Appl. Math. and Comp. Sci., 1993, vol.3, No.2, 383–392

AUTOMATED REALIZATION OF BOOLEAN ALGEBRA THEOREMS IN THE SIMPLIFICATION OF LOGICAL EXPRESSIONS

JANUSZ SZAJNA*

This paper presents an approach to the automation of symbolic transformation and simplification of complex logical expressions. The transformation is performed with the aid of Prolog – one of the two programming languages being used for artificial intelligence. Due to the declarative nature of Prolog the specification of a particular theorem together with its variations becomes its realization at the same time. Moreover, each predicate is a direct reflection of a heuristic way of the expressions transformations, written in Prolog convention.

1. Introduction

The development of artificial intelligence opens new, very promising possibilities of computer–aided realization of symbolic operations as well as operations based on heuristic methods. Among others, intensive research is being performed on construction of translators, pattern analysis, analysis and translation of natural language etc. (Warren, 1980; Clocksin, Mellish, 1987; Schalkoff, 1990).

A lot of logic circuits design problems can be solved as well, using heuristics methods and symbolic transformations. A multi logic synthesis can be taken as an example. On of the purposes of recent research in this field is providing a well–defined, abstract mathematical model, independent of the implementation technology, from which multi logic synthesis could be done based on algebraic formal transformations using the Boolean algebra theorems. This type of approach, based on symbolic transfomations, recursion and heuristics (factoring, symbolic minimization, restructuring, decompositin) has been proposed for example by Brayton, (1989) and Hachtel, (1992).

This paper presents an approach to using Prolog (one of two artificial intelligence programming languages) for solving these kind of tasks. Transformation and simplification of complex logical expressions is given as an example.

The expressions being analysed can be arbitrarily complicated and can contain multiple nested bracketed subexpressions. Several examples are given below:

* Department of Computer Engineering and Electronics, Higher College of Engineering, Zielona Góra, Poland

$$y_1 = a^*(b + /c) + d$$

$$y_2 = a^*(b + /(x + y)) + d$$

$$y_3 = a^*((p + /r * s) + /(x + y)) + d$$

The transformation of expressions is performed in a heuristic way and is based on the following theorems of Boolean Algebra:

1. $A + 1 = 1$
2. $A + /A = 1$
3. $A + A = A$
4. $A + A^*B = A$
5. $/(A + B) = /A^*/B$
6. $/(A^*B) = /A + /B$
7. $A^*B = B^*A$
8. $A^*1 = A$
9. $A^*A = A$
10. $A^*/A = 0$
11. $A^*(B + C) = A^*B + A^*C$

2. Representation of Compound Logical Expressions by Recursive Prolog Terms

Arbitrary compound (nested) logical expressions can be easily represented recursively. For this purpose the following set of Prolog terms can be defined:

```
expression = sumTerm*
 sumTerm = element*
 element = variable(symbol);
           not_variable(symbol);
           exprWithinBrackets(expression);
           not_exprWithinBrackets(expression)
```

In the above definitions it is assumed that a logical expression is written as a sum of terms (sumTerm*), and every sum term as a product of elements (element*). An element is a variable or its complement or any Boolean expression within brackets (recurrence) – complemented or not.

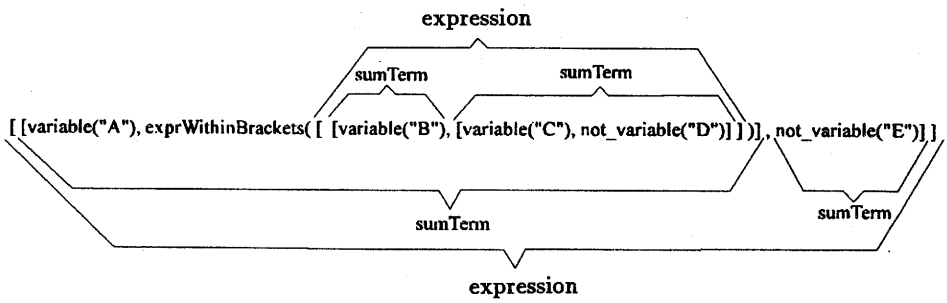

Fig. 1. Prolog term representing the expression.

Figure 1 presents an example of a Prolog term constructed in this way, and representing the expression.

$$W = A^*B + C^*/D) + E$$

where $A..E$ are variables.

3. Realization of General Simplification Strategy

The application of Prolog allows the automation of transformation in a very simple and natural way despite arbitrary complexity of the expressions. The main predicate of transformation and simplification of expressions can be written as follows:

```
PREDICATES
SimplificationOfExpression (expression, expression)     %(in,out)

CLAUSES
SimplificationOfExpression (InputExpression, [[variable("1")]])
    :- Th1and2_TestOfOne (InputExpression, !.
SimplificationOfExpression (InputExpression, OutputExpression)
    :- Th3and4_ReducedTerms (InputExpression, Expr1),
       TransfOfTermsAndBrackets (Expr1, OutputExpression).
```

In a way similar to typical manual realization, the analysis starts from the search for terms equal to 1 or complementary elements (like A and $/A$) within one product term. This task is realized by the first clause of the predicate *SimplificationOfExpression* (a call to the predicate *Th1and2_ TestOfOne*). If it returns true, the analysis is finished, as the whole expression equals 1, represented here as *[[variable("1")]]*. In the other case (second clause) the theorems 3 and 4 are checked in order to eliminate repeated and reduced product terms. Next, due to the fact, that the transformation of the expression may not proceed without earlier transformation of its terms, the analysis of individual terms has to be done. In general, this part forms a crucial step of the simplification of the expression as it is connected with the analysis of nested subexpressions seized by brackets. Respective Prolog declaration can have the following form:

```
PREDICATES
TransfOfTerms (expression, expression)          (in,out)
CLAUSES
TransfOfTerms ([], []) :- !.
TransfOfTerms ([Term | Tail], OutputExpression)
    :- Th10_AandNotA (Term), !,
       TransfOfTerms (Tail, OutputExpression).
TransfOfTerms ([Term | Tail], OutputExpression)
    :- TransfOfBrackets_and_Products (Term, Term1),
       TransfOfTerms (Tail, Term1),
       append ([Term1], Expr1, Expr2),
       PassingOverSuperfluonsBrackets (Expr2, Expr3),
       Theorems1_2_3_4 (Expr3, OutputExpression).
```

If the analyzed product term fulfills the conditions of theorem 10, then it is eliminated. In the other case it has to be transformed in a proper way (predicate *TransfOf-Brackets_ and_ Products*), and then appended to the rest of the expression. The mentioned part of the expression is transformed earlier by recursively

calling predicate *TransfOrTerms*. When these operations are finished, superfluous brackets are eliminated and theorems 1–4 are applied.

The transformation of arbitrarily nested products of terms seized by brackets can be done by the following predicate:

```
PREDICATES
TransfOfBrackets_and_Products (sumTerm, sumTerm)        % (in,out)
Product_and_Th7to11 (element, element,  sumTerm)        % (in,in,out)
SimplificationOfExpression (expression, expression)     % (in,out)

CLAUSES
TransfOfBrackets_and_Products ([ exprWithinBrackets (InExpression ) ],
                               [ exprWithinBrackets (OutExpression) ])
    :- !, SimplificationOfExpression (InExpression, OutExpression).
------
TransfOfBrackets_and_Products ([ not_exprWithinBrackets (InExpression) ],
                               [ exprWithinBrackets (OutExpression) ])
    :- !, Th5and6_deMorgan (InExpression, Expr1),
       SimplificationOfExpression (Expr1, OutExpression).
------
TransfOfBrackets_and_Products ([Variable], [Variable]) :- !.
------
TransfOfBrackets_and_Products ([Head | Tail], OutTerm)
    :- TransfOfBrackets_and_Products ([Head],  [X1]),
       TransfOfBrackets_and_Products ( Tail, [X2]),
       Product_and_Th7to11 (X1, X2, OutTerm).
```

The first clause of the predicate defines the sequence of actions when the input term is seized by brackets. In such a case the bracketed expression should be simplified by the predicate *SimplificationOfExpression* discussed earlier. The second clause relates to the complemented expression within brackets product. In this case the simplification is preceded by application of de Morgan theorems. The third clause is used when the input term is a single literal. Such a term remain unchanged. The last clause describes the method of analysis, when the input is a simple product (i.e. the list can be divided into head and tail). In this case subsequent literals are sequentially taken and transformed (through recursive execution of the predicates for head and tail), and then the product is formed again.

The formulation of the product is connected with the application of theorems 7–11, whose Prolog realization is described in section 4.

4. Prolog–Based Realization of Boolean Algebra Theorems

Due to the declarative character of Prolog a theorem's specification itself (including cases) is a realization of the theorem. It will be illustrated by the example of theorems 1 and 2, which define conditions necessary for an expression to be equal to 1.

Theorems 1 and 2. *Theorems 1 and 2 have the following form:*

$$A + 1 = 1 \tag{1}$$

$$A + /A = 1 \tag{2}$$

This means, that the result is equal to 1, denoted as

$$[[variable("1")]]$$

if theorem 1 is applied to a list:

$$[[...], [variable("1")], [...]]$$

or theorem 2 to a list:

$$[[...], [variable("A")], [...], [not\_variable("A")]$$

Obviously, theorems 1 and 2 can be presented by the following verbal description: analyzed expression tends to 1, when at least one of its components equals 1 or when it contains complementary components (A and $/A$).

Keeping in mind that the symbol A which appears in theorem 2 can denote: a variable; a complement of a variable; a subexpression within brackets; a complement of a subexpression, the above verbal description can be represented in Prolog as follows:

```
CLAUSES
SimplificationOfExpression (InExpression, [[variable("1")]])
    :- Th1and2_TestOfOne (InExpression), !.

Th1and2_TestOfOne([[variable("1")  ]          |_  ]) :-  !.
Th1and2_TestOfOne([[variable(A)    ]          |Tail]) :- member
                                                        ([not_variable(A)],
                                                         Tail), !.
Th1and2_TestOfOne([[not_variable(A)]          |Tail]) :- member
                                                        ([variable(A)],
                                                         Tail), !.
Th1and2_TestOfOne([[exprWithinBrackets(X)]    |Tail]) :- member
                                                        ([not_exprWithinBrackets(X)],
                                                         Tail), !.
Th1and2_TestOfOne([[not_exprWithinBrackets(X)]|Tail]) :- member
                                                        ([exprWithinBrackets(()],
                                                         Tail), !.
Th1and2_TestOfOne([ _                          |Tail]) :- Th1and2_TestOfOne (Tail).
```

Therefore, this specification of theorems 1 and 2 is their realization at the same time.

For example, the first clause of the predicate denotes that the expression tends to one if the very first component (the head of the list) is equal to one. The second clause reflects the fact that the expression will evaluate to one if the first component is A and there exists A's complement among its other components, etc. The meaning of the last clause is that when none of the preceding clauses succeeds, then during the analysis the first component should be omitted and the rest of the expression should be re–analyzed (recursive call to the same predicate for the tail of the list).

Theorems 3 and 4. *The theorems:*

$$A + A = A \tag{3}$$

$$A + A^*B = A \tag{4}$$

The Prolog realization is done as search of the expressions components and proper reductions whenever identical or absorbed components are found. Both theorems are realized by the same predicate, as theorem 3 is a special case of theorem 4.

A reduction of absorbed components according to the formula

$$A + A^*B = A$$

does not require any analysis of B. For example, if B is a subexpression delimited by brackets:

$$B = exprWithinBrackets([[...],[...],...])$$

then the reduction takes place without any analysis of the subexpression B.

The Prolog notation of theorems 3 and 4 can be written as follows:

```
PREDICATES
Th3and4_ReducedTerms (expression, expression)
RemovingSuperfluonsTerm (sumTerm, expression, expression)
ContainingOfTerms (sumTerm, sumTerm)
CLAUSES
Th3and4_ReducedTerms ([], []).

Th3and4_ReducedTerms ([Term | RestOfSum], [Term | Tail])
    :- RemovingSuperfluonsTerm (Term, RestOfSum, RestAfterRemoving) ,!,
       Th3and4_ReducedTerms (RestAfterRemoving, Tail).

Th3and4_ReducedTerms ([_ | Tail], L)
    :- Th3and4_ReducedTerms (Tail, L.

% --------

RemovingSuperfluonsTerm (_, [], []).

RemovingSuperfluonsTerm (Term0, [Term1 | RestOfSum], R)
    :- ContainingOfTerms (Term0, Term1), !,
       RemovingSuperfluonsTerm (Term0, RestOfSum, R)

RemovingSuperfluonsTerm (Term0, [Term1 | RestOfSum], [Term1 | Tail])
    :- not (ContainingOfTerms (Term0, Term1) ),
       RemovingSuperfluonsTerm (Term0, RestOfSum, Tail)

% --------

ContainingOfTerms ([], _).
ContainingOfTerms ([Head | Tail], Term1)
    :- member (Head, Term1)
       ContainingOfTerms (Tail, Term1)
```

Theorems 5 and 6 (De Morgan's). *The theorems:*

$$/(A + B) = /A^*/B \tag{5}$$

$$/(A^*B) = /A + /B \tag{6}$$

Similarly to other theorems, the realization of these theorems in Prolog is very simple. The crucial point is to keep in mind that the symbols A and B can

represent complex objects. Moreover, according to our declaration of terms, there exists a difference in the way how the product terms and literals are transferred to respective lists.

The Prolog notation for De Morgan's theorems can have the following form (in the realization of theorem 5 the last clause of the predicate Complement is used):

```
PREDICATES
Th5and6_deMorgan (expression, expression)
Complement (sumTerm, element)
ComplementProduct (sumTerm,  expression)
ComplementElement (element, element)
CLAUSES
Th5and6_deMorgan ([], [[]]).
Th5and6_deMorgan ([Head | Tail], [[Head1 | Tail1]])
    :- Complement (Head, Head1),
       Th5and6_deMorgan (Tail, Tail1).

% -------
Complement ( [variable(A)     ]           , not_variable(A)          ) :- !.
Complement ( [not_variable(A)]            , variable(A)              ) :- !.
Complement ( [exprWithinBrackets(W)     ] , not_exprWithinBrackets(W)) :- !.
Complement ( [not_exprWithinBrackets(W)] , exprWithinBrackets(W)     ) :- !.
Complement ( X                            , exprWithinBrackets(X1)   ) :-
ComplementProduct`([,`[`).

% -------
ComplementProduct ([], []).
ComplementProduct ([Head | Tail], [[Head1] | Tail1])
    :- ComplementElement (Head, Head1),
       ComplementProduct (Tail, Tail1).

% -------
ComplementElement (variable(A)               , not_variable(A)          ).
ComplementElement (not_variable(A)           , variable(A)              ).
ComplementElement (exprWithinBrackets(X)     , not_exprWithinBrackets(X)).
ComplementElement (not_exprWithinBrackets(X) , exprWithinBrackets(X)    ).
```

Theorems 7, 8 and 9. *Theorems 7, 8 and 9, as well as 10 and 11, which all deal with product of components, are realized by the following predicates:*

```
PREDICATES
Product_and_Th7to11 (element, element, sumTerm)      % (in,in,out)
Product_and_Th8to11 (element, element, sumTerm)      % (in,in,out)
Th10_AandNotA (sumTerm)                              % (in)
```

Theorems 7, 8 and 9 are formulated as follows:

$$A^*B = B^*A \tag{7}$$

$$A^*1 = A \tag{8}$$

$$A^*A = A \tag{9}$$

Their realization is straightforward and does not require any comments.

Realization of theorem 7 $A^*B = B^*A$:

```
Product_and_Th7to11 (A, B, C)
    :- Product_and_Th8to11 (A, B, C), !
       or
       Product_and_Th8to11 (B, A, C).
```

Realization of theorem 8 $A^*1 = A$:

```
Product_and_Th8to11 (A, variable("1"),  [A]) :- !.
```

Realization of theorem 9 $A^*A = A$:

```
Product_and_Th8to11 (A, A, [A]) :- !.
```

Theorem 10. *Theorem 10 relates to a product of complements and is formulated as follows:*

$$A^*/A = 0 \tag{10}$$

It is realized in a way very similar to theorem 2 relating to a sum of complements (simple Prolog notation of heuristic realization):

```
Product_and_Th8to11 (A, B, [])
    :- Th10_AandNotA (A, B), !.

Th10_AandNotA ( [variable(A),                | Tail])  :-  member
                                                           (not_variable(A),
                                                            Tail), !.
Th10_AandNotA ( [not_variable(A),            | Tail])  :-  member
                                                           (variable(A),
                                                            Tail), !.
Th10_AandNotA ( [exprWithinBrackets(X)       | Tail])  :-  member
                                                           (not_exprWithinBrackets(),
                                                            Tail), !.
Th10_AandNotA ( [not_exprWithinBrackets(X)   | Tail])  :-  member
                                                           (exprWithinBrackets(X),
                                                            Tail), !.
Th10_AandNotA ( [_                           | Tail])  :-  Th10_AandNotA (Tail).
```

Theorem 11. *Theorem 11 is formulated as follows:*

$$A^*(B + C) = A^*B + A^*C \tag{11}$$

It's realization comprises three cases depending on the form of A.

Theorem 11 – case $A^*(B+C) = A^*B + A^*C$:

```
Product_and_Th8to11 (variable(A),
                     exprWithinBrackets(InExpression),
                     [exprWithinBrackets(OutExpression)])
   :- ProductOfExpressions ( [[variable(A)]], InExpression, OutExpression).
```

Theorem 11 – case $/A^*(B+C) = /A^*B + /A^*C$:

```
Product_and_Th8to11 (not_variable(A),
                     exprWithinBrackets(InExpression),
                     [exprWithinBrackets(OutExpression)])
   :- ProductOfExpressions ( [[not_variable(A)]], InExpression, OutExpression).
```

Theorem 11 – case $(A+B)^*(C+D) = A^*C + A^*D + B^*C + B^*D$:

```
Product_and_Th8to11 (exprWithinBrackets(InExpression_1),
                     exprWithinBrackets(InExpression_2),
                     [exprWithinBrackets(OutExpression)])
   :- ProductOfExpressions (InExpression_1, InExpression_2, OutExpression).
```

The Prolog notation of theorem 11 contains predicates *productOfExpressions* and indirectly, *productOfTermAndExpression*. Both predicates realize symbolic logical multiplication.

Product of Expressions:

```
ProductOfExpressions ([], _, []).
ProductOfExpressions ([Head | Tail], Expression, OutExpression)
   :- ProductOfTermAndExpression (Head, Expression, Expr1),
      ProductOfExpressions (Tail, Expression, Expr2),
      append (Expr1, Expr2, Expr3),
      Theorems1_2_3_4 (Expr3, OutExpression).
```

Product of Term and Expression:

```
ProductOfTermAndExpression (_, [], []).
ProductOfTermAndExpression (Term, [Head | Tail], OutExpression)
   :- append (Term, Head, X),
      Th10_AandNotA (X), !,
      ProductOfTermAndExpression (Term, Tail, OutExpression).
ProductOfTermAndExpression (Term, [Head | Tail], [X1 | Tail1])
   :- append (Term, Head, X),
      PassingOverBracketsOfSumTerm (X, X1),
      ProductOfTermAndExpression (Term, Tail, Tail1).
```

Realization of symbolic logical multiplication in Prolog language is very simple and is performed by proper application of a predicate append. In the above predicates the multiplication operations are additionally tied with application of theorems 1, 2, 3, 4 and 10 as well as with reduction of superfluous brackets.

5. Conclusions

In the paper we have tried to show the simplicity of automation of logical expressions symbolic transformation with the aid of declarative programming (Prolog). The problem specification itself is practically equivalent to its implementation and every predicate is a set of clauses, each of which represents one rule of heuristic transformation of an expression. The presented method does not depend on the complexity of the expression.

References

Bratko I. (1986): *Prolog Programming for Artificial Intelligence.* — Addison Wesley, Reading (MA).

Brayton R.K., Rudell R., Sangiovani–Vincentelli A.L. and Wang A.R.R. (1989): *Multi–Level Logic Synthesis.* — Oxford/Berkeley summer Engineering Programe, University of California, Berkeley.

Chang C. and Lee R.C. (1973): *Symbolic Logic and Mechanical Theorem Proving.* — New York: Academic Press.

Clocksin W.F. and Mellish C.S. (1987): *Programming in Prolog.* — Berlin: Springer–Verlag.

Hachtel G. Jacoby R.M., Keutzer K. and Morrison C.R. (1992): *On properties of algebraic transformations and the synthesis of multifault–irredundant circuits.* — IEEE Trans Computer–Aided Design of Integrated Circuits and Systems, v.11, No.3, pp.313–321.

Kowalski R. (1989): *Logic for Problem Solving.* — Warszawa: WNT Press, (in Polish).

Lloyd J.W. (1987): *Foundations of Logic Programming.* — Berlin: Springer–Verlag.

Malpas J. (1987): *Prolog. A Relational Language and its Application.* — Engelwood Cliffs (NJ): Prentice Hall.

Schalkoff R.J. (1990): *Artificial Intelligence: An Engineering Approach.* — London: McGraw–Hill Publishing Company.

Szajna J., Adamski M. and Kozłowski T. (1991): *Programming in Logic. Turbo Prolog.* — Warszawa: WNT Press (in Polish).

Szajna J. and Kozłowski T. (1991): *Application of Prolog to digital systems design.* — Scientific Notes, Higher College of Engineering in Zielona Góra, (Zeszyty Naukowe WSInż., Zielona Góra, No.95, pp.5–16, (in Polish)).

Thayse A. (Ed.) (1988): *From Standard Logic to Logic Programming.* — Introducing a Logic Based Approach to Artificial Intelligence. -Chichester: John Wiley & Sons.

Towsend C. (1987): *Advanced Techniques in Turbo Prolog.* — San Francisco: Sybex.

Warren D.H.D. (1980): *Logic programing and compiler writing.* — Software – Practice and Experience, v.10, pp.97–125.

Weskamp K. and Hengl T. (1988): *Artificial Intelligence Programming with Turbo Prolog.* — New York: John Wiley & Sons.

Received March 23, 1993
Revised June 3, 1993