STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLV**, Number 2, 2000

TERM REWRITING SYSTEMS IN LOGIC PROGRAMMING AND IN FUNCTIONAL PROGRAMMING

DOINA TĂTAR, GABRIELA ŞERBAN

Abstract. Automated theorem proving and term rewriting system are fields with big interest since some years. Often these fields have a common development. Is it not amazingly that logic programming and functional programming, which belongs to both these fields, offers simple solutions to problems arising at the frontier of them. In [8], the author submitted a challenge for "finding an optimum way to implement the rewriting systems ". This paper presents the way in that the logic programming and functional programming offer their concision to realize a sound implementation of the TRS.

1. Introduction

In the first section we will presents shortly the equation systems, the TRS, the "critical pair" idea and the completion algorithm [1, 5, 7, 10]. In the following sections we will outline some problems and their solution in our implementation in Prolog (section 2) and in Lisp (section 3).

Definition 1 An equational theory (F,V,E) consists of:

- a set F of function symbols (with the same sort, for simplicity).
- a set V of variables.

Let T(F,V) be the set of terms build from F and V.

- a set of pairs of equations, s=t,s,t $\in$ T(F,V).

The set of equations E defines a syntactical equality relation $==_E$ on T(F,V), usually defined as "replacing equals by equals".

The fundamental problem in an equational theory is the "validity" or "word problem", which is undecidable:

"Give s and $t \in$ T(F,V), does $s ==_E$ t ?"

The undecidability (more precisely, the semidecidability) of the "word problem" is transferred on the approach by the TRS, but this approach is, on the our opinion, more algorithmically.

2000 *Mathematics Subject Classification.* 68T15.

1998 *CR Categories and Descriptors.* D.1.6. [**Software**] : Programming Techniques – *Logic Programming*;I.2.3. [**Computing Methodologies**] : Artificial Intelligence – *Deduction and Theorem Proving.*

Definition 2. A TRS is a set of rules: $R = \{l \to r \mid l, r \in \mathrm{T(F,V)}$, every variables occurring in term r also occurs in term l $\}$.

A TRS defines a rewrite relation $\to_R$:

Definition 3. $s \to_R t$ iff there is a rule $l \to r \in R$ and an occurrence p in s such that the subterm of occurrence p, noted $s \mid_p$ and the term t have the property:

$$s \mid_p = \sigma(l), t = s[p \leftarrow \sigma(r)]$$

for some substitution σ. Here notation $s[p \leftarrow \sigma(r)]$ represents the term obtained from s by replacing the subterm of occurrence p by the term $\sigma(r)$.

We denote by $\to_R^*$ and $\leftarrow\!\!\to_R^*$ the reflexive-transitive and reflexive-transitive-symmetric closure of $\to_R$.

In order to solve the "word problem" for an equational theory E, compute an TRS R_E such that $s =\!=_E t$ is a relation equivalent with $s \leftarrow\!\!\to_R^* t$. Let us denote R_E as *associated with E*.

The TRS R_E is the canonical (terminating and confluent) TRS *associated with E*, obtained as output of the completion procedure Knuth -Bendix. This algorithm has as input the set E and a reduction order over T(F,V).

Definition 4 The normal form of a term t , denoted $t \downarrow_R$, is a term with the followings properties:

$$1. t \to_R^* t \downarrow_R$$
$$2. t \downarrow_R \textit{ irreducible.}$$

Observations:

1. If a TRS R has the property that every term has a unique normal form, then:

$s \leftarrow\!\!\to_R^*$ iff $s \downarrow_R = t \downarrow_R$, because $s \leftarrow\!\!\to t_R^*$ is $s \to_R^* s \downarrow_R$ and $t \to_R^* \downarrow_R$. Thus, testing $s \leftarrow\!\!\to t_R^*$ is is the same as testing that $s \downarrow_R = t \downarrow_R$.

2. In a canonical TRS ,R, every term has a unique normal form.

We won't describe the well known Knuth-Bendix algorithm. Instead, we will survey the critical pair idea, staying on the ground of this algorithm.

Definition 5 Let $l_1 \to r_1$ and $l_2 \to r_2$ be two rules in R. By renaming the variables we may assume that they do not share common variables. If $\sigma_1(l_1) = \sigma_2(l_2)$, then the pair of terms $(\sigma_1(l_1), \sigma_2(l_2))$ is a critical pair for R.

The Knuth-Bendix algorithm computes, for every critical pair (t_1, t_2) of R', the normal forms $t_1 \downarrow_{R'}$ and $t_2 \downarrow_{R'}$. If this normal forms are different, then a rule $t_1 \downarrow_{R'} \to'_R t_2 \downarrow_{R'}$ or converse, (depending of the case $t_1 \downarrow_{R'} > t_2 \downarrow_{R'}$ or the converse), is added to R'. Let observe that the procedure fails if neither $t_1 \downarrow_{R'} > t_2 \downarrow_{R'}$ nor the converse is true.

2. Implementation in Prolog

A set of problems for implementation in Turbo Prolog derives from the fact that in this language does not exist the standard predicates **functor**, ==, and **op**. This fact lead as construct two specific domains in section **domains** of our programs as follows:

```
domains
  term=var(symbol);con(symbol);cmp(symbol,terml)
  terml=term*
  termll=terml*
```

For example, if we must introduce the term **f(x,y,a)** , we will write:

cmp(f,[var(x),var(y),con(a)]), respecting the conventions for syntax of formulas in first-order logic. Also,if we must introduce the formula **p(x,f(y,z))** we will write:

atom(p,[var(x),cmp(f,[var(y),var(z)])]) .

A TRS R of I rules, as in definition 2, is done by a couple of predicates **l(t,N)** and **r(t,N)** where **t** is a term and N=1, $\cdots$,I is the index of the rule. We worked in this program with the three starting rules associated with the theory E of groups.

l(cmp("f",[con(e),var(a)]),1).
l(cmp("f",[cmp("g",[var(a)]),var(a)]),2).
l(cmp("f",[cmp("f",[var(a),var(b)]),var(c)]),3).
r(var(a),1).
r(con(e),2).
r(cmp("f",[var(a),cmp("f",[var(b),var(c)])]),3).

The predicates which realizes the rewriting relation $X \to Y$ with a rule N in definition 3 is the predicate **rewrite (X,Y,N)** .

```
rewrite(X,Y,N):-l(X,N),r(Y,N),!.                              (1)
rewrite(X,Y,N):-member_left(X,L1,L2,N),                       (2)
                list_var(X,L_var),                            (3)
                lg_list(Lnou,K),                              (4)
                l(M_stg,N),                                   (5)
                aplic_subst(M_stg,Nou_m_stg,L1,L2),           (6)
                tr_term_str(Nou_m_stg,St_stg),                (7)
                aplic_subst(X,NouX,L1,L2),                    (8)
                list_var(NouX,L_var_n),                       (9)
                lg_list(Lnoun,K),                            (10)
                tr_term_str(NouX,St),                        (11)
                r(M_dr,N),                                   (12)
                aplic_subst(M_dr,Nou_m_dr,L1,L2),            (13)
                tr_term_str(Nou_m_dr,St_dr),                 (14)
                strsr_first(St,St_stg,St_dr,Nou_string),     (15)
                tr_str_term(Nou_string,Yinterm),             (16)
                sc_lista(L2,L1,L2nou,L1nou),                 (17)
                aplic_subst(Yinterm,Y,L2nou,L1nou),!.        (18)
```

The predicate **member-left** (denoted by (1)) is defined as follows:

```
/*  member_left(X,L1,L2,N):-the rule  N-th has the property that
    his left side unifies with a subterm of term X, and the unifier
    has the domain L1 and the codomain L2.    */
```

One of the clauses for **member-left** must be:

```
member_left(X,L1,L2,N):-subterm(S,X),
                        l(Z,N),
                        unify(S,Z,L1,L2).
```

The predicate **aplic-subst(t,s,L1,L2)** denoted by (6) applies the substitution σ =(L1/L2) to t obtaining s. The predicates **tr-term-str** transforms a term (e.g. f(a,x)) in a string (f2ax). The reason for this transformation is to provide to predicate:

strsr-first (St,St-stg,St-dr,Nou-string), denoted by (15),
his first three arguments (the lines (7),(11),14)). Thus, one step of the realization of the relation $\rightarrow$ is accomplished by the predicate **strsr-first**. This is defined as:

```
/* strsr-first(S1,S2,S3,S):- the string S is obtained by
   replacing in the string S1 the first occurrence of the
   substring  S2 by the string S3. */
```

The converse transformation of a string into a term is realized by the predicate **tr-str-term** (16). A clause for this one must be:

```
tr_str_term(X,Y):-str_len(X,L),L>0,frontstr(1,X,Z,U),
                  frontstr(1,U,N,W),
                  str_int(N,N1),
                  frontstr(N1,W,WW,WWW),
                  tr_str_terml(WW,V),
                  tr_str_terml(WWW,V1),
                  append(V,V1,V2),
                  Y=cmp(Z,V2),lg_list(V2,N1).
```

The relation $\rightarrow_R^*$ defined as the reflexive -transitive closure of $\rightarrow_R$ is realized by the predicate **rewrite***. The clauses for this predicate are:

```
rewrite*(X,Y):-rewrite(X,Y,N).

rescrie*(X,Y):-rewrite(X,Z,N),!,rewrite*(Z,Y).
```

The predicates **critical-pair** and **normal-form** are defined as:

```
critical-pair(X,Y):-l(X,N),member_left(X,L1,L2,M),l(Z,M),
                    aplic-subst(Z,Y,L1,L2).
normal-form(X,Y):-rewrite*(X,Y),not(rewrite(Y,_,_)).
```

At the end of the application of the Knuth-Bendix algorithm, the canonical TRS is given as usually by 10 rules. (Some intermediary rules are deleted because they have been rewritten in the same terms.) The obtained canonical TRS can be used for demonstrate some theorem in group theory. For example, if we want to prove that $t_1 = i((i(a)+a)+(b+i(b)))$ is equal with $t_2 = b+(i(a+b)+a)$, then we run the program with **normal-form**(t_1,X) and **normal-form**(t_2,Y). We will obtain X=Y.

3. Implementation in Lisp

In this section our aim is to present how the rewriting relations could be defined in LISP.

3.1. **LISP representations.** First, we have to establish the way in which the terms are represented in LISP.

- a variable **x** is represented as a list **(var x)**;
- a constant **a** is represented as a list **(con a)**;
- a functional symbol **f** is represented as a list **(cmp f)**;
- a function **f(LA)** where **f** is a functional symbol and **LA** is a list of arguments, is represented as a list **((the list corresponding to f) (the list of arguments))**; for example, **f(a,x)** is represented as a list **((cmp f) ((con a) (var x)))**.

With the above considerations, if we must introduce the term **g(x,f(y,z))** we will write ((cmp g) ((cmp f) ((var y) (var z)))).

A rule **l→r** from a TRS is represented as a list **(list-l list-r)**, where **list-l** and **list-r** are the representations in LISP of the terms **l** and **r**. For example, a rule $\mathbf{f}(\mathbf{a},\mathbf{x})\rightarrow\mathbf{x}$ is represented as the list **(((cmp f) ((con a) (var x))) (var x))**.

A TRS R of N rules is represented as a list of rules **(rule-1 rule-2 ... rule-N)**, each rule is represented as we described above.

In the followings, we work with the three starting rules associated with the theory of groups. The list of rules is denoted by **LR** and is the following:

```
(setq LR '(
    ( ((cmp f) ((con e) (var a)))
      (var a)
    )
    (
      ((cmp f) (((cmp g) ((var a))) (var a)))
      (con e)
    )
    (
      ((cmp f) (((cmp f) ((var a) (var b))) (var c) ))
      ((cmp f) ((cmp f) ((var a) ((cmp f) ((var b) (var c))))))
    )
  )
)
```

3.2. **Functions defined for rewriting rules.** The functions which realize the rewriting relation **X→Y** with a rule N in definition 3 is the function **(rewrite X N LR)** which returns Y.

```
(defun rewr (X N LR)
; LR represent the list of rules
  (prog (RN)
```

```
    (setq RN (rule-N N LR))
    (cond
       ((equal (car RN) X) (return (cadr RN)))
       (t (setq Y (cadr RN))
          (setq UNIF (member-left X (car RN)))
          (cond
            ((null UNIF) nil)
            (t
                (setq L1 (car UNIF))
                (setq L2 (cadr UNIF))
                (return (apply-subst L1 L2 Y))
            )
          )
       )
    )
  )
)
```

The function **(rule-N N LR)** returns the N-th rule from the list of rules LR.

```
(defun rule-N (N LR)
  (cond
      ((null LR) nil)
      ((= N 1) (car LR))
      (t (rule-N (- N 1) (cdr LR)))
  )
)
```

The function **(member-left X Y)** is defined as follows:

- if Y (the left side of a given rule) unifies with a sub-term of X, and the unifier has the domain $L1$ and the codomain $L2$, then the function returns the list **(L1 L2)** (this list is calculated by the function **(unify X Y)**);
- else the function returns **NIL**.

```
(defun member-left (X Y)
  (cond
      ((not (equal (length X) (length Y))) nil)
      (t (unify X Y))
  )
)
```

The function **(apply-subst L1 L2 Y)** applies the substitution $\sigma = (L1/L2)$ to Y and returns the result.

```
(defun apply-subst (L1 L2 Y)
   (subst Y L1 L2)
)
```

The function **(rewrite X)** is defined as follows:

- returns a list of elements having the form **(N Y)**, where Y is the right side of the rewriting relation **X**→**Y** with the rule N (if it is possible) - this list is calculated by the recursive function **(rewrite-rule X N LR)** which returns the result of rewriting X with the N-th rule of LR;
- returns NIL, if no rewriting relations for X are possible.

```
(defun rewrite-rule(X N LR)
  (cond
    ((> N (length LR)) nil)
    (t
       (setq RN (rewr X N LR))
       (cond
         ((not (null RN)) (cons (list N RN)
                                (rewrite-rule X (+ N 1) LR)
                          )
         )
         (t (rewrite-rule X (+ N 1) LR))
       )
    )
  )
)

(defun rewrite (X)
  (rewrite-rule X 1 LR)
)
```

The relation defined as the reflexive-transitive closure of the rewriting relation R is defined as the function **(rewrite* X)**.

```
(defun rewrite* (X)
  (setq Y (rewrite X))
  (append Y (rewr* Y))
)

(defun rewr* (Y)
 (cond
   ((null Y) nil)
   (t (append (rewrite (cadar Y)) (rewr* (cdr Y))))
 )
)
```

The **normal-form** is defined as a function **(normal-form X)**.

```
(defun normal-form (X)
  (n-form (rewr* X))
)
```

```
(defun n-form (Y)
  (cond
    ((null Y) nil)
    ((null (rewrite (cadar Y))) (append (car Y) (n-form (cdr Y))))
    (t (n-form (cdr Y)))
  )
)
```

Examples

(1) if X is ((cmp f) (((cmp g) ((var b))) (var b))), then the result of rewriting X este ((2 (CON e)));

(2) if X is ((cmp f) ((con e) (var b))), then the result of rewriting X este ((1 (VAR b)));

(3) if X is ((cmp f) ((con e) (var a))), then the result of rewriting X este ((1 (VAR a))).

(4) if X is ((cmp f) (((cmp f) ((var a) (var b))) ((cmp g) ((var c))))), then the result of rewriting X este (3 ((cmp f) ((cmp f) ((var a) ((cmp f) ((var b) ((cmp g) ((var c)))))))).

References

[1] Avenhaus J., Madlener K. : "Term rewriting and Equational Reasoning" in Formal Techniques in A.I., A coursebook, R.B.Banerdji (ed) 1990.

[2] K.H. Blasius, H.J. Burkert: "Deduction systems in Artificial Intelligence", Ellis Horwood Ltd.,1989.

[3] Buchberger B.: "History and basic features of the Critical-Pair Completion Procedure", J. of symbolic Computation 3, 1987, pp. 3–38.

[4] W.F. Clocksin, C.S. Mellish : Programming in Prolog, Springer-verlag, 1984.

[5] Huet G., Oppen D.D.: "Equations and rewrite rules: A survey", in "Formal languages: theory, perspectives and open problems", ed. R. Book, 1980.

[6] Jouannaud J.P., Lescanne P.: "Rewriting Systems", in Technology and Science of Informatics, 1987, pp. 181–199.

[7] Knuth D.E., Bendix P.P.: "Simple word problem in Universal Albgebra", Comp. prob. in Abstr. Alg. (ed. J. Leech), 1970.

[8] Lescanne P.: "Current trends in rewriting techniques and related Problems", IBM int. symp. on Trends in Computer Algebra, Germany, 1987.

[9] Rusinowitch M.: "Demonstration automatique. Techniques de reecriture" Inter. Edition, Paris, 1989.

[10] Tatar D.: "A new method for the proof of theorems", Studia Universit. Babes-Bolyai, Mathematica, 1991, pp. 83–95.

[11] Tatar D.: "Term rewriting systems and completion theorems proving: a short survey", Studia Univ. Babes-Bolyai, Mathematica, 1992, pp. 117–125.

Department of Computer Science, Faculty of Mathematics and Computer Science, "Babeş-Bolyai University, 1, M. Kogălniceanu St., RO-3400 Cluj-Napoca, Romania