

AN AUTOMATIC THEOREM PROVER GENERATING A PROOF IN NATURAL LANGUAGE

Masakazu Nakanishi, Morio Nagata and Kenji Ueda
Faculty of Engineering, KEIO University
Hiyoshi, Yokohama, 223 JAPAN

An automatic theorem prover which displays a proof in natural language is described. This system proves properties of recursive programs, and constructs a proof tree corresponding to the proof. Then it translates the tree into the proof text in English by means of the tree traverse. The proof written by this system is easy to read, because the English text is placed instead of notations of logics, and some redundant and trivial statements are disappeared.

1. INTRODUCTION

Many automatic theorem provers have been implemented to verify programs or to prove properties of programs [2,5]. Some of them are based on a formal system in logics. However, users of such theorem provers are often confused by technical terms of logics appeared in the proof. And so it is desired that the system should display the proof in natural language without proper terms of logics. Chester [1] gives an algorithm which translates formal proofs into English. But we rarely find the systems which have such facilities. An algorithm to generate statements which are suitable to the context of a proof for a Gentzen-type formal system representing properties of functions is described.

TKP (Tsukuba-Keio Prover) is a family of theorem provers based on the formal system. As an ordinary prover, it combines a sequent calculus (decomposing a complicated theorem into parts which are easier to prove). The sequent is decomposed by applying the rules of inference. TKP displays a proof figure if the proof of a given theorem is completed. The proof figure consists of logical symbols and terms which are used properly in logics. Though these terms are important tools to prove theorems automatically, they are regarded as codes which only logicians or experts in artificial intelligence can read.

In this paper, we describe the method for displaying proofs in natural language without proper terms of logics.

2. OUTLINE OF TKP

The original version, TKP1 [4], proves

properties of recursive programs and generates the proof figures. TKP2, described in this paper, generates effective proofs in natural language furthermore. The formal system of TKP includes the propositional calculus of Gentzen's LK and additional rules to prove properties of recursive programs. Its completeness is shown by T. Nishimura [3].

Our processor accepts a sequent to be proved and user supplied knowledge, then it generates a tree of the proof of the given problem.

Def: A sequent is called a node.

Def: If A is a sequent decomposed from B by applying a rule, the name of the rule is the indicator from B to A.

Def: A proof tree is defined as follows.

1. A node is a proof tree.
2. If m is a sequent, $t_1, t_2, \dots, t_n$ are proof trees and $i_1, i_2, \dots, i_n$ are indicators from m to $t_1, t_2, \dots, t_n$ respectively, then

```

m
|
| i1
*---- t1
|
.
.
| in
*---- tn

```

is a proof tree.

3. The only proof trees are those given by 1 and 2.

If TKP proves the given problem, the proof tree will be generated (see APPENDIX in [4]). The later passes (analysis, reduction, generation), converts it into an English text.

3. ANALYSIS AND REDUCTION

In analysis, each proposition, called a P-type formula, is valuated and formulas of each node are simplified, and an analyzed tree is generated.

Def: An environment is a list of pairs of a P-type formula and the truth value. The initial state of the environment is empty.

The algorithm of analysis is as follows:

Traverse the given proof tree from its root.

When a sequent S including the P-type formula p which does not occur in the environment, split the tree T whose root is S . Analyze T recursively in the new environment which is appended the pair of p and true to the environment E . Next, analyze T in the environment which is appended the pair of p and false to E . Construct a new tree T' whose root is S simplified in E and, whose subtrees are the above two analyzed trees (Fig. 1).

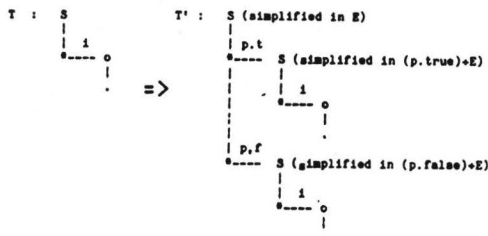


Fig. 1 Analysis (1)

When a sequent S including P-type formulas which occur in the environment is encountered, simplify S (the rules of simplification is in Fig. 3). However, if S is transformed from S_1 by the user supplied knowledge or the induction hypothesis, and P-type formulas occur in S by the replacement, simplify S except the formula replaced. The new node is named S' . Next, analyze the tree whose root is S , append the indicator VAL and the analyzed tree to S' (Fig. 2).

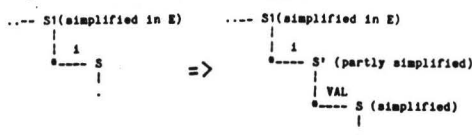


Fig. 2 Analysis (2)

In reduction, the analyzed tree is reduced and a final tree is recomposed by removing redundant nodes or subtrees from the analyzed tree.

The algorithm of reduction is as follows:

Traverse the analyzed tree.

If a node is identical to its son, remove the node together with the branch.

If a node satisfies either (I) or (II), remove the subtree connected with the node.

(I) The node becomes a logical axiom.

(II) $\Gamma \rightarrow \Delta$ is already proved and

The node is $\Gamma \rightarrow \Delta_1, \Delta_2$

1: P-type formulas are replaced by their truth value, if exist in the environment.

2: Formulas and sequents

- a) $A \& t, t \& A, f \vee A, A \vee f \Rightarrow A$
- b) $A \& f, f \& A, \neg t \Rightarrow f$
- c) $A \vee t, t \vee A, \neg f \Rightarrow t$
- d) $\Gamma_1, t, \Gamma_2 \rightarrow \Delta \Rightarrow \Gamma_1, \Gamma_2 \rightarrow \Delta$
- e) $\Gamma \rightarrow \Delta_1, f, \Delta_2 \Rightarrow \Gamma \rightarrow \Delta_1, \Delta_2$

Fig. 3 Rules of simplification

4. GENERATION OF A PROOF TEXT

Finally, the proof text in English is generated. A skeleton of the proof text is prepared for each type of tree. A skeleton is a rule of conversion of a tree to strings of characters. A typical skeleton is shown in Fig. 5. Traversing the reduced proof tree, a proof text is constructed by applying the skeletons.

In principle, a tree is traversed from root to leaf. A tree free from branch, however, is traversed from leaf to root (Fig. 4).

APPENDIX II is generated from the final proof tree in APPENDIX I which is the reduced proof tree of the proof shown in [4].

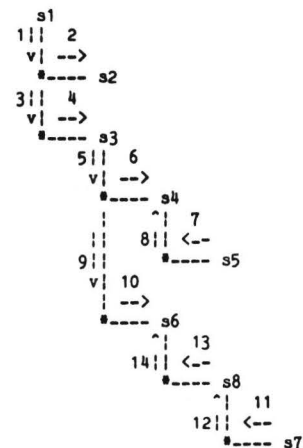


Fig. 4 Tree traverse

```

[ M ] z.
|
| We distinguish two cases.
|
| CASE (1): Assume that P is true, the
| following statement can be obtained.
|----- T1
|
| <Proof text of T1>
|
| P.f
|----- T2
|
| Thus CASE (1) is proved.
|
| CASE (2): Assume that P is false, the
| following statement can be obtained.
|
| <Proof text of T2>
|
| Thus CASE (2) is proved.

```

Fig. 5 An example of skeleton

5. CONCLUSION

Our system is implemented in about 1000 lines of Lisp program. In actual program, analysis, reduction and generation are executed at the same time. These are implemented in about 600 lines. Our algorithm, except for the last pass, is independent of languages, then TKP2 can write in other language by changing the skeletons of the statements.

The proof written by this prover is very easy to read, because the English sentences are placed instead of the logical symbols, and some redundant or trivial statements are disappeared. We believe that this system will be used as a training tool of program verification.

ACKNOWLEDGEMENT

We would like to thank Prof. Toshio Nishimura for several very helpful comments.

APPENDIX I

```

F<?>(K) -> G<?>(K,1)
|L-v
*--F<0>(K) -> G<?>(K,1)
|L-v
*--F<N+1>(K) -> G<?>(K,1)
|R-v
*--F<N+1>(K) -> G<N+1>(K,1)
|D-1
*--P(K)&1 v \P(K)&KxF<N>(K-1) -> G<N+1>(K,1)
|P.t
*--1 -> G<N+1>(K,1)
|D-2
*--1 -> P(K)&1 v \P(K)&G<N>(K-1,Kx1)
|VAL
*--1 -> 1
|P.f
*--KxF<N>(K-1) -> G<N+1>(K,1)
|HYP
*--KxG<N>(K-1,1) -> G<N+1>(K,1)
|D-2
*--KxG<N>(K-1,1) -> P(K)&1 v \P(K)&G<N>(K-1,Kx1)
|VAL
*--KxG<N>(K-1,1) -> G<N>(K-1,Kx1)
|T-1
*--G<N>(K-1,Kx1) -> G<N>(K-1,Kx1)

```

APPENDIX II

```

Prove that
if a = F-?-(K) then a = G-?-(K,1)
where
1. F-N+1-(X) = P(X)&1 v \P(X)&KxF N (X-1)
2. G-N+1-(X,Y) = P(X)&Y v \P(X)&G N (X-1,KxY)
using the theorem
1. if a = KxG-N-(Y,2) then a = G N (Y,Kx2)
PROOF
[1] if a = F-?-(K) then a = G-?-(K,1)
will be proved by induction on N.
BASIS, F-0-N = 0.
[2] if a = F-0-(K) then a = G-?-(K,1).
This is valid, because F 0 (K) is undefined.
INDUCTION STEP. Assume (as the induction hypothesis)
that for every l < N+1,
if a = F-l-(K) then a = G-l (Y,1).
Now let's prove
[3] if a = F-N+1-(K) then a = G-?-(K,1).
It is sufficient to prove that
[4] if a = F-N+1-(K) then a = G-N+1-(K,1).
By definition 1, [4] is equivalent to
[5] if a = P(K)&1 v \P(K)&KxF-N-(K-1)
then a = G-N+1-(K,1).
We distinguish two cases.
CASE (1): Assume that P(K) is true. The following
statement can be obtained.
[6] if a = 1 then a = G-N+1-(K,1).
Now, consider the following valid statement.
[7] if a = 1 then a = 1.
Since P(K) is true, [7] may be written as follows:
[8] if a = 1 then a = P(K)&1 v \P(K)&G N (Y-1,Yx1).
By definition 2,
[9] if a = 1 then a = G-N+1 (K,1).
Thus CASE (1) is proved.
CASE (2): Assume that P(K) is false. The following
statement can be obtained.
[10] if a = KxF N-(K-1) then a = G-N+1 (Y,1)
Now, consider the following valid statement.
[11] if a = G N (K-1,Kx1) then a = G N (Y-1,Yx1)
Since P(K) is false, [11] may be written as follows:
[12] if a = G N (K-1,Kx1)
then a = P(Y)&1 v \P(Y)&G N (X-1,Yx1)
By definition 2,
[13] if a = G N (K-1,Kx1) then a = G-N+1 (Y,1)
By theorem 1,
[14] if a = KxG N (Y-1,1) then a = G-N+1 (Y,1).
By the induction-hypothesis,
[15] if a = KxF N (Y-1) then a = G-N+1 (Y,1).
Thus CASE (2) is proved.
This completes the proof.

```

REFERENCES

- [1] Chester, D. "The translation of formal proofs into English." *Artif. Intell.* 7:3 (1976) 261-278.
- [2] King, J.C. "A program verifier." PhD thesis, Carnegie-Mellon Univ., 1969.
- [3] Nishimura, T. "Gentzen-type formal system representing properties of functions." *Comment. Math. Univ. St. Pauli.* 23:1 (1974) 37-44.
- [4] Nishimura, T., Nakanishi, M., Nagata, M. and Iwamaru, Y. "Gentzen-type formal system representing properties of functions and its implementation." In *Proc. IJCAI-75*, Tbilisi, Sep., 1975, pp. 57-64.
- [5] Suzuki, N. "Verifying programs by algebraic and logical reduction." In *Proc. of Int. Conf. Reliable Software*, 1975, pp. 473-481.