Zbornik radova 49. Konferencije za ETRAN, Budva, 5-10. juna 2005, tom III
Proc. 49th ETRAN Conference, Budva, June 5-10, 2005, Vol. III

PARAMETRIC ZOOMING OF SYNTAX FORMS

Aleksandar Perović, Nedeljko Stefanović, Momčilo Borovčanin, Aleksandar Jovanović
Group for intelligent systems (GIS), Mathematical Faculty Belgrade

Abstract *We shall present various procedures for the syntax analysis (in certain formal systems) of the patterns (syntax forms) defined by user which are integrated in an implementation of a proof checker for the propositional calculus.*

1. INTRODUCTION

By syntax zooming we assume variety of algorithms for processing of syntax forms, i.e. algorithms that can allow us to resolve problems such as syntax correctness, pattern matching, proof correctness, and in general, analysis and better understanding of informational bearing structures. In this article we shall describe an implementation of a proof checker for various propositional theories defined by user.

Roughly speaking, propositional theory T is, in our case, a finite nonempty set of propositional formulas. The set of all propositional formulas is the smallest set containing the propositional letters which is closed under logical connectives. In the terms of recursion, the set of all propositional formulas can be defined by the following clauses:

- Propositional letters are propositional formulas.
- If A and B are propositional formulas, then $(\neg A), (A \wedge B), (A \vee B), (A \rightarrow B)$ and $(A \leftrightarrow B)$ are propositional formulas.
- The propositional formula can be obtained only by finite application of the above clauses.

The only inference rule is modus ponens

$$\frac{A \quad A \rightarrow B}{B}.$$

The proof in the given propositional theory T is a finite sequence of formulas where each formula is either axiom of T, or it can be derived from some preceding formulas in the sequence by modus ponens.

2. SYNTAX CORRECTNESS

We shall present in full detail an algorithm for checking the syntax correctness of the given propositional formula by transforming it into a postfix form. To avoid unnecessary brackets, we shall introduce the operator precedence for the logical connectives as follows: negation (highest priority), conjunction and disjunction (the same priority lesser than negation's priority), implication and equivalence (the same priority lesser then conjunction's priority).

We shall need a stack (empty at beginning) for logical connectives and a list (also empty at beginning) for the final postfix form of the given formula. Mentioned list we shall call the *postform*.

We shall also need one indicator variable *ind* (with value *formula* at beginning) with two possible values: *formula* and *connective*. During the reading (token by token) of the usual infix form of the given formula, we shall use this indicator for the saving of the current expectation - subformula or connective.

Now we start with reading of the given formula (token by token) from the input file and then proceed by following rules depending on values of the *ind* and last token.

The value of *ind* **is** *formula*:

- If the last token is propositional letter then we write it at the end of the list, change the value of *ind* into *connective* and read the next token.
- If the last token is the negation or the left bracket, we push it on stack and go on the next token.
- In all other cases we have the syntax incorrectness.

The value of *ind* **is** *connective*:

- If the last token is the negation, then we have the syntax incorrectness.
- If the last token is some other logical connective, then if stack is nonempty and on top of it is a connective of higher priority than the last token, or it is the same as the last token, then we pop it from stack and write it at *postform*. We repeat this procedure as long as it possible.

  After that, if we found on the top of the stack a connective of the same priority as the last token, it must be different from it, so we can conclude the syntax incorrectness.

  The good cases (when we proceed with reading) are when either the stack is empty, or on the top of it are the left bracket or a connective of the lower priority. In those cases we push the last token on the stack, change value of the *ind* to *formula* and read the next token.
- If the last token is the right bracket then:
  - If the stack is empty, then we have the syntax incorrectness.
  - If on the top of the stack is connective, then we pop it from stack and write it on the *postform*.

  We repeat this procedure as long as the stack is nonempty and on the top of it is not the left bracket. If we encounter the left bracket, we pop it from stack and read the next token.

- In all other cases we have the syntax incorrectness.

Let us remark that each occurrence of syntax incorrectness automatically halts the algorithm.

We repeat described procedure as long as there are any tokens left in the input file. If the stack is empty, the *postform* represent the postfix form of the given formula. Otherwise, we pop from the top of the stack each connective. If we encounter the left bracket, then we have the syntax incorrectness.

Example The following table represent a step by step walk through the algorithm for the input formula

$$a \wedge (\neg b \rightarrow c \vee d) \rightarrow e.$$

step	token	*ind*	stack	*postform*
1	a	$formula$		a
2	$\wedge$	$connective$	$\wedge$	a
3	$($	$formula$	$\wedge($	a
4	$\neg$	$formula$	$\wedge(\neg$	a
5	b	$formula$	$\wedge(\neg$	ab
6	$\rightarrow$	$connective$	$\wedge(\rightarrow$	$ab\neg$
7	c	$formula$	$\wedge(\rightarrow$	$ab\neg c$
8	$\vee$	$connective$	$\wedge(\rightarrow \vee$	$ab\neg c$
9	d	$formula$	$\wedge(\rightarrow \vee$	$ab\neg cd$
10	$)$	$connective$	$\wedge$	$ab\neg cd\vee \rightarrow$
11	$\rightarrow$	$connective$	$\rightarrow$	$ab\neg cd\vee \rightarrow \wedge$
12	e	$formula$	$\rightarrow$	$ab\neg cd\vee \rightarrow \wedge e$
13		$connective$		$ab\neg cd\vee \rightarrow \wedge e \rightarrow$.

3. PATTERN MATCHING

First we will need an algorithm for the acquisition of a "right tail" from the given propositional formula which "heart" (i.e. connective with lowest priority) is a binary logical connective. For instance, in the formula $(A \rightarrow B) \wedge C$ the "heart" is $\wedge$, the "left tail" is $A \rightarrow B$, and the "right tail" is C.

Input: propositional formula which heart is a binary connective.

Output: the right tail of the given formula.

- $counter = 1$
- $list = \emptyset$
- Change the matrix of the given formula into postfix form, then start to read it token by token.
- If token = binary connective, then we increase *counter* by 1, write the token at *list* and read the next token.
- If token = unary connective, then *counter* is unchanged, we write the token at *list* and read the next one.
- If token = letter, then we decrease *counter* by 1 and write the token at *list*. If *counter* = 0 algorithm halts and *list* represent right tail. Otherwise, we read the next token.

The pattern matching algorithm

We have formulas F_1 and F_2 in postfix notation. Start with comparing of last tokens in both formulas.

- If last token of F_1 is a connective, then the last token of the formula F_2 must be the same connective. Otherwise, the formula F_2 is not an instance of the formula F_1. If mention tokens are the same, we remove them.
- If the last token of F_1 is a letter, we use earlier described algorithm to acquire the subformula from the end of F_2, and set substitutional condition letter = subformula. The algorithm stops if this condition contradicts some earlier introduced condition.

We continue with this procedure until we remove all tokens from one of the formulas F_1 and F_2. The formula F_2 is an instance of the formula F_1 if described algorithm removes all tokens simultaneously from both formulas.

4. PROOF CORRECTNESS

In this section we shall present the main algorithm, i.e. the integration of previously described procedures.

Input: a finite list of axioms and a finite list of propositional formulas.

Output: YES, if the given list represents a proof in the given propositional theory; NO otherwise.

1. Transform given formulas into postfix form. On any occurrence of syntax incorrectness the algorithm halts. If each formula is correct, then form a sequence from the list of formulas which are not axioms. Let m be a length of the formed sequence and let A_n be an n-th formula of the sequence.

2. $n = 0$

3. Output = YES

4. If $m = n$, then algorithm halts. Otherwise, increase n by 1 and do the following:

   - Check whether A_n is an instance of any of axioms or not. If the answer is positive, goto 4. Otherwise, check whether A_n can be obtained from preceding members of the sequence or not. If the answer is positive, then goto 4. Otherwise, change the value of Output into NO and stop the algorithm.

5. FURTHER RESEARCH

We plan to modify presented proof checker to case of predicate logic. This modification can be used not only for verification of formal proofs in some classical formal systems (such as first order predicate logic, modal logic, Heyting algebra etc.), but also for an analysis of spoken languages.

Our main goal is to develop similar procedures for music scores and DNA chains, since both can be treated as syntax forms and there is a need of finding an adequate similarity criteria (or developing some sort of invariance theory) for such forms.

REFERENCES

[1] Melvin Fitting, "First-Order Logic and Automated Theorem Proving," 1996, 2nd ed., *Springer-Verlag*

[2] Raymond M. Smullyan, "First-Order Logic," 1968, *Springer-Verlag*

[3] S. C. Kleene, "Introduction to Metamathematics," 1952, *North-Holland*

[4] Editors A. Robinson and A. Voronkov, "Handbook of Automated Reasoning," vol 1,2, 2001, *Elsevier Science*