The Syntax Zooming

Aleksandar Perović, Nedeljko Stefanović, Aleksandar Jovanović

GIS (Group for Intelligent Systems), Faculty of Mathematics, Belgrade
E-mail: peramail314@yahoo.com

Abstract: We shall present various procedures for the syntax analysis in certain formal systems defined by the user, which are integrated in the proof checker. The flexibility and the applicability of the method lies in the fact that the program treats formal theories as parameters (both axioms and derivation rules are parts of the input), so developed algorithms work for any formalism which can be expressed within the language of the predicate logic.

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 information bearing structures. We shall present a solution for the mentioned problems in the form of the proof checker. The flexibility and the applicability of the method lie in the fact that the program treats formal theories as parameters (both axioms and derivation rules are parts of the input), so developed algorithms work for any formalism which can be expressed within the language of the first order predicate calculus (recall that the language of the predicate calculus contains just variables, constant, functional and relational symbols, logical connectives, quantifiers, brackets, the comma symbol and the symbol for equality).

Let us proceed with some basic definitions. A predicate theory T is, in our case, a finite nonempty set of predicate axioms, axiom schemata and derivation rules. The notion of a term we define recursively as follows:

- Constant symbols and variables are terms.
- If F is an arbitrary functional symbol of arity n and if $t_1,...,t_n$ are arbitrary terms, then the string $F(t_1,...,t_n)$ is also a term.
- Terms can be obtained only by finite use of the above clauses.

The notion of an atomic formula we shall define by the following two clauses:

- For arbitrary terms t and t' the string $t = t'$ is an atomic formula.
- For arbitrary relational symbol R of arity n and arbitrary terms $t_1,...,t_n$, the string $R(t_1,...,t_n)$ is also an atomic formula.
- Atomic formulas cannot be obtained in any other way.

The notion of a formula we recursively define as follows:

- Atomic formulas are formulas.
- If A and B are arbitrary formulas and if x is an arbitrary variable, then the strings $\neg A$, $(A \wedge B)$, $(A \vee B)$, $(A \to B)$, $(A \leftrightarrow B)$, $\forall xA$ and $\exists xA$ are also formulas.
- Formulas can be obtained only by finite use of the above clauses.

Let A be a formula and let x be a variable which occur in the formula A. If Q is an arbitrary quantifier, then we say that each occurrence of the variable x in the formula QxA is bounded by the quantifier Q. Each occurrence of a variable which is not bounded by some quantifier we shall call the free occurrence.

The set of axioms is any subset of the set of all formulas, and derivation rules are relations on formulas of arity greater than 1. If ξ is a derivation rule and $\xi(A_1,...,A_n,B)$ is true, then we say that formula B is immediate consequence of formulas $A_1,...,A_n$ by the rule ξ.

The proof in the given predicate 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 by any derivation rule.

2 Input Format

The main difficulty lies in the fact that most of classical logics do not have a finite set of axioms. For instance, the classical propositional calculus has the following axioms:

$A \to (B \to A)$, $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$

and $(\neg B \to \neg A) \to (A \to B)$, where A, B and C are arbitrary propositional formulas, so we have a countable (thus infinite) set of axioms. Since we want to treat the classical logics as well, we have to deal with the meta formulas (such as the above schemata) as well.

Back to the format of the input file: We have the following keywords and special symbols:

- The commands for the meta identifiers: **\function_variable, \constant_variable, \relation_variable, \variable_variable, \term_variable, \formula_variable**. For instance, the command **\function_variable**{X} has the following effect: the value of the identifier X ranges over functional symbols.
- Identifiers (variables, functional, relational and constant symbols): **\variable, \constant, \function, \relation, \term**.
- Other commands: **\axiom, \bounded, \free, \term, \substitution, \relation, \rule**.
- Special symbols: ~ /\ \/ => <=> { } ( ) , =

There are no reserved words. The identifiers are nonempty sequences of letters, digits and underscore symbols starting with letter or underscore symbol. The syntax is case sensitive. All white spaces and their sequences are equivalent to one space symbol.

The input file with theory description is an ASCII file containing a sequence of commands of the form

\constant|{*identifier*}

\variable|{*identifier*}

\function| {*identifier*}{*arity*}

\relation|{*identifier*}{*arity*}

\constant_variable|{*identifier*}

\variable_variable|{*identifier*}

\function_variable|{*identifier*}{*arity*}

\relation_variable{*identifier*}{*arity*}

\term_variable|{*identifier*}

\formula_variable{*identifier*}

\axiom{*identifier*}{*description of formula*}{*conditions*}

\rule{*identifier*}{*arity*}{*description of formula*}{*conditions*}

The conditions are commands of the form \free(x,A),

\bounded(x,A),

\partially_substituted(A,x,t,B)

separated by commas, where A and B are formulas or variables for formulas, x is a variable or a variable for variables and t is a term or a variable for terms. For instance, the meaning of the last command is that the formula A can be obtained from the formula B by replacing any free occurrence (not necessary all of them) of the variable x by term t, where all replacements are regular.

Input file which contains the (potential) proof is an ASCII file with nonempty sequence

of descriptions of formulas separated by ";" without variables for constants, variables, functional

symbols, terms and formulas. Description of formula is given in the usual mathematical syntax

(possibly) with the using of the command **\substitution** (A,x,t),

where A is any formula or a variable for formulas, x is any variable or a variable for variables, and t is any term, or any variable for constants or any variable for terms.

3 The Syntax Correctness

We shall present an algorithm for checking the syntax correctness of the given predicate formula by transforming it into a postfix form. We have two recursive procedures: for the extraction of a first term (from left to right) from the given string and writing its postfix form, and for the transformation of the given formula into a postfix form. To simplify notation, by term unit we shall call variables, constant symbols, variables for variables, variables for terms and variables for constant symbols.

The "postfix form of a term" algorithm

Input: a sequence S of tokens (possibly empty). If $m < n$, then we assume that the sequence $s_n, \ldots, s_m$ is empty.

Output: a sequence $s_k, \ldots, s_n$ of the tokens that are not scanned yet; an error report; as a side effect, an appending in the global variable Postfix of the postfix form of the first subterm (from left to right) of the given sequence.

- If S is empty, then the procedure halts with an error report.
- If s_1 is a term unit, then append s_1 in Postfix and return the rest of the S.
- If s_1 is neither a term unit nor a functional symbol, then the procedure halts with an error report.

- If s_1 is a functional symbol (or a variable for functional symbols) and s_2 is not a left bracket, then the algorithm halts with an error report.
- If s_1 is a functional symbol of arity k (or a variable for functional symbols of arity k), and s_2 is a left bracket, then:

  (1) Initialize i to 1.

  (2) Recursively call the function with the sequence $s_3,\ldots,s_n$ as input. If the return of the recursive call is an error report, then the procedure halts with an error report. Otherwise let us denote the returned string by $s_1,\ldots,s_l$.

  (3) If $l < 1$ then the procedure halts with an error report.

  (4) If $i < k$ and s_1 not equal to “,” or if $i = k$ and s_1 differs from “)”, then the procedure halts with an error report.

  (5) If $i < k$ then increment i by 1 and go to step (2). Otherwise, append scanned functional symbol/variable for functional symbols in Postfix.

The “postfix form of a formula” algorithm

Input: a sequence S of tokens (possibly empty). If $m < n$, then we assume that the sequence $s_n,\ldots,s_m$ is empty.

Output: a sequence $s_k,\ldots,s_n$ of the tokens that are not scanned yet; an error report; as a side effect, an appending in the global variable Postfix of the postfix form of the first subformula (from left to right) of the given sequence.

- If S is empty, then the procedure halts with an error report.
- If s_1 is a variable for formulas, then append it in Postfix and return $s_2,\ldots,s_n$.
- If s_1 is a term unit, or a functional symbol, or a variable for functional symbols then:
  - Call the “postfix form of a term” procedure with the same input. On error report the procedure halts with an error report. Otherwise, let us assume that the return is a sequence $s_1,\ldots,s_l$.
  - If s_1 differs from “=”, the procedure halts with an error report.

- Call the “postfix form of a term” procedure with input $s_2,\ldots,s_l$. On error report the procedure halts with an error report. Otherwise, append “=” in Postfix and return the same sequence as the “postfix form of a term” procedure.

- If s_1 is equal to “~” then recursively call this same procedure with the sequence $s_2,\ldots,s_n$ as input. On error report the procedure halts with an error report. Otherwise append “~” in Postfix and return the same sequence that is returned from the recursive call.
- If s_1 equals to “(“ then:
  - Recursively call this same procedure with the sequence $s_2,\ldots,s_n$ as input. On error report the procedure halts with an error report. Otherwise, let us assume that the returned sequence is $s_1,\ldots,s_l$.
  - If $l < 1$ or s_1 is not a binary logical connective, then the procedure halts with an error report. If not, memorize s_1.
  - Call this same procedure with $s_2,\ldots,s_l$ as input. On error report the procedure halts with an error report. Otherwise, let us assume that the returned sequence is $s_1,\ldots,s_m$.
  - If $m < 1$ or s_1 differs from “)”, the procedure halts with an error report. Otherwise append the memorized connective in Postfix and return the sequence $s_2,\ldots,s_m$.
- If s_1 is a quantifier then:
  - Memorize it.
  - In $n < 2$ or s_2 is neither a variable nor a variable for variables, then the procedure halts with an error report. Otherwise memorize s_2.
  - Recursively call this same procedure with the sequence $s_3,\ldots,s_n$ as input. On error report the procedure halts with an error report. Otherwise, append in Postfix first the memorized variable, then the memorized quantifier, and return the same sequence that is returned from the recursive call.
- If s_1 is a relational symbol (or a variable for relational symbols) and s_2 is not a left bracket, then the procedure halts with an error report.

- If s_1 is a relational symbol of arity k (or a variable for a relational symbol of arity k) and s_2 is a left bracket, then:

  (1) Initialize i to 1.

  (2) Call the “postfix form of a term” procedure with the sequence $s_3,...,s_n$ as input. If the return of the call is an error report, then the procedure halts with an error report. Otherwise let us denote the returned string by $s_1,...,s_l$.

  (3) If $l < 1$ then the procedure halts with an error report.

  (4) If $i < k$ and s_1 not equal to “,” or if $i = k$ and s_1 differs from “)”, then the procedure halts with an error report.

  (5) If $i < k$ then increment i by 1 and go to step (2). Otherwise, append scanned relational symbol in Postfix.

- If s_1 is the operator **substitution** and s_2 is not a left bracket, then the procedure halts with an error report.

- If s_1 is the operator **substitution** and s_2 is a left bracket, then:

  (1) Recursively call this same procedure with the sequence $s_3,...,s_n$ as input. On error report the procedure halts and returns an error report. Otherwise let us denote the returned sequence by $s_1,...,s_l$.

  (2) If $l < 4$ or at least one of s_1 and s_3 differs from “,”, or s_2 is not a variable (or a variable for variables), then the procedure halts with an error report. Otherwise, append the scanned variable in Postfix.

  (3) Call the “postfix form of a term” procedure with the sequence $s_4,...,s_l$ as input. On error report the procedure halts with an error report. If not, let us denote the returned sequence by $s_1,...,s_m$.

  (4) If $m < 1$ or s_1 differs from the right bracket, then the procedure halts with an error report. If not, append **substitution** in Postfix and return $s_2,...,s_m$.

4 The Pattern Matching

In this section we shall present the procedure that checks whether the given formula B is an instance of the given axiom A or not. We shall give the detailed algorithm for the propositional case. The predicate case is much more complicated (and lengthy as well), so we shall give just the brief sketch of that case.

The first subterm procedure

Input: a propositional formula in postfix form.

Output: the first subtrerm (from right to left) of the given formula

- Initialization: *Counter* = 0, *List* = empty.
- If the scanned token is a binary connective, then we increase *Counter* by 1, append the token in *List* and scan the next token.
- If the scanned token is the negation, then we scan the next one.
- If the scanned token is propositional letter, then we append the token in *List* and decrease C*ounter* by 1. If *Counter* = 0, then the procedure halts and returns *List*, otherwise we scan the next token.

Propositional pattern matching

Input: a pattern A (i.e. a Boolean combination of formula variables), and a propositional formula B .

Output: YES, if the formula B is an instance of the pattern A ; NO otherwise.

- Transform A and B into postfix form and then start to scan them simultaneously token by token from right to left.
- The scanned token of the pattern A is a logical connective. If the scanned token of the formula B is not the same connective, then the matching is impossible, so the procedure returns NO. In the case of matching we scan the next token in both formulas. The algorithm halts and returns NO if one of A and B is exhausted by the scanning process and the other is not. If both A and B are scanned, the procedure returns YES.
- The scanned token of the pattern A is a letter. Then we extract the subformula of the formula B which starts with the current scanned token of B , and then we set the condition: letter = subformula. The procedure returns NO if this condition contradicts some of earlier introduced conditions. On soundness we scan the next token of A and the first token of B (from right to left) which doesn't belong to the extracted subformula. The algorithm halts and returns NO if one of the formulas is exhausted by the scanning process and the other is not. If both formulas are scanned, the procedure returns YES.

The pattern matching procedure for the predicate case is recursive one and has several subprocedures such as term matching, determination of propositional structure of the given predicate formula, matching of propositional structures and unification in the presence of substitution.

5 The Proof Correctness

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

Input: a representation of a predicate theory and list of predicate formulas.

Output: YES, if the given list represents a proof in the given predicate 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 the n-th formula of the sequence.

(2) n=0

(3) Output = YES

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

Check whether A_n is an instance of an axiom or not. If the answer is positive, go to (4). Otherwise, check whether A_n can be obtained from preceding members of the sequence by any derivation rule or not. If the answer is positive, then go to 4. Otherwise, change the value of Output into NO and stop the algorithm.

6 Further Research

We plan to modify presented proof checker to wide class of logics. 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 natural languages.

Our main goal is to develop similar procedures for the information bearing structures such as music scores representation and sequences in molecular biology, 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*, Springer-Verlag 1996

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

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

[4] **A. Robinson, A. Voronkov** (editors), *Handbook of Automated Reasoning*, vol 1,2, Elsevier Science 2001

[5] **J. Shoenfield**: *Mathematical logic*, Addison-Wesley 1967

[6] **A. Jovanović**: *Group for Intelligent Systems - Problems and Results*, Intelektualnie sistemy, Lomonossov Un and RAN, 6, 2002