JCS&T Vol. 7 No. 2 April 2007

ECOLE* : a pedagogical environment for non procedural languages

Pr D.E Zegour (d_zegour@ini.dz)
Institut National d'informatique, Oued Smar, Alger
and
W.K Hidouci (Hidouci@ini.dz)
Institut National d'informatique, Oued Smar, Alger

ABSTRACT

The work described in this paper is related to three areas in the programming world : logic, functional and object programming. The main objective is essentially pedagogical since it is question here to make a synthesis on non procedural languages. To achieve this, we have considered many construction types, each one represents the one of evoked programming. Many fully-documented environments have been developed for writing constructions of any type, transforming them in order to evaluate them by showing the work really accomplished in the least detail.

Keywords : Logic programming, Functional programming, Object programming, Compiler, Frames, Actors, Combinators.

* Work supported by the Ministry of the research of Algiers under the research project entitled "ECOLE".

1.INTRODUCTION

The project ECOLE is the continuation of project CONCORD[8]. The latter makes a synthesis on procedural languages in which the main task is to give the procedure or the algorithm to follows to resolve a given problem. In other words, this type of programming is based on Von Neumann machines in which the assignment operation is the basic one. In CONCORD, we have studied mainly three basic schemes corresponding essentially to programming with (1) only conditional and unconditional jump as statement sequence control, (2) exclusive control structures due to Dijkstra : if then else and While, (3) the unique control structure if then else with recursive calls. For each language we have developed an syntactic editor. A complete taxonomy of algorithm transformation is made in CONCORD. Beside the interpreters and symbolic evaluators made in CONCORD, different approaches as well on formal systems as on the fix point theory are implemented to prove algorithms corresponding to the languages considered.

ECOLE make a synthesis as well on non procedural programming (logic and functional programming) as on object programming in all its forms. If in procedural programming the solution is expressed explicitly, the solution is rather implicit in a non procedural one. Considering the existing languages (Lisp, Prolog, Act1, Plasma), we have conceived simple languages to covers the focused areas of programming. The extended normal forms of Backus-Naur corresponding to these languages are in [7].

The constructions considered represent modes of resolution of problems wholly different. Each mode is devoted to resolve a specific class of problems. In object programming with classes (Oc), we focus the reuse of code. Many systems have used this type of programming to elaborate convivial interfaces : Window, Visual Basic, Turbo-Vision, Delphi. The object programming by frames (Of) is specialized to manage knowledge bases. The object programming by actors(Oa) is used in a parallel environment in which several objects (actors) cooperate for the elaboration of a given task. The logic programming(L) is used in artificial intelligence and the expert systems. The functional programming(Fv for functional programming with variables and Fc for functional programming with combinators) is another way to write solutions expressed only with functions represented as lists. Some bibliographical references concerning these types of programming are in [1, 2, 3, 4, 5,6].

ECOLE offers tools to express solutions in any type among the constructions considered {L, Fv, Fc, Oc, Of, Oa}.

ECOLE studies the transformation from any logic construction to the clausal form and from any functional construction to λ-calculus. ECOLE provides also all the internal forms produced by our compilers. The different mechanisms are studied in ECOLE for the implementation of interpreters related to object languages, the resolution methods related to logic constructions and the reduction engines related to functional constructions.

Since the objective of ECOLE is above all pedagogical, it shows the whole clearness on the languages, transformations, interpreters, resolution techniques, reduction engines, etc..

The paper is organized as follows : we begin by describing the different construction types. Then, we present the construction, transformation and evaluation phases. Once the architecture of ECOLE environment is depicted, we give briefly the works realized. We present then two screen dumps : the first one deals with the object programming by class, the second with the reduction engines for Fx-constructions. Finally, the educational aspect is described, and concluding remarks are given.

2. THE LANGUAGES CONSIDERED

In our project, we consider the following constructions corresponding to one purely logic language, two purely functional languages : with variables and with combinators and three purely object languages : by class, by frame and by actor. We give below the different types of construction. We use the term “purely” to designate any construction without the sequencing as control structure and without the assignment operation.

L-construction

It is related to logical programming. In this type of language, one uses the language of the first order predicate logic with only the clauses of Horn in order to specify the characteristics of the problem to be solved. The interpreter, who in this case is a theorem demonstrator, will have to undertake the resolution of the specified problem. The language we have considered is inspired of Prolog. The variables are identifiers starting with a capital letter or the character ' _ '. the handled objects is the terms, definite recursively by:

- any constant or variable is a term.
- if f is a n-ary symbol of function and t1, t2... tn are terms, then f(t1,t2...tn) is also a term.

A statement of the language (a declaratory instruction) is:

- either a fact of the form p(t1,t2... tn). with p a n-ary symbol of predicate
- or a rule of the form p(...) if q1(...) and q2(...) and... with p,q1,q2... symbols of predicates.

To invoke the inference engine, we must ask a question relating to the problem already described. This question (or instruction-Goal) is of the form: q1(...) and q2(...) and...
Example: Membership of an element to a list:

```
Exist(X, [ X/_]).                          a fact
Exist(X, [ _ / L ])    if  Exist(X, L).    a rule
```

Goal: Exist(X,[1,2,3]) and Exist(X,[2,5,3,8]) and X > 2.
Answer X = 3 (only one solution)

Fv-construction
It is related to functional programming with variables. This type of programming avoids the side effects. Likewise the purely functional languages, our language is based essentially on the calculation of functions and is characterized by the absence of the assignment operation, explicit control and sequentiabilty. Our language is based on λ-calculus like pivot language, and inspired of Lisp in its syntax. The single structure of control is the call of functions. There is neither assignment neither sequential, nor operation of control break. The conditional operation (if cond exp1 exp2) is regarded as a non strict function with three arguments (cond, exp1 and exp2). A functional program is a succession of definitions of functions followed by a principal expression to evaluate.
Example:

```
(def  sum  (L)  (if  (Nul  L)  0  (+  (head L)  (sum  (tail L)))))
(def  length  (L)  (if  (Nul  L)  0  (+  1  (length  (tail  L)))))
(def  avg  (L)  (/  (sum  L)  (length  L)))
```

? (avg ‘(1 2 3 4 5)) /* Main expression */
this program is evaluated into 3. The variables are the parameters of the functions. They are local and with single assignment.

Fc-constructions
It is related to functional programming without variables. In this language a program is specified by a combination of functions while avoiding the handling of variables. It is inspired by FP [Backus 78]. The objects handled by the functions are divided into two categories:
- atoms: like numbers, characters, symbols etc...
- sequences (or lists): [x1, x2... xn] where each xi is an object.

The application of a function F to an object X is noted: F app X.
Certain functions known as " primitive " are predefined like the arithmetic operations such as +, -, *, /, the relational predicates, the operations of handling of lists (head, tail ...) etc.
Some constructions (or functional forms) makes it possible to combine functions (primitive or not) to form new functions. Among these forms one quotes:

- the composition of functions: (F . G) app x = f(g(x))
- construction: [f1, f2... fn] app x = [f1(x), f2(x)... fn(x)]
- the conditional one: (p? F: G) app x = { if p(x) then f(x) else g(x) }
- the mapping of a function to all the elements of a sequence:
  (Map F) app [x1, x2... xn] = [f(x1), f(x2)... f(xn)]
- etc...

Examples: Definition of the function ' Factorial':

```
Def Fact = (Zero?  ' 1: * .  [ id, (Fact .  Decr) ])
```

where id is the function identity: f(x) = x whereas Zero and Decr are defined as follows:

```
Def Zero = equal .  [id, ' 0 ]
```

with equal the predefined function testing the equality of its arguments. Then, function zero tests if its argument is equal to 0.

```
Def Decr = - .  [ id, ' 1 ]
```

Function Decr allows to decrement its argument.
Explanation:
Fact is defined as being a conditional statement. The latter tests if its parameter is 0, in which case it returns 1 otherwise combines the operation ' * ' with the result of the sequence ' [id, (Fact . Decr)] '. This sequence represents a couple of values: the parameter and the result of the composition of functions Fact and Decr.
One sees starting from the preceding examples that the programming of new function is done without the use of variables, but simply by combination of functions.

Oc-construction
It is related to object programming with classes. A class is a set of objects. Each class is defined by its attributes and methods (encapsulation). It must also specify the parent classes (inheritance). The objects exchanges information by means of messages. In this type of construction the single operation is sending of messages from an object to another. Certain objects are primitive, i.e. they are instances of predefined classes (like integers, strings,...). The language is inspired by SmallTalk and allows
dynamic typing and multiple inheritance. There is a predefined class (*object*) which is on the top of the hierarchy from which three classes derive : *type*, *action* and an *empty object*. *Type* is the parent of classes *integer*, *boolean*, *character*, *string* and *collection*. The objects of class *action* represent blocks of instructions. For example the following block: { i: = i/2 + j; j: = j * 10; k: =k+1 } is regarded as being an object to which one can send messages representing the various control structures (the While loop, the conditional if-then-else, etc...).
A sending of message can take one of the three following forms:
1. Unary message: if the called method does not have a parameter.
Example: 5 Fact; the receptor is the object 5
" Hello world. " print;
the receptor is the object “Hello world”
Fact should be defined in the class *integer* and print in the class *string*.
2. Binary message: in the case where the method is an operator (special sign) having only one parameter.
Example:
5 + 6; the receptor is object 5, the called method is ‘+’ with object 6 as parameter.
3. message with key words: if the method uses one or more key words corresponding to the parameters. Example:
5 pgcd: 15;
An_array swap: 3 and: 6;
Here an example of adding a method to the Integer class:

```
Integer AddMethod :
/ * To enrich the class Integer by the method Fact * /
{ Fact
{
{ return 1;  }    if:  (Receiver=1 Or :  Receiver=0)
else :  { return Receiver * (Receiver -1) Fact;  }
}
}
```

The body of the recursive method Fact is composed of a conditional statement (the block { return 1; } which one

sends the message ‘if <cond> else <bloc>’ with the following semantic : if the receiver of the
message is value 1 or 0 then return value 1 as result, otherwise (the eceiver is an integer N > 1) return N * Fact(n-1) as result.

Of-construction
It is related to object programming with frames. This type of programming is directed towards the representation of knowledge. It uses the concept of ' Frame' which makes it possible to model a knowledge through a set of objects dependent between them by semantic links (' kind-of', ' is - a', …). The definite language is inspired by KRL, FRL and Shirka. Each frame is at the same time an instance and a prototype allowing to generate other instances. It is consisted a set ' of attributes' which can each one, to have several 'facets'. There are declaratives facets (representing data) and procedural facets (called 'reflexes' or demons) which are: “if_needed” , “if_possible”, “if_addition” and “if_remove”
Here an example of use of this type of conventional programming language (calculation of the factorial)

```
Frame Fact
        {
        Number :        Type integer
                        If_needed Read( );
        Facto :         if_needed
Compute_Facto( );
        };

Function Read ( ) : Integer;
        Integer value;
        {
                Output(“Enter an integer : “);
                Input(value);
                Return value;
        };
Function Compute_Facto( ) : Integer;
        Integer x , y ;
        {
                x := Read_Z(Fact* , Number);
                if x = 0 then Return 1
                else    {
                        Write(Fact * , Number ,
x–1);
                        y := Read_Z(Fact * ,
Facto);
                        Return (x * y);
                }
        };
Function Main( ) : Null;  /* main program */
        Integer Res;
        {
                Res := Read_Z(Fact , Facto);
                Output(“Result = “, Res);
        };
```

The instruction Read_Z(Frame*, Attribute) makes it possible to find the attribute value of a frame by using a traversal of the ineritance graph according to order Z.

Oa-construction
In this formalism, the resolution of a problem consists in making cooperate a set of active and autonomous objects, communicating between them by synchronous and/or asynchronous messages in a parallel execution environment. It is related to object programming with actors. An actor is an entity defined by its state and its script. It must also specify the delegation (inheritance), the continuation and the type of synchronization (synchronous / asynchronous). In our Oa-language, the actors are created and destroyed dynamically. The script defines the behavior of the object according to events transmitted by others actors. The script is described by control structures allowing to express them in a structured way. We can cite 'If Then Else', 'While' and 'Case' with the simple reading, writing and assignment operations. The actors exchange information by messages which are actors themselves. When one sends a message to an actor one can possibly specify a list of continuations which represent in fact the actors who will receive the result of the message. The general syntax of a sending of message is as follows:

Name-method [list of parameters], CONT [list of actors] → An Actor

In the example below, one gives a solution to the problem of the factorial of a number with our language:

```
Actor Fact
Script:
  OBS  met[ n(Integer) ] , CONT[u]
  begin
        [ac , se]           /*  local variables */
        ifelse [n=0] →{ met[1] → u;
        ac := Create(script: OBS met[h(integer)] begin
met[n*h] →u end);
        se := Create(Fact);
        met[n-1],CONT[ac] → se
                                 }
  End

Actor Message
Script:
  OBS met[n(Integer)] begin  Affich[“The result is : “];
Write[n]→Integer end

Program
DATA: R(Message) , F(Fact)
SCRIPT:
  OBS test-fact[]
  Begin
        R := Create(Message);
        F := Create(Fact);
        Met[3],CONT[R] → F
  end
```

3. THE PHASES OF ECOLE

Construction phase
This phase offers tools to built constructions of different types. On the one hand, we have an assisted mode in which we help the user to built constructions by offering him many schemes. For example, in the case of a object language by class (Oc), the user has a dialogue box to create a class. This dialogue box enables you to insert in the editor a text constituting the definition of a new class. This definition is carried out in an interactive way thus facilitating the message calls. As options of the dialogue box one can quote a name of the class, names of the variable super-classes of class, Variables of instances, etc. One can also check the syntax of the body of the method and activate the insertion mode with indentation. For each proposed construction, we offer a fully documented integrated environment.

Transformation phase

A transformation of programs is a source-to-source modification. A transformation must be comprehensible to ensure equivalence and precise to allow an automation. We have developed compilers in order to achieve the following transformations :
- from a logical language to a clausal form
- from a functional language with and without variables to a λ-calculus.
- from object languages to adequate internal forms.

Evaluation phase

We mean by evaluation the following tasks :
- Empirical tests : to run Ox-constructions with x in [f, a, c]
- Reductions : to run Fx-constructions with x in [c, v].
- Proofs : to run L-constructions.

For this purpose, we have implemented for all the x-languages (x in { L, Fv, Fc, Oa, Oc, Of}) :
- interpreters in order to unroll object constructions,
- reduction engines in order to unroll functional constructions,
- provers to unroll logic constructions.

For the resolution methods, two classes of algorithms are developed according to the traversal is in *breadth* or *depth*. For the reduction engines, several execution schemes are considered :
Call by value, *by name* and *by need* which lead evidently to the same normal form. (Thesis of Church-Rosser). More information about these methods are in [1, 2, 13].

4. ARCHITECTURE OF ECOLE ENVIRONMENT

The ECOLE environment is very simple. Any user who wants to work with this system begins by choosing a construction type. Then, he can use the construction environment in order to built his program (x-construction). In logic part, he may undertake the following tasks : transformation to clausal form, proof by several mechanisms. In functional part, he may undertake the following tasks : transformation to λ-calculus, reduction by several approaches. In object part, he begins by choosing the type of object language (Class, Frame or Actor) before to undertake the following tasks : transformation to internal form, interpretation.

5. WORKS REALIZED IN ECOLE

Work began on ECOLE in earlier 1997 and is in its final phase. All the components of ECOLE are implemented by students in their last year of undergraduate. For each application, we have developed a friendly user interface with Delphi.
We have developed syntactic editors for all the constructions suggested including modules of indentation, compression and decompression.
A full complement of interactive debugging facilities are available for each construction type with tracing, breakpoints and animation of the execution. For each x-construction, we have added tools which help the curious user to unlock the mystery of each programming type by showing the work undertaken by the interpreters, the provers and the reduction engines.
An environment of construction and correction of L-constructions has been developed. It consists in assisting the user to write and unroll logic constructions. An indentation procedure is developed in order to make the construction clear and easy to read.
Tools are developed in order to assist the user for constructing, indenting and unrolling F-construction on data. The pedagogical aspect to explain the different types of evaluation is also approached by giving the work made by the reduction engine.
We have developed an environment of construction and correction of Oc-construction. It consists in conceiving an editor, an interpreter and pedagogical tools. The editor offers tools for writing Oc-construction. The interpreter allows the unrolling them in continuous or step by step mode.
Tools are offered to build object constructions by class and unroll them.
Several techniques of resolution for the L-constructions are implemented in ECOLE. For all the resolution methods developed, the user has an hypertext (offered by the facilities of Delphi) presenting theses methods. In addition, the user may follow the resolution step by step in order to understand the resolution mechanism. More details are given below.
A simulator is written in order to unroll Oa-constructions on a single processor microcomputer. The execution is animated by stopping it at any time to show the active actors, the passive ones and also the waiting actors. What allows us to see the states of all the actors with the continuations and the return points in the scripts. A curious user see thus the inner work of object programming by actors.
A complete list of the works realized in ECOLE is in reference from 9 to 16.

Abstracts

We give below brief abstracts of the various works realized in ECOLE, each one represents a contribution of a student in its graduation final phase .

(i) Construction of functional programs with variables & their translation into equivalent λ-expressions.

In this work we will be interested in the project part related to the realization of an environment of construction and transformation of functional algorithms with variables. It is a question of defining a functional language by determining its syntax and its semantics, then to develop the following tools:
- syntactic editor to assist the construction of the programs.
- a translator towards a pivot language (λ-calculus).

(ii) Realization of an environment of construction and transformation of logical algorithms.

It is a question of conceiving an environment of construction, correction and transformation of logical programs belonging to the language LOGFC which we conceived beforehand. The generated code is a clausal form, which constitutes the pivot code for the mechanism of resolution applied during the interpretation of the logical programs. The edition of a program LOGFC could be carried out by an assistance system.

(iii) Automatic demonstrators for a prototype of logical language.

This work consists to develop a demonstrator of theorems for the execution of logical programs (in clausal form) by adopting various strategies based on the Robinson resolution and to integrate it in the application already carried out. The application is equipped with a tool which makes it possible to transform any formula of the first order predicates logic towards an equivalent clausal form.

(iv) Machine with reduction for a functional language.

It is question here of developing a machine for the evaluation of functional programs (in the form of λ-expressions) by adopting various strategies of Beta-conversion (evaluation by value, by need, lazy...) and of integrating it in the application already carried out. The application is equipped with a tool which makes it possible to check the equivalence of mathematical functions written in usual form (Interconvertibility of λ-expressions)

(v) Ecolo: an environment of programming object by class.

It acts to conceive a language purely object while taking as a starting point existing languages, provided at least with the following characteristics:
Encapsulation: regrouping data and methods under the same entity (object).
Inheritance : transmission system of properties between classes.
Polymorphism: dynamic link of the methods (principle of the virtual methods) .
Generic classes: definition of the type of the object independently of its implementation.
Dynamic typing: possibility of creating new types during the execution of a program.
Work consists in developing an environment of construction of object programs as well as an interpreter able to solve the techniques of the pure object programming (characteristics quoted above)

(vi) Environment of construction and interpretation for a prototype of object language by ' Frame'

In the programming by ' frame', the graph of inheritance is typically dynamic. Also, contrary to the programming by class, each object (frame) is regarded at the same time as authority and generator of other objects. This part consists with the design of an object language typically by ' Frame' while emphasizing the various concepts of this programming method. The main goals are :

- implementation of these concepts with an aim of the development of an interpreter.
- the development of a teaching environment of programming facilitating the construction of the programs by ' frames' of the language suggested and showing the various aspects related to this type of programming.

(vii) Environment of construction and interpretation for a prototype of object language by ' Actor'.

In the programming by 'actors', the objects (actors) achieve independent tasks (scripts) et are communicated by sending messages. What implies a parallelism on the level of the treatments. This come back to

- to conceive typically an object language by 'Actors' while emphasizing the various concepts of this programming method.
- to implement these concepts with an aim of working out an interpreter simulating the parallelism.
- to develop a teaching environment of programming facilitating the construction of the programs by ' Actors' of the language suggested and showing the various aspects related to this type of programming.

(viii) Environment of functional programming without variable.

A typically functional language is conceived, inspired of the language FP of Backus, in which the programming is done without the use of variables. Two aspects are developed:

- passage towards a pivot language: combinatory logic
- writing of a reduction machine to evaluate the combinative forms.

The product is designed of such kind so that it is used at teaching ends.

Some screen dumps

We present below two applications in ECOLE, the first deals with the object programming by class (ECOLO), the second with the reduction engines for functional programs (LAMBDA).

(i) ECOLO : an environment of object programming by class.

We have developed an efficacious tool for the object programming by class named ECOLO. As any integrated programming environment, it allows to write Oc-constructions and to execute them. This environment uses a text editor allowing the classical operations on files and offering edition and navigation tools in the text. Facilities to move and size windows are also possible. The environment can be entirely personalized at the convenience of the user. Several bars are available on the welcoming screen making the environment very convivial and easy to use. We have the following :
Title bar : gives the name of the file present in the text editor. It holds the standard system menu with the operations Restore, Move, Size, Minimize, Maximize, etc.
Tools bar : holds buttons to make visual and then more practical the choice of commands.
Command bar : holds buttons for the edition.
Status bar : informs us on the actual state of the file.
Menu bar : holds the sub-menus File, Edition, Execute, Tools, Help
The sub-menu Execute is the most important and consists of the following items :
- Execute : executes the current edited file.
- Syntax : verifies only the syntax of the current file.
- Execute for the pedagogical explorer : allows to visualize graphically the sending of messages.
- See the error message : displays/hides the last error message.
- See the output screen : allows to see the last output screen.

A set of dialogue boxes enriches ECOLO with several others features. We can cite Add/Modify a tool, Definition of methods , Pedagogical explorer, etc. ECOLO owns a very important tool : the pedagogical explorer which offers means to control the execution of an Oc-construction. What allows us to understand more easily the concepts of object programming by class. The pedagogical explorer holds the following parts :
I. SEARCHING OF METHODS
This part shows the inheritance graph and allows to see the evolution of searching process in this graph. It visualizes also all the information related on the method being searched. Two panels are available : the one for the information on the method and the other for the control operations. An inheritance graph is also drawn in this part. We describe briefly this below :

Information panel

. Value : gives the value of the receptor if it is a constant, its name if it is an identifier. In the case of class message, it gives the name of the receptor class.
. Class :gives the name of the receptor class. This class appears as a leaf in the graph.
. Selector : gives the selector of the searched method in order to execute the current message.
. Arguments :gives the list of arguments related to the current message. This area has no effect if it is question of an unary message. Information on the arguments are known once the method is found, because the arguments are treated only at the moment of calls.

Control Panel

. Button 'Step by Step' : Allows to follow step by step searching in the partial inheritance graph.
. Button Next Message : goes to the next message
. Button Stop : Interrupt the execution of the current Oc-construction and stop the pedagogical explorer.
. Button Properties : displays information related to the selected class.
. Button Information : allows to display or hide the information panel.

Inheritance graph

The partial inheritance graph about the searching mechanism of the current method is drawn. The root of this graph is the class 'Object', leaf is the class of receptor. A node holds the code of the class. In order to see its
name or its descriptive if suffices to click the appropriate buttons.

II. IMAGE MEMORY

This part consists in the visualization of the content of different champs of data structures used by the compiler to represent the various objects. It allows also to display the graph of instances and the one of classes.

Control panel

. Button Next message and Button Stop as previously.
. Button class : displays the graph representing the directory of classes and hides the one of instances.
.Button instance : displays the graph representing the directory of instances and hides the one of classes.

Graph of classes

The graph representing the directory of classes is drawn. It suffices to click the button left or right to obtain its descriptive and all the relative information respectively.

Graph of instances

The graph representing the directory of instances is drawn. Additional information are given by simple clicks

III. CODE SOURCE

It consists to visualize the code source of the Oc-construction being unrolled in order to well situate the current message.

Text Source

A generic window in which you can only read your Oc-construction.

Control panel

. Buttons ‘Next message’ and ‘Stop' as before.
. Button ‘Execute until cursor’ : allows to continue the execution until a stopping point is reached.
. Button ‘Output screen’ : uses the output screen to see the current results..

Status line

A status line allows you
- to locate the current message in the text by following the progression indicator
- to define a stopping point to reach it directly by using the control panel described above.

A set of marks allows us to show a progression indicator, the place where the execution is stopped , a valid stopping point and a non valid one.

IV. SYMBOLIC STACK

It consists to see the call stack at a given time through a drawing.

Control panel

with the same buttons described above.

Graphical stack

The graph symbolized the call stack at a given time is drawn. You can select one method in the stack to obtain information on it in the information panel situated to the left of the stack.
More information about this work is in [11]. Figure 1 gives some screen dumps. Figure 1.a presents the welcoming screen. Figure 1.b and 1.c show some states of the pedagogical explorer : inheritance graph and graph of instances (memory dump) respectively.

(ii) LAMBDA : Reduction engine

We have also developed an engine, named LAMBDA, in order to achieve reductions on λ-calculus expressions. It offers a spectrum of services to the user. Basic debugging facilities are provided in order to follow thoroughly the reduction with various methods. LAMBDA includes also an independent environment for the equivalence of functions with its own debugging facilities. We give bellow some technical information on the methods implemented. The following execution schemes are considered :

Call by value

The substitution to realize at each step is the innermost and the leftmost function. It consists of calculating sequentially the values of arguments before the call to the function.

Call by name

The leftmost and the outermost among the functions is made first. This mechanism do no consist in the evaluation the arguments at the moment of call but rather to conserve the function which permits to evaluate this argument.

Call by need

It is a refinement of the ‘call by name’ method. The arguments are evaluated only one time. The technique is then doted by an additional task allowing to detect the first evaluations and save the results already calculated.

It is evident that all these techniques lead to the same normal form (Thesis of Church-Rosser). For the three execution schemes the evaluation of a λ-expression is in weak head normal form, what means that the reduction can stop even if remains radicals. Recall that a λ-expression is called in weak head normal form if and only if it is of the form (FA1A2.....An). F is being a variable, a constant, a predefined function or a λ-expression and (FA1A2.....An) is not a redex. We have implemented the technique called graph reduction suggested recently. It is based on axioms and transformation rules with the consideration of the intrinsic properties of expressions. In this technique, the internal form, i.e. the code to reduce, is represented as a graph. The application of an axiom involves the reorganization of the graph. The sub-graph representing the redex is replaced by the one representing the contractum. The representation by graph is based on the pointers and avoids thus the reevaluation of a same sequence. Instead of duplicate the graph of the sequence , it suffices only to duplicate the pointer towards this graph. For more details of implementation see [12]. The problem of equivalence between mathematical functions is not decidable. In spite of the whole consistence of the λ-calculus, it does not permit the interconvertibility of two expressions. We have elaborated a tool to verify automatically the equivalence of two mathematical functions for an important class of functions related to the arithmetical operations algebra. The equivalence is defined by a congruence operation.
Figure 2 gives some screen dumps. Figure 2.a shows a functional program compiled with success. Figure 2.b presents the work accomplished by the reduction engine by showing : source program, λ-expression and execution trace. Figure 3.c shows a step by step execution.

6. EDUCATIONAL PURPOSE

ECOLE may be used for pedagogical purposes at two levels : Firstly, for students having modest experience in classic programming. Indeed, ECOLE is fully documented on each construction type and on mathematical concepts behind these languages as λ-calculus or first order predicates logic.

ECOLE is very simple to use. Thanks to its syntactic editors, it permits the writing of x-constructions syntactically correct. And thanks to its interpreters, it allows the unrolling of constructions.
Secondly, for the students having already a good knowledge of programming and who want to know more. Indeed, ECOLE proposes more advanced features. It explains each type of programming thanks to simple languages considered, the work undertaken as well by transformations and compiler for each type as by interpreters, engines and simulators considered. ECOLE permits thus to see the unrolling of various constructions step by step in order to understand the mechanisms of each programming type.
Besides providing the pedagogical objectives, the outcome of this project will permit to solve a certain number of problems known difficult. In particular that of the equivalence of function.

7.CONCLUSION

It is certain that ECOLE is a good synthesis on non procedural languages and object ones. We think that the aimed software will be of a great interest for teaching the art of programming for students having a first experience and who want to learn more. This synthesis has allowed us to clarify many concepts and technical details. In particular, how writing compilers for object languages in all its forms and for logic and functional ones. We thus revealed all the mysteries of the theorem demonstrators and reduction machines.

Acknowledgments
We would like to express special thanks to all the students who took part in ECOLE for their important contribution in the implementation of different components of ECOLE. A major expression of thanks must go to the Professor Gérard Levy for his advice and helpful comments and suggestions

References

[1] Peyton jones. "Mise en oeuvre des langages fonctionnels de programmation".Masson 1990.
[2] J. Hogger. "Programmation en logique". Masson 1988.
[3] J. J. Meyer "Initiation à la POO à travers Turbo Pascal". Radio 1990.
[4] 'The Object-Oriented Pre-Compiler, Programming Smalltalk-80 Methods in C Language' - Brad J.Cox.ACM SIGPLAN Notices, 18(1) :15-22, 1983.
[5] 'Les langages à objets, Langages de classes, langages de frames, langages d'acteurs' - Gérald Masini & Amedeo Napoli & Dominique Colnet & Daniel Léonard & Karl Tombre.InterEditions, Paris, 1990.
[6] 'A Survey of Object-Oriented Concepts' - O.M.Nierstrasz.Active Object Environnements, Centre Universitaire d'Informatique, Université, de Genève, 1988.
[7]. D.E Zegour. " ECOLE : présentation". Rap. Interne. INI, 1997.
[8] D.E Zegour, G. Levy, CONCORD:an environment of CONStruction, CORrection anD transformation of algorithms, Information and Software Technology 40 (1998), 281-290.
[9] S. Dellys, Conception d'un langage fonctionnel et sa transformation vers λ-calcul, Mémoire d'ingénieur d'état en informatique, INI, 1996.
[10] H. Djouabi, Conception d'un langage logique et sa transformation vers la forme clausale, Mémoire d'ingénieur d'état en informatique, INI, 1996.
[11] M.Smati , S. Bordji, Conception d'un langage objet pur par "classe", Mémoire d'ingénieur d'état en informatique, INI, 1997.
[12] D. Boukhelef, Machines à réduction pour un langage fonctionnel de programmation, Mémoire d'ingénieur d'état en informatique, INI, 1997.
[13] A. Allaloui, D, Mallem, Démonstrateurs automatiques pour un prototype de langage logique, Mémoire d'ingénieur d'état en informatique, INI, 1997.
[14] W. Lassouani, Conception d'un langage objet pur par "acteur", Mémoire d'ingénieur d'état en informatique, INI, 1999.
[15] B. Behlouli & F. Abazi Conception d'un langage objet pur par "frame", Mémoire d'ingénieur d'état en informatique, INI, 1998.
[16] R. Delli & D. Saidani Conception d'un langage fonctionnel sans variables", Mémoire 'ingénieur d'état en informatique, INI, 1999.

Received: Apr. 2006. Accepted: Oct. 2006

.