記 号 処 理　27－1
（1984. 3. 12）

ENVIRONMENT PROBLEMS IN FORMULA MANIPULATION SYSTEMS

Tateaki Sasaki*)　and　Fumio Motoyoshi**)

*) The Institute of Physical and Chemical Research
Wako-shi, Saitama 351, Japan

**) Electrotechnical Laboratory, The Ministry of ITI
Niihari-gun, Ibaraki 305, Japan

ABSTRACT

In formula manipulation systems, evaluation of the symbol (i.e., updating the value expression of the symbol), simplification of the expression, and reordering of the symbols in the expression are basic processes to perform. When the system performs these processes automatically, which is necessary in application-oriented systems, the system must check the environment to the expression. Checking environment is rather expensive if performed elementarily, hence we call automatic symbol evaluation, reordering and simplification the environment problems. In this paper, we introduce an environment number to each of the evaluation, simplification and reordering environments and propose a method for controlling each environment efficiently and simply.

§1. Introduction

The term "environment" is used in several different meanings in computer science. The most popular one is the programming environment. A less famous but more technical one is the environments for the symbols in Lisp [1]. In this paper, we propose new concepts of environment in the context of formula manipulation by computer. Our environments are concerned with the evaluation of symbols, simplification of expressions, and reordering of symbols in an expression.

As we will explain in §2, the environments in formula manipulation systems are changed by the user consciously or unconsciously as the calculation proceeds. The formula manipulation system must check the changes of environments if the system supports the symbol evaluation, simplification, and reordering automatically, which is necessary in application-oriented systems. Checking the changes of environments is, however, rather expensive if performed elementarily, as we will explain in §2. We, therefore, call the automatic environment control the environment problem in formula manipulation systems.

In §3 ~ 5, we give a solution to each environment problem by introducing an environment number for each of the evaluation, simplification, and reordering environments. The introduction and usage of the environment numbers are very simple and almost the same for all the three environment problems.

§2. Environment problems in formula manipulation

Before explaining the concepts of environment in formula manipulation,

let us explain the basic process of formula manipulation by computer. We may divide the process of formula manipulation into the following five steps: 1) input, 2) symbol evaluation, 3) computation (in a narrow sense), 4) simplification, and 5) output. We explain the second, third, and fourth steps by an example:

```
1:  START ;
2:  A := s**2 ;
3:  B := t**2 ;
4:  s := x**2+y**2 ;
5:  t := x**2-y**2 ;
6:  RULE x**3 -> 0 ;
7:  C := A+B ;
```

The 6th line defines a left-to-right rewrite rule. Consider the processing in the 7th line. When "A+B" is input, the values of the symbols A and B are evaluated first as follows (symbol evaluation step):

$$A \Longrightarrow s^2 \Longrightarrow (x^2+y^2)^2 = x^4 + 2x^2y^2 + y^4,$$

$$B \Longrightarrow t^2 \Longrightarrow (x^2-y^2)^2 = x^4 - 2x^2y^2 + y^4.$$

Second, the values of A and B are added (computation step):

$$A + B \Longrightarrow 2x^4 + 2y^4.$$

Third, the rewrite rule is applied to the above result (simplification step):

$$2x^4 + 2y^4 \Longrightarrow 2y^4.$$

This is the final expression of A+B and it is assigned to C.

As the above example shows, we assume in this paper that, except for symbols used as independent variables, any user-defined expression to be saved in the system is assigned to another symbol and saved in the VALUE cell of the symbol.

Now, we explain the environment problems by referring to the above example. By the evaluation of symbol s we mean updating the value of s. If s has no value then the evaluation of s gives the s itself. If s has a value then every symbol in the value is evaluated and substituted, as the above example shows. For a given expression, it is unclear which symbols in the expression have values and which symbols do not unless we make some check. The simplest method of the check is to scan the expression, which is rather expensive if the scanning is performed for every symbol in the input. Note that, in many cases, all the symbols in an expression are independent variables, and the check is unnecessary in such cases. The evaluation environment problem is to find an efficient method of controlling the evaluation environment.

By a simplification rule we mean a left-to-right rewrite rule as illustrated by the sixth line in the above example. In order to simplify an expression by a rule, it is necessary to scan the expression so as to find terms matching with the rewrite pattern in the rule. Pattern matching is a relatively expensive procedure. Furthermore, scanning the expression is repeated until no simplification rule applies to the expression. Hence, the simplification is usually quite expensive, and it is very desirable to find an efficient method of controlling the simplification environment. This is the simplification environment problem.

Many of the formula manipulation systems constructed so far contain some devices to avoid unnecessary checks for simplification. For example, in REDUCE [2], a flag of two states (TRUE or FALSE) is attached to the

expression showing whether or not the simplification rules have been applied to it. MACSYMA [3] also has a similar device. An idea we propose in this paper is a more advanced one.

In most formula manipulation systems, all the symbols in any expression are uniquely ordered and the internal representation of the expression is constructed on the basis of this ordering. When the ordering of the symbols has been changed, the ordering must be adjusted to the current ordering before the computation: expressions in different orderings are not able to be manipulated correctly by most arithmetic procedures. Therefore, if the system allows the user to change the ordering of the symbols, which is very common in formula manipulation systems, the system must check the ordering. An elementary method of checking the ordering is to scan the expression and investigate compatibility with the current ordering. However, scanning every expression before computation is wastefull, and the ordering environment problem is to find an efficient method of controlling the ordering environment.

Note that not every system suffers from the environment problems in the above sense. For example, in MACSYMA [3], the evaluation and simplification are controlled by the user and the system needs not check the evaluation and simplification environments. In REDUCE [2], the symbol reordering is allowed only in the output expression and not in the internal representation. Hence, there is no ordering environment problem in REDUCE.

§3. Evaluation environment number and its usage

The evaluation environment number (eval-env number) we propose in this paper is an integer specifying the eval-env uniquely. When an expression is assigned to a symbol s as its value, we attach the current eval-env number to the value expression. Then, by referring to the eval-env number attached with an expression, we easily know in which eval-env the value expression was evaluated. The details are as follows:

Definition and usage of the eval-env number

1) At the beginning of the computation: EVAL_ENV := 0 ;
2) When the eval-env is changed: EVAL_ENV := EVAL_ENV + 1 ;
   Here the eval-env is changed in the following cases:
   (a) when the value of the variable is changed (introduction of new independent variables does not change the eval-env),
   (b) when new simplification rules are defined,
   (c) when commands concerning to the evaluation method are issued (for example, commands for delayed evaluation or unevaluation).
3) Attach the current EVAL_ENV to the evaluated value when it is saved into the VALUE cell of a symbol s;
4) When the value of a symbol s is evaluated, the pair of the current EVAL_ENV and the evaluated value is saved into the SAVE_VALUE cell of s;
5) When we evaluate the symbol s, we first read out the content of the SAVE_VALUE cell of s and if the saved value was evaluated in the current eval-env then return the value, otherwise we read out the content of the VALUE cell of s and evaluate it.

We illustrate the usage of the eval-env number by an example:

```
1:  START ;                          /* EVAL_ENV = 0 */
2:  A := s**2 ;
3:  B := t**2 ;
4:  C := A**2+A*B+B**2 ;
5:  s := x+1 ;                       /* EVAL_ENV = 1 */
6:  t := y+2 ;                       /* EVAL_ENV = 2 */
7:  C := A**2+A*B+B**2 ;             /* EVAL_ENV = 3 */
```

In the 4th line of this example, the values of A and B (i.e., s^2 and t^2) need not be evaluated because A and B are introduced in the zeroth eval-env and the 4th line is in the same eval-env. On the other hand, in the 7th line, the values of A and B must be evaluated because the eval-env number here is 2 (the eval-env number increases to 3 at when the C is reassigned). Note that, in the 7th line, each of the symbols A and B is appearing twice but it is essentially evaluated only once because of the use of SAVE_VALUE.

§4. Simplification environment number and its usage

We divide the rewrite rules into two classes, single-function rewrite rules and others (for example, multi-factor/term rewrite rules). Here, by the single-function rewrite rule we mean that the left hand side of the rule is a single function of power one. For example,

```
sin(x+y) -> sin(x)*cos(y) + cos(x)*sin(y),
F(x**2+y**2) -> G(x)**2 + G(y)**2,
F(sin(x)*cos(x)) -> sin(G(x)) + cos(G(x))
```

are single-function rewrite rules, and

```
sin(x)*cos(x) -> sin(2*x)/2,
sin(x) + cos(x) -> SQRT(2)*sin(x + #pi/4)
```

are multi-factor/term rewrite rules.

The single-function rewrite rules are advantageous to process in that, if each function form in the input has been simplified, we need not apply the single-function rewrite rules any more until new rewrite rules are defined. This is because that the argument of the function is never mixed with other expressions. We must, of course, apply the single-function rewrite rules again if new rewrite rules are defined. Note that, the advantageous nature mentioned above does not hold for other rewrite rules. For example, the rule

```
sin(x)**2 -> (1-cos(2*x))/2
```

is applicable if expressions which have been already simplified to contain sin(x) are multiplied.

The simplification environment number (simp-env number) we propose in this paper applies to only the single-function rewrite rules, hence our solution to the simp-env problem is partial. We comment, however, that most simplification rules in application programs are single-function rewrite rules. In actual implementation of the simplification by rewrite rules, we had better classify the rules into more details. Then, we can perform the simplification more efficiently.

Definition and usage of the simp-env number

1) At the beginning of the computation: SIMP_ENV := 0 ;

2) When the simp-env is changed: SIMP_ENV := SIMP_ENV + 1 ;

   Here the simp-env is changed in the following cases:

   (a) when new simplification rules are defined,

(b) when commands concerning to the simplification method are issued (for example, commands for delayed simplification or unsimplification).

3) When the simplified expression is saved into the VALUE cell or SAVE_VALUE cell, attach the current SIMP_ENV to it;

4) When the function form in the input is evaluated, apply the single-function rewrite rules to it;

5) When the value expression is evaluated, apply the single-function rewrite rules only when the value was simplified in a simp-env which is different from the current one;

6) The simplification rules other than the single-function rewrite rules are applied after each computation step.

We illustrate the usage of the simp-env number by an example:

```
1:  START ;                                  /* SIMP_ENV = 0 */
2:  FOR ALL x,y RULE                         /* SIMP_ENV = 1 */
      sin(x)**2 -> 1-cos(x)**2 ,
      sin(x+y) -> sin(x)*cos(y) + cos(x)*sin(y) ;
3:  A := sin(u+v)**2 ;
4:  B := cos(u+v)**2 ;
5:  C := A*B ;
```

In the 3rd and 4th lines, the single-function rewrite rule is applied when the function forms sin(u+v) and cos(u+v) are evaluated, and another rewrite rule is applied after the computation of sin(u+v)**2 and cos(u+v)**2. On the other hand, in the 5th line, only the rule for sin(x)**2 is applied to the result of A*B and we need not apply the single-function rewrite rule.

§5. Ordering environment number and its usage

In this paper, by "A $>_o$ B" we mean that symbol A is of higher order than symbol B. In order to control the ordering environment (ord-env) simply, we specify the method of symbol ordering as follows: We assume that two commands ORDER and UNORDER change the ord-env. The ORDER command is used as

ORDER S1, S2, . . . , Sn ;

causing the new ordering

S1 $>_o$ S2 $>_o$ ... $>_o$ Sn $>_o$ any other symbols.

This usage is common in most systems. The UNORDER command is used as

UNORDER ;

clearing the current ord-env. That is, the UNORDER command recalls the previous ord-env, and this usage is advantageous to control the ord-env efficiently as we will see below.

The usage of UNORDER command is, although it seems to be rather peculiar, quite reasonable from the viewpoint of actual computations. Reordering symbols in formula manipulation is required mostly in the following cases: (1) some universal ordering is required which is different from the default one, (2) a local ordering is necessary in a procedure, and (3) a special ordering is wanted in the output expression. Among these cases, the reordering is essentially unnecessary in the case 1 because we may specify the universal ordering at the beginning of the computation. In the case 3, there is no ord-env problem because the reordered expression to be output is not used for later computation. Hence, mostly the case 2 is essential for the ord-env problem. In the

case 2, it is quite desirable to recall the previous ord-env when the procedure finishes. Thus, with the use of UNORDER command, we can avoid the reordering drastically in actual formula manipulation.

Definition and usage of the ord-env number

1) At the beginning of the conputation:

   ORD_ENV := MAX_ORD_ENV := 0 ;

2) Preserve the history of the ord-env as a list of

   (ORD_ENV . ordering rule);

3) When the ORDER command is input:

   ORD_ENV := MAX_ORD_ENV := MAX_ORD_ENV + 1 ;

4) When the UNORDER command is input, recall the previous ord-env from the history list and ORD_ENV := previous ORD_ENV ;

5) When an expression is evaluated and saved into the VALUE cell or SAVE_VALUE cell, attach the current ORD_ENV to the expression. When the expression is called and reevaluated, reorder the expression only when the ORD_ENV attached to it is different from the current ORD_ENV.

We illustrate the usage of the ord-env number by an example:

```
1:  START ;              /* ORD_ENV = 0 */
2:  ORDER X,Y,Z ;        /* ORD_ENV = 1 */
3:  A := X+Y ;
4:  B := Y+Z ;
5:  ORDER Z,Y ;          /* ORD_ENV = 2 */
6:  C := A*B ;
7:  UNORDER ;            /* ORD_ENV = 1 */
8:  C := A*B ;
9:  UNORDER ;            /* ORD_ENV = 0 */
```

The ordering is $X >_o Y >_o Z$ at the 2nd line, $Z >_o Y >_o X$ at the 5th line, and $X >_o Y >_o Z$ at the 7th line. In the 6th line, reordering of the values of A and B are necessary before the computation because they are defined in the first ord-env and the current ord-env number is 2. On the other hand, in the 8th line, the values of A and B need not be reordered (note that the values of A and B in the 8th line are the same as those defined in the 3rd and 4th lines, respectively).

References

[1] J. Moses, "The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem", SIGSAM Bulletin, 15, pp. 13-27 (1970).

[2] A. C. Hearn, "REDUCE user's manual", Version 3.0, The Rand Corporation, 1983.

[3] The MATHLAB Group, "MACSYMA Reference manual", 9th Version, Laborato for Computer Science, MIT, 1977.