Serdica
Bulgariacae mathematicae publicationes

Сердика
Българско математическо списание

SERDICA — Bulgaricae
mathematicae publicationes
19 (1993) 36-44

THE *W* LANGUAGE AND ITS APPLICATIONS TO THEOREM PROVING

MAGDALINA TODOROVA

ABSTRACT. The paper presents a brief description of the W language. The way of using it to prove theorems is illustrated too. Moreover it has been shown that this language can enable us not only to prove the validity of a certain theorem but also to find out the preconditions for which it holds.

***W*: Equational Programming Language of Logic Style.** The *W* language implements an approach of integrating functional and logic programming styles [3]. It is an equational programming language which accomplishes generalization through:

– replacing functional terms by words of a context-free language;

– replacing rewriting by narrowing.

Generally speaking a program written in the *W* language is a system of equations since in the equational style each function is defined by one or more equations. Usually equational interpreters function as functional term transformers while programs written in the *W* language can describe string transformations (in particular functional term ones). The structure of these strings is described through a context-free grammar. The known equational interpreters are used to evaluate only terms which have no variables. The *W* language allows both functional and logic evaluation of words of a context-free language. Therefore the *W* language is called an equational programming language in a logic style.

The generalizations made in the equational programming style enable us to use the *W* language for evaluating words whose free variables are functions which is not the case with pure logic languages.

The programs written in the *W* language involve syntax rules and equations. All variables used in the equations are defined in the variable declaration part. The Sample1 program [4] is an example of a program written in the *W* language.

```
Transformer Sample1:<expr>;

   type
      <expr>::= <nat> | <bool>;
      <nat>::= <func_name>'('<obj>')' | <nat>'+'<nat>| '0'| '1';
      <bool>::= 'eq('<nat>','<nat>')' | 'true' | 'false';
      <obj>::=  '['<obj>','<obj>']' | 'nil';
      <func_name>::= 'len' | 'depth';

   var
      <x>, <y>, <z> : <obj>;
      <m>, <n> : <nat>;
      <fn> : <func_name>;

   begin
T1    'len(nil)' ==> '0';
T2    'len(['<x>','<y>'])' ==> 'len('<y>')+1';
T3    'depth(nil)' ==> '0';
T4    'depth(['<x>','<y>'])' ==> 'depth('<x>')+1';
T5    'eq(0, 0)' ==> 'true';
T6    'eq('<m>'+1,0)' ==> 'false';
T7    'eq(0,'<n>'+1)' ==> 'false';
T8    'eq('<m>'+1,'<n>'+1)' ==> 'eq('<m>','<n>')'
   end.
```

The type declaration part consists of syntax rules and it assigns a context-free grammar. The latter defines words which can be evaluated through Sample1. The axiom `<expr>` of this grammar is given in the program heading immediately after its name Sample1. The sequences of terminal symbols are put in quotes. Variables are described in the variable declaration part of the program. Finally the program involves equations (rewriting rules). Some variables are given in the equations. Each equation is numbered so as to avoid ambiguity.

Each word of `<expr>` type can be evaluated logically by the Sample1 program. The logic evaluation is reduced to a functional one with words having no variables. The leftmost and outermost replacement strategy is used for evaluating words [4].

Examples.

(i) If the Sample1 program is run for the input word

```
'depth([ [ nil, [nil, nil ]], nil ])'
```

then the latter is reduced to the word `0+1+1`, i.e. to 2 (see Fig.1).

```
'depth( [ [nil, [nil, nil ] ], nil ])'
                ↓ T4
 'depth( [ nil, [ nil, nil ] ]) + 1'
                ↓ T4
       'depth( nil ) +1 +1'
                ↓ T3
             '0+1+1'
```

Fig 1.

Arrows are followed by the numbers of the equations applied to the current word. The evaluation is functional since the evaluated word has no variables.

(ii) If the Sample1 program is run for `'eq (len('<z>'),0+1+1)'`, then this word is reduced to the following list:

```
(     false  if  z = nil,
      false  if  z = [ x, nil ],
      true   if  z = [ x, [ x1,nil ] ],
      false  if  z = [ x, [ x1, [x2, y ] ] ]        )
```

where `z`, `x1` and `x2` are variables of `<obj>` type. This reduction is illustrated in Fig. 2 and there the evaluation is logic.

From the examples given above it follows that the output of a program written in the W language can be one or a number of ground or nonground words. Also this output can include some substitutes of the variables of the input word. A detailed description of the W language is given in [5] and [7].

The programs written in the W language are called also generalized transformers. They are described and proved in [6, 7].

Moreover not every generalized transformer can ensure an appropriate program environment. For example, there exist some generalized transformers with which some of the words of the context-free language generated by the transformer grammar have several different normal forms. It is very difficult to ground the choice of the evaluation strategy with such transformers. For that reason we shall consider only confluent generalized transformers [6, 7].

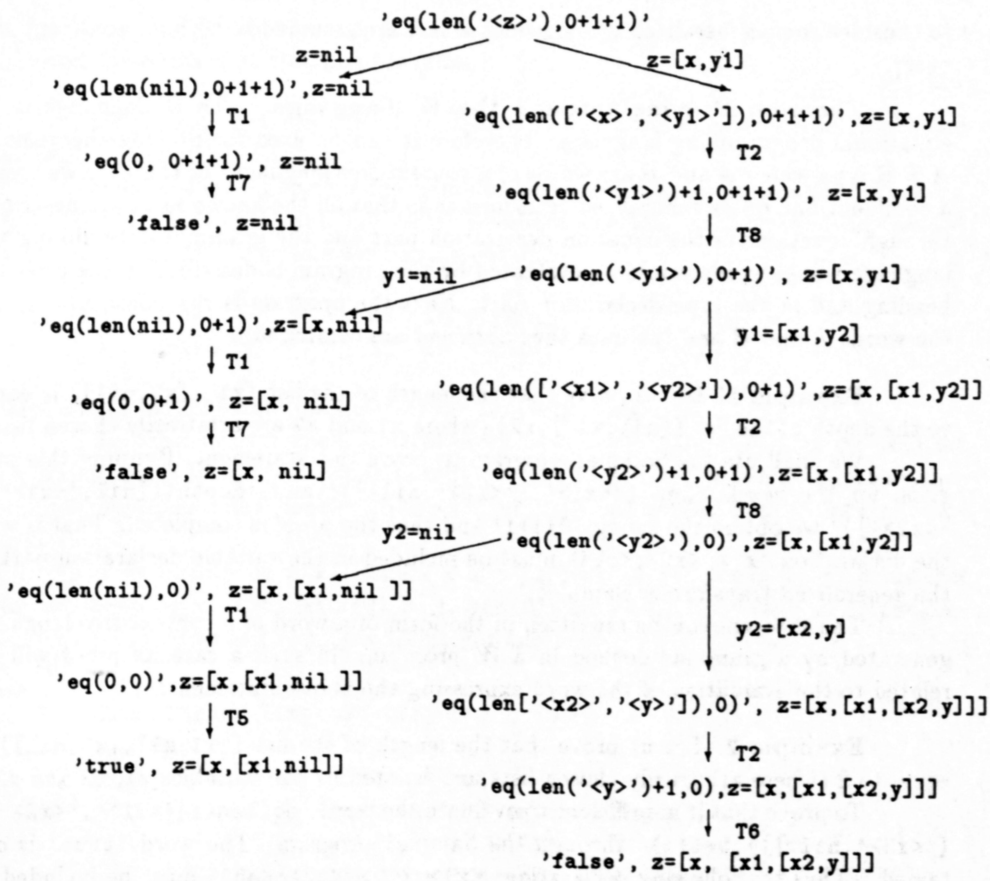

Fig. 2.

Some words generated by a confluent generalized transformer may have a normal form which cannot be found by applying the leftmost and outermost replacement strategy. The process of evaluating such words becomes infinite in practice. For that reason we assume that the confluent generalized transformers are outer [6, 7]. That property enables us to find out the normal form of words of that type.

It has been proved [1, 2] that narrowing is complete with confluent equational theories having the property of termination. If a confluent generalized transformer is characterized by an outer property, then it possesses the property of termination as well, which is proved in [6, 7]. In order to verify whether a certain generalized transformer is confluent and outer, some criteria have been found out in [7]. If we check Sample1 for these criteria, we shall see that these properties are typical of it. Next we are going

to consider some generalized transformers which are assumed to be both confluent and outer.

Theorem Proving through the *W* Language. The *W* language is an equational programming language. Therefore it can be used for proving theorems of $A = B$ type where A and B are words of a context-free language. In this case we create a confluent and outer generalized transformer so that all the known facts are described through equations in the equation declaration part and the grammar determining the language whose words would be evaluated by the program is described in the program heading and in the type declaration part. Next the program is run consecutively for the words A and B and the data thus obtained are compared.

Example 1. Let us prove that the length of the list `[x1, [x2,nil]]` is equal to the depth of the list `[[nil,x1 ],x2]` (where `x1` and `x2` are arbitrarily chosen lists).

We shall use the Sample1 program to prove this statement. Running this program for the words `'len(['<x1>',['<x2>',nil]])'` and `'depth([[nil,'<x1>'], '<x2>'])'` we obtain the string `'0+1+1'` and thus the proof is completed. That is why the declaration `<x1>, <x2>:<obj>` must be included in the variable declaration part of the generalized transformer Sample1.

The theorem can be rewritten in the form of a word of a context-free language generated by a grammar defined in a *W* program. In such a case its proof will be related to the evaluation of the word expressing the above statement.

Example 2. Let us prove that the length of the list `[[x1,x2],[x3,nil]]` is equal to 2 (where arbitrarily chosen lists are denoted by the variables `x1`, `x2` and `x3`).

To prove that it is sufficient to evaluate the word `'eq(len([['<x1>','<x2>'], ['<x3>',nil ]]),0+1+1)'` through the Sample1 program. The word `'true'` is obtained. (Thus the following declaration: `<x1>, <x2>, <x3>:<obj>` must be included in the variable declaration part of Sample1).

Moreover, one can also find out the preconditions for which a certain statement holds.

Example 3. When is the depth of list z equal to 2?

Let us evaluate the word: `'eq(depth('<z>'),0+1+1)'` by means of Sample1. The evaluation results in the following list:

```
(       false   if      z=nil,
        false   if      z=[ nil, y ],
        true    if      z=[ [ nil, y1 ], y ],
        false   if      z=[ [ [ y3, y2 ], y1 ] ,y ]     )
```

where `y1`, `y2` and `y3` are variables of `<obj>` type and they are generated by the language interpreter. From the evaluation it follows that this statement will be true if `z` is equal

to `[[nil, y1], y]` where `y` and `y1` are arbitrary lists. The evaluation includes also the cases for which that statement is false.

Example 4. Is there a function defined in the Sample1 program for which the list `[nil, [nil,nil ]]` is equal to 2?

To prove that let us run the Sample1 program in order to evaluate the word `'eq('<fn>'([nil, [nil,nil ]]),0+1+1)'` where a certain function name is denoted by the variable `<fn>`. The following list is obtained:

```
(       true            if  fn = len,
        false           if  fn = depth          )
```

which shows that the `len` function has the property mentioned.

This example is a good illustration of how the *W* language may be used for evaluating words whose free variables may be functions. The generalized transformer Sample2 described below enables us to add, multiply and compare equal natural numbers.

Let us denote by `s(x)` (where `x` is a variable of the `<nat>` type) a natural number following `x`. Thus let `s(0)` be `1`, `s(s(0))` be `2`, etc. It is easy to extend the Sample2 program so as to apply it to integer numbers.

```
Transformer Sample2: <T>;
   type
      <T>::=    <nat> | <bool>;
      <nat>::=  <func_name>'('<nat>','<nat>')' |
                's('<nat>')' | '0';
      <bool>::= 'eq('<nat>','<nat>')' | 'true' | 'false';
      <func_name>::= '+' | '*';
   var
      <x>, <y> : <nat>;
      <f>, <g> : <func_name>;
   begin
      '+(0,'<x>')'  ==> <x>;
      '+(s('<x>'),'<y>')' ==> 's(+('<x>','<y>'))';
      '*(0,'<x>')' ==> '0';
      '*(s('<x>'),'<y>')' ==> '+(*('<x>','<y>'),'<y>')';
      'eq(0,0)' ==> 'true';
      'eq(s('<x>'),0)' ==> 'false';
      'eq(0,s('<x>'))' ==> 'false';
      'eq(s('<x>'),s('<y>'))' ==> 'eq('<x>','<y>')'
   end.
```

The generalized transformer Sample2 is confluent and outer. Now let us use it to prove some simple statements.

Example 5. Let us prove that the root of equation `x+1=2` is equal to `1` (where `x` is a variable of the `<nat>` type). To prove that let us evaluate the word `'eq(+('<x>',s(0)),s(s(0)))'` by the Sample2 program. The evaluation results in the following list:

```
(       false           if  x=0,
        true            if  x=s(0),
        false           if  x=s(s(0)),
        false           if  x=s(s(s(x3)))       ),
```

where `x3` is an arbitrary variable of the `<nat>` type (the type of the variable `x`) and it is generated by the language interpreter.

Example 6. Let us prove that there exists no natural number which can be a root of equation `x+3=1`.

Let us run the Sample2 program so as to evaluate the word `'eq(+('<x>', s(s(s(0)))), s(0))'`. The above statement is proved directly by the following evaluation output obtained:

```
(       false           if  x = 0,
        false           if      x=s(0),
        false           if      x=s(s(x2))      ),
```

where `x2` is a variable of the `<nat>` type and it is generated by the W language interpreter.

Example 7. Let us prove that the unique values of parameter `a` for which equation `x+a=1` has natural roots are `0` and `1`.

Let us use Sample2 again in order to evaluate the word `'eq(+('<x>','<a>'), s(0))'`. The declaration `<a>:<nat>` must be included in the variable declaration part of Sample2. The following list is obtained:

```
(       false   if   x = 0, a = 0,
        true    if   x = 0, a = s(0),
        false   if   x = 0, a = s(s(a2)),
        true    if   x = s(0), a = 0,
        false   if   x = s(x1), a = s(a3),
        false   if   x = s(s(x2))               )
```

Thus, the statement is proved and the roots searched are found.

Example 8. Let us prove that (+,+) and (*,*) are the sequences of operations + or * such that if we substitute them for the symbol ? in the series (1?2)?3 we are going to obtain an arithmetic expression with a value equal to 6.

We can prove this statement by evaluating the word

```
'eq('<g>'('<f>'(s(0),s(s(0))),s(s(s(0)))), s(s(s(s(s(0))))))  )'
```

(where <f> and <g> are variables of <func_name> type) through Sample2.

The evaluation results in:

```
(       true    if      f = +, g = +,
        false   if      f = +, g = *,
        false   if      f = *, g = +,
        true    if      f = *, g = *        ).
```

This example shows how we can use the *W* language to make a logical evaluation of words having higher-order functions as their variables.

Finally let us point out that higher-order functions are one of the most appealing features of functional programming languages. The uses demonstrated above have shown that one of the advantages of the *W* language is that it has this feature.

REFERENCES

[1] M. Bellia and G. Levi, The relation between logic and functional languages: a survey, *The Journal of Logic Programming* **3** (1986), 217-236.

[2] St. Holldobler, Foundations of equational programming, LNAI 353, Aug. 1989.

[3] A. Radensky and M. Todorova, An approach to programming by means of equations: transformation programs and an interpreter for such programs. Конференция КНВВТ по автоматизации информационных процессов на персональных ЭВМ, Будапест, 1986, 167-181.

[4] A. Radensky and M. Todorova, Functional programming, logic programming, equational programming. *W*: A language which implements two generalizations to equational programming, First Franco-Bulgarian Workshop on Logic Programming and Automated Inferencing. Nov., 1988, 21-25.

[5] M. Todorova, *W*: functional programming language based on unification, *Informatika - God.* **21** (1987), 143-151.

[6] M. TODOROVA, Determination classes of reduction sequences in generalized subtree replacement systems. *Annuaire de l'Universite de Sofia "Kl. Ohridski"* **79** (1985), 127-133.

[7] M. TODOROVA, Generalized subtree replacement systems, Ph.D.Thesis, University of Sofia, Faculty of Mathematics and Informatics, 1987.

Sofia University "St. Kl. Ohridski"
Faculty of Mathematics and Informatics
5, James Bourchier str.
1126 Sofia
BULGARIA

Received 24.04.1992
Revised 18.11.1992