empty as a result of the transformations, F will give us the desired solution. We perform the transformations, always choosing the leftmost equation in S to *operate* on, until S becomes empty, in which case we terminate successfully, or until no transformation applies, in which case we stop with failure. Upon successful termination of the transformation sequence, if F is $< X_1=t_1,\ X_2=t_2,\ \ldots\ ,X_q=t_q >$, then the sequential substitution $< t_1/X_1,\ t_2/X_2,\ \ldots\ ,t_q/X_q >$ is an answer.

3.2 Transformations

Below is the set of transformations (described using terminology as consistent with [10] as possible) used by the FLOOP interpreter. Lists are represented as $< \ldots >$ and $\bullet$ will mean list concatenation. To avoid duplication, we shall regard $a=b$ as representing $b=a$ also. The pair of lists will be represented by (L_1,L_2) etc.

Removal of Trivial Equations:

$$(L_1,\ < a = a > \bullet L_2) \Rightarrow_t (L_1, L_2)$$

if a is a variable, a constant or an object.

Variable elimination 1:

$$(L_1, < X = a > \bullet L_2) \Rightarrow_{ve1} (L_1 \bullet < X = a >, L_2[a/X])$$

if X is a variable, and a is a variable, or a constant and $X \neq a$.

Variable elimination 2 (objects):

$$(L_1, < X = a > \bullet L_2) \Rightarrow_{ve2} (L_1 \bullet < X = copy(a) >, L_2[a/X])$$

if X is a variable, and a is an object. Note that a **copy** of a is placed in F. Furthermore, each occurrence of X in L_2 is replaced by a distinct **copy** of a.

Imitation:

$$(L_1, < X = c(s_1, \ldots ,s_n) > \bullet L_2)$$
$$\Rightarrow_i (L_1 \bullet < X = c(Y_1, \ldots ,Y_n) >, < Y_1 = s_1, \ldots ,Y_n = s_n > \bullet (L_2[t/X]))$$

where c is a constructor, X is a variable, X does *not* occur in $c(s_1, \ldots ,s_n)$, $c(s_1, \ldots ,s_n)$ does not contain any defined function symbols, $t = c(Y_1, \ldots ,Y_n)$, and $Y_1, \ldots ,Y_n$ are all new variables. This rule helps incrementally bind a variable to its final value.

Exposure 1:

$$(L_1, < X = t > \bullet L_2) \Rightarrow_{e1} (L_1, < Y = t[\alpha], X = t[\alpha \leftarrow Y] > \bullet L_2)$$

where X is a variable, Y is a new variable, $t = c(a_1, \ldots, a_n)$, c is a constructor, $t[\alpha]$ is the subterm of t at address α, $t[\alpha \leftarrow Y]$ is the term obtained by replacing the subterm of t at address α with the term (in this case variable) Y, $t[\alpha] = f(\ldots)$, f is a defined function symbol, and $f(\ldots)$ is not a proper subterm of any $g(\ldots)$ where g is a defined function symbol and $g(\ldots) = t[\gamma]$. This rule exposes a reducible expression in a goal equation so that lazy narrowing can be applied to it. The restriction that X be a variable ensures that the *imitation* rule and this rule are not simultaneously applicable.

Exposure 2:

$$(L_1, < s = t > \bullet L_2) \Rightarrow_{e2} (L_1, < Y = t[\alpha], s = t[\alpha \leftarrow Y] > \bullet L_2 >)$$

where Y is a new variable, $t = MSG(a_1, a_2, \ldots, a_{n'})$, s is *any* term and the rest is the same as in *Exposure 1*. Note that there is no restriction on s.

Decomposition:

$$(L_1, < c(s_1, \ldots, s_n) = c(t_1, \ldots, t_n) > \bullet L_2) \Rightarrow_d (L_1, < s_1 = t_1, \ldots, s_n = t_n > \bullet L_2)$$

where c is a constructor, and $s_1, \ldots, s_n, t_1, \ldots, t_n$ are all terms.

Lazy Narrowing:

$$(L_1, < t = f(s_1, \ldots, s_n) > \bullet L_2)$$
$$\Rightarrow_{ln} (L_1, < a_1 = s_1, \ldots, a_n = s_n, b_1 = d_1, \ldots, b_m = d_m, t = rhs > \bullet L_2)$$

where $f(a_1, \ldots, a_n) : (b_1 = d_1, \ldots, b_m = d_m) \rightarrow rhs$ is a rewrite rule where all variables have been renamed so that they are different from the variables occurring in the two lists.

Simplification:

$$(L_1, < t = MSG(arg_1, arg_2, \ldots, arg_n) > \bullet L_2) \Rightarrow_s (L_1, < t = s > \bullet L_2)$$

where $arg_1, arg_2, \ldots, arg_n$ are all either message terms or object terms. Let *obj* be the object returned by the Smalltalk interpreter upon the evaluation of the message contained in arg_2 to the object contained in arg_1 with the arguments $arg_3 \ldots arg_n$; then s is the object term containing *obj*. If any arg_i is a message term, then it is (recursively) simplified to obtain an object term to be used as described above.

Negation:

$$(L_1, < t = NOT(a_1 = b_1, \ldots, a_n = b_n) > \bullet L_2) \Rightarrow_{neg} (L_1, < t = bool > \bullet L_2)$$

where if $< a_1=b_1, \ldots, a_n=b_n >$ can be shown to have at least one solution (using the transformations recursively) then *bool* is the constant *false*, otherwise, it is the constant *true*. This is a version of the *closed world assumption* in the context of (conditional) equality theories.

This concludes the transformations. An observation about the first list is that at all times it contains equations only of the form $X=t$ where X is a variable and t is either a variable, a constant, an object, or a term of the form $c(Y_1, \ldots, Y_n)$ for some n, and each Y_i is a variable. Consequently variables in the original set of equations will not be bound to terms containing any defined function symbols, or terms of the form $MSG(\ldots)$.

Example 1 To demonstrate how the transformations work, let $P = \{ f(X):() \Rightarrow a \}$ be a FLOOP program, and $E = \{ Z=c(f(Z)) \}$ be the set of equations to be solved. We have

$$\begin{aligned}
&(<>, < Z = c(f(Z)) >) \\
&\Rightarrow_{e1} (<>, < W_1 = f(Z), Z = c(W_1) >) \\
&\Rightarrow_{ln} (<>, < X_1 = Z, W_1 = a, Z = c(W_1) >) \\
&\Rightarrow_{ve1} (< X_1 = Z >, < W_1 = a, Z = c(W_1) >) \\
&\Rightarrow_{ve1} (< X_1 = Z, W_1 = a >, < Z = c(a) >) \\
&\Rightarrow_{i} (< X_1 = Z, W_1 = a, Z = c(W_2) >, < W_2 = a >) \\
&\Rightarrow_{ve1} (< X_1 = Z, W_1 = a, Z = c(W_2), W_2 = a >, <>)
\end{aligned}$$

The answer substitution σ is obtained from the first list, i.e. $\sigma = < Z/X_1, a/W_1, c(W_2)/Z, a/W_2 >$ (remember that σ is a *sequential* substitution). We confirm that

$$\begin{aligned}
&(Z)\sigma \\
&= (Z) < Z/X_1, a/W_1, c(W_2)/Z, a/W_2 > \\
&= (Z) < a/W_1, c(W_2)/Z, a/W_2 > \\
&= (Z) < c(W_2)/Z, a/W_2 > \\
&= (c(W_2)) < a/W_2 > \\
&= (c(a))
\end{aligned}$$

as we had expected.

4 FLOOP Through Examples

In this section we give two examples of FLOOP programs demonstrating the key features of the system. We assume reader familiarity with Smalltalk.

Example 2 The following is the insertion sort algorithm that sorts *any* list of objects that accept the messages < and >=. Each line in the program is numbered so that they can be explained in detail.

```
(1) | rb answer |
(2)  rb := RuleBase new2.
(3)  rb addConstructor: 'c'.
(4)  rb addRules: '
(5)     iSort(nil):()->nil.
(6)     iSort(c(H,T)):()->insert(H,iSort(T)).
(7)     insert(X,nil):()->c(X,nil).
(8)     insert(X,c(H,T)):(MSG(X,{#<},H)={true})->c(X,c(H,T)).
(9)     insert(X,c(H,T)):(MSG(X,{#>=},H)={true})->c(H,insert(X,T)).
(10)    list1():()->c({4},c({2},c({5},c({3},nil))))'.
(11) answer := rb solveEquations: ' ( X = iSort(list1()) ) '
                  instantiating:'(X)'.
(12) ^ answer.
```

Explanation:

(1) Declare the local variables "rb" and "answer." Note that the FLOOP interpreter is implemented in Smalltalk, and the underlying object expression evaluator is also Smalltalk, hence the Smalltalk code in "setting up" the FLOOP program.

(2) Assign a new instance of Rulebase to "rb." "rb" will hold the rewrite rules that make up the FLOOP program. "new2" is a class method for "Rulebase" that returns an instance of "Rulebase."

(3) Declare "c" to be a constructor.

(4) Self explanatory.

(5) Self explanatory.

(6) Self explanatory.

(7) Self explanatory.

(8) is being sent the message "# <." Note the free mixing of logical variables and messages to objects that the variables will be instantiated to.

(9) Similar to (7), except the message now is "# >=."

(10) A constant function that evaluates to a list of Smalltalk objects is defined (note the curly brackets around the numbers which signify that their contents are Smalltalk expressions to be evaluated).

(11) The rulebase (rb) is given the message to solve a set of equations, and instantiate the variable *X* with the solutions found to the equations. In general, if '(X_1, X_2, … ,X_n)' is the list of variables to be instantiated, then { (X_1, X_2, … ,X_n)σ | σ is a substitution found by the transformations } is what is returned as an answer.

(12) Return the answer, in this case

```
"X:c( 2 c( 3 c( 4 c( 5 nil ) ) ) )"
```

In the Smalltalk implementation of FLOOP, every FLOOP program has to be in the format shown in Example 2 . In this scheme an advantage that we notice is that we can have more than one rule base at any given time, each containing a different set of rules. This amounts to having a module system of rewrite rules.

Example 3 This following program demonstrates the true modeling capabilities of the combined functional/logic/object-oriented paradigms. We have again labeled the program for easy explanation. This program represents information about 4 cities, including the distances between them, and computes the minimum traveling distance between two of the cities.

```
(1)  | rb answer aStream smallestDistance aNumber |

(2)  Tuple          subclass: #City
              attributeNames: 'name population'
              attributeTypes: 'Symbol Integer'.

(3)  rb := RuleBase new2.

(4)  rb addConstructor: 'c'.

(5)  rb addRules: '

(6)       city(a):()->{ (City new)  name:#Atlanta;
                                    population: 1600 }.

(7)       city(b):()->{ (City new)  name:#Birmingham;
                                    population: 1000 }.
```

```
(8)       city(c):()->{ (City new)  name:#Cambridge;
                          population: 3000 }.

(9)       city(d):()->{ (City new)  name:#Denver;
                         population: 200 }.

(10)      dist(a,b):()->{10}.
          dist(a,d):()->{50}.
          dist(b,c):()->{20}.
          dist(b,d):()->{30}.
          dist(c,d):()->{5}.

(11)      twoWayDist(X,Y):()->dist(X,Y).
          twoWayDist(X,Y):()->dist(Y,X).

(12)      member(X,nil):()-> false.
          member(X,c(X,Z)):()->true.
          member(X,c(Y,Z)):( NOT(X=Y)=true )->member(X,Z).

(13)      comp(Name1, Name2):( city(A)=C1,
                          MSG(C1,{#name}) = Name1,
                          city(B)=C2,
                          MSG(C2,{#name}) = Name2
                          )
                         ->  computeDistance( c(A,nil), A, B).

(14)      computeDistance(L,A,B):()->twoWayDist(A,B).

(15)      computeDistance(L,A,B):
                  ( twoWayDist(A,C)=D,
                    NOT(C=B)=true,
                    member(C,L)=false,
                    computeDistance(c(C,L),C,B)=E
                  )
                 -> MSG(D,{#+},E) '.

(16)  answer  := rb solveEquations: '( W=comp({#Denver},{#Atlanta}))'
                    instantiating: '(W)'.

(17)  answer do:
        [ :anOrderedCollection |
              aStream := ReadStream on: anOrderedCollection.
              aNumber := (aStream peek) object.
              (smallestDistance isNil)
```

```
                    ifTrue: [ smallestDistance := aNumber ]
                    ifFalse: [ (smallestDistance > aNumber)
                                ifTrue: [ smallestDistance := aNumber ]
                             ]
        ].

(18) ^ smallestDistance
```

Explanation:

(1) Local variables for the Smalltalk code that follows.

(2) Tuple is a class whose instances respond to the = and *deepCopy* messages (as described in the text). Subclasses of Tuple that are created with this message have attribute names with types, and each instance of such a class responds to the message *x* and *x:* if *x* is an instance variable of its class. Here we are creating "City" as a subclass of Tuple with instance variable names "name" and "population" whose types are "Symbol" and "Integer" respectively.

(3) Get a new instance of class "Rulebase" which will hold the rewrite rules defining the FLOOP program.

(4) Define "c" to be a constructor.

(5) Add the rules contained in the string to the rulebase.

(6) The rule evaluates to the "City" object whose *name* attribute is "Atlanta" and *population* attribute is *1600* (numbers made up!). "a" (which is a plain constant) in the argument of the rewrite rule will be used to easily refer to this object, and act as a kind of object identifier for it.

(7) Similar to (6)

(8) Similar to (6)

(9) Similar to (6)

(10) Distance between any two cities that are directly connected.

(11) Distance between any two cities without directionality.

(12) The familiar *member* function. Note that the condition part of a rule can contain *only* equations.

(13) Compute the distance between two cities whose names are *Name1* and *Name2*. In the condition part, we find the constants that are the object identifiers for the cities involved, and use them to compute the

distance in the body of the rule. This is a perfect example of the separation of the internals of objects, and their relationships to one another.

(14) See **(15)**

(15) *L* is the list of cities visited so far (to avoid infinite loops). These rules implement a graph traversal algorithm, keeping track of the nodes visited so far in *L*. The underlying object evaluator (i.e. Smalltalk) is used to evaluate the numerical expressions.

(16) Ask the rulebase to solve the given goal, instantiating only the variables specified as the answer.

(17) *answer* is assigned a *Set* of *orderedCollections* of *Terms*. Each term, if it contains an object, responds to the message *object* and returns the object it contains. This piece of code checks each of the values *W* was bound to, and returns the smallest of them (could have been done in many other ways too).

(18) Return *smallestDistance* as the answer, in this case "35."

5 Related Work

LIFE [11] and *MaudeLog* [13] are languages that represent significant efforts in joining the three paradigms of functional, logic and object oriented programming.

LIFE (Logic, Inheritance, Functions, Equations) is based on the idea of generalizing the notion of a term to an *Order Sorted Feature (OSF* for short*) term* which permits self reference, has attributes and belongs to a *sort*. A hierarchy of sorts is possible, consequently LIFE has *structured type inheritance*. LIFE programs consist of Prolog-style clauses and function definitions which are basically rewrite rules.

It is possible to write very expressive programs in LIFE mainly due to the OSF terms which allow for complex constraints to be placed on their attributes, and an (obviously powerful) OSF term unifier that unifies two OSF terms. However, to what degree does structured type inheritance (the only object oriented feature in LIFE) qualify LIFE to be an object oriented language, when such ideas as *local state in an object* , *method definitions*, *inheritance of methods*, *method overriding* and *method refinement* which are so central to the object oriented paradigm are missing? These very features are what make object oriented programs reusable, maintainable and robust. FLOOP, on the other hand, has all the facilities of the underlying object oriented component (i.e. Smalltalk), in addition everything we have come to expect from functional/logic programming, such as the *logical variable*, *non-determinism* and *higher order functions*, in a coherent framework.

MaudeLog (an extension of *Maude*) which is described in [13] is based on *rewriting logic*. *MaudeLog* unifies the paradigms of functional, relational and *concurrent* object oriented programming. *MaudeLog* programs consist of *rewrite rules* in *system modules*, *equations* in *functional modules* and *methods* in *object oriented modules*. Object oriented modules define the classes and the attributes of instances of the class. The idea of a method belonging to an object and modifying only that object is generalized to a method not belonging to any object, but rather a "set" of objects that it can operate upon. Methods are then rewrite rules that expect to be passed as one of their parameters each object they "belong to" and which they can modify. The result of a rewrite rule being applied to a set of objects is the same set of objects, but possibly with different internal states. The whole process is dynamic, with a common store of objects and messages to objects "floating in a vacuum." If a message is applicable to a set of objects, the corresponding rewrite rule is activated, possibly changing the internal state of the objects it operated upon, hence the concurrency.

MaudeLog is not directly comparable to FLOOP. It incorporates concurrency, and a new way of looking at method definitions. Disregarding the concurrency aspect, however, despite the fact that the ideas behind "rewriting logic" are simple, the language itself is very involved, and does not seem to be suitable for use as a general purpose programming language. Another question is whether *MaudeLog* can be implemented with reasonable efficiency: according to [13] its implementation is still far away. We do have for FLOOP however a working prototype, and the language FLOOP was designed with practicality in mind.

6 Conclusion and Future Research Directions

In this paper, we described a way of integrating the three paradigms of functional, logic and object oriented programming without sacrificing any of the desirable aspects of each component paradigm. The system we designed allows programs consisting of conditional term rewriting systems to be run inside Smaltalk. A fixed set of transformations are used for the operational semantics of the system. The main highlights of the system are logical variables, programs composed of conditional rewrite rules that can contain arbitrary object expressions, higher order features and all the facilities provided by the underlying object expression evaluator (i.e. Smalltalk) such as creating classes, class hierarchies, and objects. The programs of the combined paradigm still maintain a declarative reading, however one that takes into account the operational aspect introduced by the calls for evaluation of object oriented expressions to the underlying object expression evaluator.

Seen in a greater perspective, we believe that our method, by providing for declarative manipulation of complex objects, lays the framework for maximum program development convenience and productivity. This is aided by the fact that FLOOP has a simple syntax and very intuitive semantics (i.e. everything works the way you would expect it to work). The programmer, using FLOOP, can approach different aspects of a problem in ways that are most convenient for those aspects.

Future work in this direction would include developing a more rigorous declarative semantics for FLOOP programs, as well as investigating the possibility of parallel execution.

References

[1] Aït-Kaci, H. An Overview of LIFE, in: Proceedings of the 1st International Database Workshop, Springer-Verlag, LNCS 504, 1991, pp. 42-48.

[2] Bayram, Z.O., and Bryant, B.R., ROSE: A Higher Order Functional/Logic Programming Language, Technical Report, CIS-TR-91-09, Department of Computer and Information Sciences, University of Alabama at Birmingham, 1991.

[3] Bosco, P.G., and Giovannetti, E., IDEAL: An Ideal Deductive Applicative Language, in: *Proceedings of the 1986 Symposium on Logic Programming*, 1986, pp. 89-94.

[4] DeGroot, D., and Lindstrom, G. (eds.), Logic Programming: Functions, Equations, and Relations, Prentice-Hall, 1985.

[5] Dershowitz, N., and Plaisted, D., Logic Programming cum Applicative Programming, in: *Proceedings of the 1985 Symposium on Logic Programming*, 1985, pp. 54-67.

[6] Dershowitz, N., and Okada, M., Conditional EquationalProgramming and the Theory of Conditional Term Rewriting, in: *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems,* 1988, pp. 337-346.

[7] Fribourg, L., SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting, in: *Proceedings of the 1985 Symposium on Logic Programming*, 1985, pp. 172-184.

[8] Gallier, J.H. and Snyder, W., A General Complete E-Unification Procedure, in: *Rewriting Techniques and Applications*, Springer-Verlag, LNCS 256, 1987, pp. 216-227

[9] Giovannetti, E., and Moiso, C., A Completeness Result for E-unification Algorithms Based on Conditional Narrowing, in: *Foundations of Logic and Functional Programming Workshop,* Springer-Verlag, LNCS 306, 1986, pp. 157-167.

[10] Hölldobler, Steffen, Conditional Equational Theories and Complete Sets of Transformations, in: *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1988, pp 405-412.

[11] Levi, Giorgio et. al., A Complete Semantic Characterization Of K-Leaf, A Logic Language With Partial Functions, in: *Proceedings of the 1987 Symposium on Logic Programming*, 1987, pp. 318-327.

[12] Martelli, and A., Montanari, U., An Efficient Unification Algorithm, *ACM Transactions on Programming Languages and Systems* 4:2 (1982) 258-282.

[13] Meseguer, J., A Logical Theory of Concurrent Objects and its Realization in the Maude Language, Technical Report, SRI-CSL-92-08, July 1992.

[14] Reddy, Uday S., Narrowing as the Operational Semantics of Functional Languages, in: *Proceedings of the 1985 Symposium on Logic Programming*, 1985, pp. 138-151.

[15] Silberman, Frank S. K., and Jayaraman, Baharat, A Domain-theoretic Approach to Functional and Logic Programming, Technical Report, TUTR 91-109, Tulane University, 1991.

[16] Snyder, W., Higher Order E-Unification, Technical Report, BU-CS TR 90-008, Boston University, 1990.

[17] Snyder, W., and Lynch, C., An Inference System for Horn Clause Logic with Equality: A Foundation for Conditional E-Unification and for Logic Programming in the Presence of Equality, Technical Report, BU-CS TR 90-014, Boston University, 1990.

[18] Snyder, W. and Gallier, J., Higher-Order Unification Revisited: Complete Sets of Transformations, *Journal of Symbolic Computation,*8 (1989) 101-140

[19] The Smalltalk/V Tutorial and Programming Handbook, Digitalk Inc., 1986.