数理解析研究所講究録
第236巻 1975年 46-75

Implementation of Gentzen-Type Formal System Representing Properties of Functions*

Yoshiaki Iwamaru, Morio Nagata and Masakazu Nakanishi
Keio Univ.
Toshio Nishimura
Tokyo Univ. of Education

ABSTRUCT. Gentzen-type proving method formulated by T. Nishimura has been implemented. This theorem prover is written in KLISP which is a subset of LISP 1.5. The input and output form is designed nearly to the natural notation in mathematics. When this prover succeeds in proving a problem, its proof figure will be displayed on line printer. Our processor can be used to investigate the properties of the functions. Using the induction and fixed point theorem, we got respective proofs for properties of recursive programs.

1. *Introduction*

In this paper we describe a kind of the implementation of the proving method in Gentzen-type formulation presented in [2]. Our processor is programmed in LISP language. The way to use this theorem prover and the outline of its processor will be shown.

The features of this theorem prover are the following.

(1) Assumptions, definitions and a sequent to be proved, we call them a problem, can be provided in the same style as

* This is partly supported by CUDI foundation.

we usually use in mathematical notation. With a processor written in LISP, it is usually very annoying to have to use an input form with many parentheses. Using the pseudo functions *advance* and *prin1*, we can design the processor which has natural input and output facilities.

(2) This processor includes efficient processing methods of propositional calculus, composition of functions and infinitary sum. Then we can treat with the fixed point operator. By this we can prove the correctness or the equivalence of programs which include loops or recursive definitions.

(3) After the proof procedures for some problems are completely made, corresponding proof figures representing the proving processes are printed on the line printer. Each proof figure is an upside down form of usual one of Gentzen-type.

2. *Grammer of Gentzen-Type Theorem Prover*

The input for this processor are assumptions, definitions and a sequent to be proved. Each of assumptions, definitions or a sequent is called a statement. A statement is usually written in a line (72 columns). If a statement continues two or more lines, we write a period at 72nd column of the previous line for presenting continuation.

2.1 *Sequent*

A sequent is generally written in the following.

$$l_1 , l_2 , \cdots , l_n \;\text{->}\; r_1 , r_2 , \cdots , r_m \qquad (n, m \geq 0)$$

where each l_i, r_j is called an expression. An expression is constructed with identifiers (alphanumeric characters of no more 32, beginning with a letter, where a letter is defined as an alphabetic letter except the letter "V") and operators as follows.

e	$(\ e\ )$	$\textcircled{u}\,e$	$e_1 \textcircled{b} e_2$	$f(e_1,e_2,\ldots,e_n)$
$?f$	$?f(e_1,e_2,\ldots,e_n)$	$f\langle n\rangle$		$f\langle n\rangle(e_1,e_2,\ldots,e_n)$

where $\textcircled{u}$ denotes a unary operator, $\textcircled{b}$ a binary operator, e and e_i an identifier or an expression.

The precedence and the meaning of operators are shown in Table 2.1. If two or more successive operators in an expression have the same precedence, the expression is recognized as if the operators are executed from left to right. For example, $a + b + c$ is recognized as $(a + b) + c$ but not $a + (b + c)$.

A fixed point operator is denoted as "?". A labeled function $f\langle n\rangle$ represents that we apply f n times*. Undefined element "Ω" is represented as "%", "ω" is as "@".

---

* In [2], $?f$ is represented as ∂f, $f\langle n\rangle$ is as f^n.

TABLE 2.1 Operator Precedence Table

Precedence	*Operator Symbols*	*Meanings*
1	.	concatenation
2	↑	exponentiation
3	* , /	multiplication, division
4	-	unary minus
5	+ , -	addition, subtraction
6	=	equality
7	\	negation (∿)
8	&	and (∧)
9	V	or (V)
10	>	implication (⊃)
11	:=:	equivalence (≡)

2.2 *Assumptions*

The general form of an assumprion is

$$e_1 \text{ -> } e_2$$

where e_1 or e_2 is an expression. An arbitrary expression is denoted as

$$\$v$$

where v is an identifier, which is called an arbitrary pattern.

For example, the distributive law of addition and multiplication is written as

```
$X * ($Y + $Z)  ->  $X * $Y + $X * $Z
```

Assumptions are numbered automatically by the order of appearance. When a proof figure is displayed, the number of applied assumption is indicated where it applied.

2.3 *Definitions*

The general form of a definition is

$$:p \;=\; e$$

where p is usually $f\langle n\rangle(x_1,x_2,\ldots,x_m)$ $(m \geq 0)$, n must be "N" or "N + i" (i = 0, 1, 2, ...) and e is an expression. A definition is considered as a kind of assumptions by the processor, but a formal parameter x_i needs not to be an arbitrary pattern $\$X_i$. For example,

$$f^{n+1}(x) = p(x)\ 1\ V\ \backslash p(x)\ x*f^{n}(x-1)$$

is written as*

---

* Using the rule of assumption and rules of algebra, this definition may be written as

```
F<$N+1>($X) = P($X)&1 V \P($X)&$X*F<$N>($X-1)
```

```
:F<N+1>(X) = P(X)&1 V \P(X)&X*F<N>(X-1)
```

2.4 *Other Rules*

The hypothesis of associative law is built in this prover. The associative law is applied for ., * and +. This hypothesis is the same effect as following assumption.

$A *op* ($B *op* $C) -> $A *op* $B *op* $C

where *op* is one of ., * and +.

Assumprions written by user are applied before the application of the associative law, so we must write the following assumption if we want to apply the inverse of above hypothesis.

$A *op* $B *op* $C -> $A *op* ($B *op* $C)

McCarthy's conditional expression is based on the form of if-then-else. For example,

if p then e_1 else e_2

is represented as

$p \,\&\, e_1 \vee \backslash p \,\&\, e_2$

An expression p like above is called p-type expression. A p-type expression has the following restrictions.

(1) An identifier beginning with the character "O", "P" or "Q" is a p-type expression.

(2) An expression $p(x_1,x_2,\ldots,x_n)$ is a p-type expression, where p is an identifier beginning with the character "O", "P" or "Q".

In proving process, the following rules are applied to the expression which contains p-type expressions.

(1) $f(p)$ is decomposed into p.

(2) $f(p\ op\ k)$ or $f(k\ op\ p)$ is decomposed into $p\ op\ f(k)$ or $f(k)\ op\ p$ respectively, where op is a logical connective.

(3) $f(\backslash p)$ is decomposed into $\backslash p$,

where p is a p-type expression and k is not.

3. *Processor*

Our processor of Gentzen-type theorem prover consists of three phases: translator, prover and visualizer. These are successively executed. Problems are read by translator, and are translated into S-expressions. Then they are put into the auxiliary memory. Prover gets problems in S-expressions translated before, proves them and puts proof trees into the auxiliary memory. Visualizer gets these proof trees, and write proof figures on line printer, which is easy to see for men. The process is shown in Fig. 3.1.

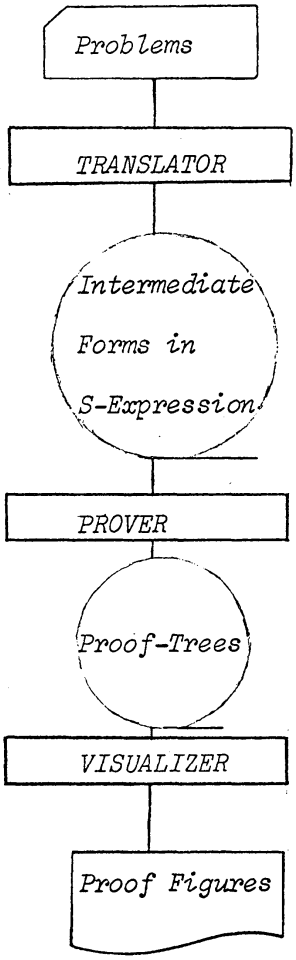


FIG. 3.1 Block Diagram of Gentzen-Type Theorem Prover

This processor is written in KLISP which is a subset of LISP 1.5. Pseudo functions *mprint* and *mread* are used for input and output of S-expressions, *advance* for input of arbitrary characters, *prin1* and *terpri* for output of arbitrary characters.

3.1 *Translator*

Using the following function*, translator reads a line. Then, applying well known parsing method**, it translates statements into prefixed forms.

```
input[] = [eq [eor ; advance[] ] -> nil ;
           t -> cons [curchar ; input[] ] ]
```

Translation rules are as follows:

(1) An identifier is translated into an atomic symbol, "@" and "%" are into "UNDEF".

(2) $-e$, $\backslash e$ is translated into (MINUS e*), (*NOT e*) respectively.

(3) e_1 *op* e_2 (*op* is a binary operator which is +, -, *, /, =, &, V, :=:, etc.) is translated into (*op** e_1* e_2*). The correspondence of *op* and *op** is shown in Table 3.1.
If two or more successive operators have the same precedence, they are translated into as the following example.

$a + b + c$ -> (PLUS (PLUS a* b*) c*)

---

* So as to be able to write a sequent (or an assumption) more than 72 characters, more complicated function is actually used.

** The method of recursive procedure using operator precedence table.

(4) The arbitrary pattern $\$v$ in an assumption is translated into (= v*).

(5) $f\langle n\rangle$ is into (LABEL f* n*).

(6) $?f$ is into (* f*).

(7) $f(e_1,e_2,\ldots,e_n)$ is into (f* e_1* e_2* ... e_n*).

where the symbol with * on its upside represents translated forms.

TABLE 3.1 Translation Table of Operators

op	*op**
.	CONC
↑	EXPT
*	TIMES
/	DIV
(unary) -	MINUS
+	PLUS
-	DIFFERENCE
=	EQSIGN
\	*NOT
&	*AND
V	*OR
>	*IMPLIES
:=:	*EQUIV

The order of translation is the following.

(1) Translation of sequents: A sequent is translated according to translation rules, then its translated form in S-expression is put into the auxiliary memory.

(2) Translation of assumptions and definitions: Assumptions and definitions are successively translated, assumption numbers and flags indicating definitions are added. Assumptions and definitions are translated into a list with the following form and put into the auxiliary memory.

(*assumption** ... *assumption** *definition**
... *definition**)

where *assumption** has the form

(*ass1** . (*ass2** . *n*))

and *ass1**, *ass2** is a translated form of left part, right part of the assumption respectively. *n* is the assumption number. *definition** is

(*def1** . (*def2** . L))

where *def1** is a translated form of

$f\langle n\rangle\ (x_1, x_2, \ldots, x_m)\quad (m \geq 0)$

Every formal parameter x_i is translated into (= x_i*).
*def2** is a translated form of right part of definition.
Every formal parameter in the right part which is also in the left part is translated into (= x_i*), too. "L" is an indicator for a definition.

3.2 *Prover*

The prover consists of main control part called proof controller and several sequent manipulation functions. Proof controller controls and applies sequent manipulation functions for the problem. A sequent manipulation function has a type which is NORMAL or ABNORMAL.

A NORMAL type function makes the object sequent matching each pattern. If pattern matching succeeds, the value of the application of the function becomes a list of transformed sequents. When proof controller applies a NORMAL type function and the matching fails, then it applies the next NORMAL type function. If pattern matching succeeds, the list of transformed sequents is successively proved by proof controller, to construct a proof tree. If a sequent of the list fails, the value of proof controller is "FAIL". There is no automatic backtracking facility in NORMAL type functions. While ABNORMAL type functions have built-in backtracking facilities.

3.2.1 *Proof Tree*

The value of proof controller is a tree constructed with S-expression. We call it proof tree. The proving process is recorded in the proof tree. The form of proof tree is as follows.

(VALID . *sequent*) (1)

(*ind sequent proof tree ... proof tree*) (2)

When a same expression appears in both parts of a sequent or when there is ω (expressed as "UNDEF") in the left part of a sequent, proof tree (1) is constructed. When a sequent is transformed to some sequents and each of them is proved, proof tree (2) is constructed. The indicator *ind* which is a specified atomic symbol shows a way of the transformation. The second element of the list (2) is the sequent before transforming.

For example, we prove

A , A>B -> B

its proof figure is in Fig 3.2 and its proof tree is the following.

```
(DEF-OF-*IMPLIES
     ( (A  (*IMPLIES  A  B) )  (B) )
     (LEFT-OR-ELIM
          ( (A  (*OR  (*NOT  A)  B) )  B)
          (NOT-ELIM
               ( (A  (*NOT  A) )  (B) )
               (VALID . ( (A)  (A  B) ) ) )
          (VALID . ( (A  B)  (B) ) )
```

where DEF-OF-*MPLIES, LEFT-OR-ELIM and NOT-ELIM are indicators*.

$$\dfrac{\dfrac{\dfrac{A \;\text{->}\; A\ ,\ B}{A\ ,\ \backslash A \;\text{->}\; B} \qquad A\ ,\ B \;\text{->}\; B}{A\ ,\ \backslash AVB \;\text{->}\; B}}{A\ ,\ A{>}B \;\text{->}\; B}$$

FIG. 3.2 Proof Figure of A , A>B -> B

---

* Indicators of our actual processor have more simplified forms for saving space.

3.2.2 *NORMAL Type Functions*

NORMAL type functions are divided into two groups. Two VALID functions are in one group. One of them is a function that tests if there exists a same expression in both parts of a sequent. Another is a function that checks whether there exista the undefined element ω in the left part of a sequent or not. The second group consists of functions for the transformation of sequents. Elimination of logical symbols and decomposition of functions are in this group.

VALID functions provide the terminal value of the proof controller, that is (VALID . *sequent*).

There are 8 functions for transformation of sequents dependent upon logical symbols. They are listed as follows.

(1) Search an expression with "*NOT" in the top level in the left part. If exists, then remove "*NOT" from the expression and move the new expression to right part.

(2) Search an expression with "*NOT" in the top level in the right part. If exists, then remove "*NOT" from the expression and move the new expression to left part.

(3) Search an expression with "*AND" in the top level in the left part. If exists, then remove "*AND" and generate two side expressions.

(4) Search an expression with "*OR" in the top level in the right part. If exists, then remove "*OR" and generate two side expressions.

(5) Search an expression with "*OR" in the top level in the left part. If exists, then generate two sequents, one includes left operand of "*OR" instead of the expression, the other does right operand.

(6) Search an expression with "*AND" in the top level in the right part. If exists, then generate two sequents, one includes left operand of "*AND" instead of the expression, the other does right operand.

(7) Every ">" in the top level of a sequent is transformed according to the following definition.

$$a>b \;\rightarrow\; \backslash a \vee b$$

(8) Every ":=:" in the top level of a sequent is transformed according go the following definition.

$$a:=:b \;\rightarrow\; a>b \,\&\, b>a$$

The decomposition procedure of functions is as follows.

(9) i. The expression whose subexpression is p-type and is not connected with a logical symbol will be transformed only into the p-type subexpression.

(9) ii. If an expression has subexpressions and also there exists p-type expression in their operands, the expression will be transformed into two subexpressions combined with the logical symbol.

For example, if p is a p-type expression, the following transformation will be done.

$f(p) \rightarrow p$

$f(pVk) \rightarrow pVf(k)$

$f(p\&e_1 V\ p\&e_2) \rightarrow p\&f(e_1)Vp\&f(e_2)$

The transformation procedure for f^{ω}, ω is as follows.

(10) f<0> is transformed to %. $\%(e_1,e_2,\ldots,e_n)$ is transformed to @.

The value of each function 1~4 and 7~10 is a list of a transformed sequent. The value of 5 or 6 is a list of two transformed sequents.

3.2.3 *ABNORMAL Type Functions*

A function for transforming fixed point operator and a transformation function for assumptions and definitions are ABNORMAL type functions.

The order of transformation of fixed point operator is as follows. $ex(x)$ denotes a expression with subexpression x.

(1) $ex(?f)$ is transformed to $ex(f$<0>$)Vex(f$<N+1>$)$.

(2) Proof controller will be called to attempt to prove the transformed expression. If proved, the value of this function will be the value of proof controller.

(3) If the value of proof controller is "FAIL" $ex(?f)$ is transformed to $ex(f$<0>$)Vex(f$<1>$)Vex(f$<N+2>$)$.

(4) Proof controller will be called to attempt to prove the transformed expression. If proved, the value of this function will be the value of proof controller. If fail to prove, the value is "FAIL".

In matching process for assumptions and definitions, it is first tried to make assumptions match, and definitions next. If an assumption matches two or more subexpressions of a sequent, it is first applied to the leftmost and innermost expression.

If an assumption (or a definition) matches a subexpression, right part of the assumption (or the definition) is substituted for the subexpression. Proof controller will be called to try proving this new sequent. If proof controller can not prove it, the sequent will be tried to apply to other subexpression with the same assumption. If the proof can succeed for no subexpression in the sequent, the next assumption (or definition) will be applied.

If a sequent transformed by an assumption (or a definition) is the same form as a sequent that has ever been in the proving process, it is considered that the matching fails. This procedure is provided for preventing to occur an endless loop.

3.2.4 *The Order of Application of Sequent Manipulation Functions*

The order of application has much influence on efficiency of the proving process. In some cases, the proof may be impossible. Our prover provides the standard order of application, but the order is changeable. The standard order of application, the feature and type of sequent manipulation functions are shown in Table 3.2.

TABLE 3.2 Standard Order of Application of Sequent Manipulation Functions

Order	*Function Name*	*Feature*	*Type*
1	check	check same expression in both parts of a sequent	NORMAL
2	undefcheck	check undefined element in the left part	NORMAL
3	*notl	elimination of \ in the left part	NORMAL
4	*notr	elimination of \ in the right part	NORMAL
5	*andl	elimination of & in the left part	NORMAL
6	*orr	elimination of V in the right part	NORMAL
7	*orl	elimination of V in the left part	NORMAL
8	*andr	elimination of & in the right part	NORMAL
9	*implies	change an expression by definition of >	NORMAL
10	*equiv	change an expression by definition of :=:	NORMAL
11	undef	rewrite f^0 to % and %(*e*) to @	NORMAL
12	decomp	decomposition of functions	NORMAL
13	fixedpoint	decomposition of fixed point operator	ABNORMAL
14	assump	apply assumptions and definitions	ABNORMAL
15	conc	apply associative law	ABNORMAL

3.2.5 *Automatic Tracing*

Only after trying, we can know if the problem is provable or not. For an unprovable problem, it may take the prover much time to try all possible cases. In some cases, the prover may not terminate proving forever.

When the value of prover is "FAIL" after exhausting every possibility, we would frequently like to know what kind of assumption is needed. So, if our processor fails to prove a problem, it automatically prints the trial process of the proof. This output is printed with the number to show the recursion level only when matching suceeds.

3.3 *Visualizer*

Visualizer is a program which displays an external representation of the process proved by the prover. We pay attention, in converting the internal proof tree into the external proof figure, to the following two, (1) each sequent in a proof figure sould be in the most natural form, (2) proof figure displayed by the visualizer shows explicitly the correspondence to the inference procedure in the proving process. For (1) We display a sequent in the manner just like the problem input, infixed form. For (2) We display the proof figure using the structure of stairs corresponding to the level of sequents in the proof tree.

3.3.1 *Retranslation of a Sequent into an External Representation*

A sequent has the following internal form.

$$(A_1 \quad A_2)$$

where A_1 and A_2 are internal representations of left and right parts respectively. They are retransformed into

$$A_1^* \text{ -> } A_2^*$$

where A_1^* and A_2^* are external representation of A_1 and A_2 respectively. A_i $(i=1,2)$ is a list

$$(B_1 \quad B_2 \quad \ldots \quad B_{n_i}).$$

Here each B_j $(j=1, 2, \ldots, n_i)$ is an expression. If we can translate B_j into B_j^* $(j=1, 2, \ldots, n_i)$, the external representation of A_i, i.e. A_i^* is as follows.

$$B_1^* \ , \ B_2^* \ , \ \ldots \ , \ B_{n_i}^*$$

If A_i is NIL then its external representation will be a null string.

Retranslation method of expression B_j $(j=1, 2, \ldots, n_i)$ is just the same as the translation method of the translator. If an internal expression is an atomic symbol then the

retranslated form is the print name of the atom. If an internal expression has the form

(*op* α β) or (*op* α)

then it is retranslated into the external expression in the form

(α* *op** β*) or (*op** α*)

where *op* is an operator in Table 3.1, α, β, an subexpression, and *op**, α* and β* are external representation of *op*, α and β respectively. Here we must consider of the following. In retranslation a labeled function, we must display the induction level of the function. For example,

(LABEL F (PLUS N 1))

will be retranslated to

F<N+1> .

It is important to erase parentheses if there is no ambiguity in evaluating an expression. So the external representation will be more natural than the fully parenthesized expression.

3.3.2 *Output of a Proof Figure*

Proof genarated by the prover is transmitted to the visualizer through auxiliary memory device as an internal representation. This representation has the structure of a proof tree 3.2.1 (1) and (2).

In order to display a proof tree externally to show the process of its inferences in the proof, we display the proof figure with the structure of stairs. A Gentzen-type proof figure in Fig.3.3 is shown internally as follows

$$(\alpha \ a \ (\beta \ b \ (\gamma \ c \ (\mathrm{VALID}.e)) \ (\mathrm{VALID}.d) \ ))$$

where α, β and γ are indicators for rules of inference. This proof tree will be represented as Fig.3.4

$$\dfrac{\dfrac{\dfrac{e}{c} \qquad d}{b}}{a}$$

FIG. 3.3 An Example of a Proof Figure

Currently, we use a line printer as an output device of the proof figure, and from the restriction of the character set, we use "↑", "*", "←" etc. to write an arc of the proof figure. From the proof tree in Fig.3.1, visualizer draws the proof figure in Fig.3.5.

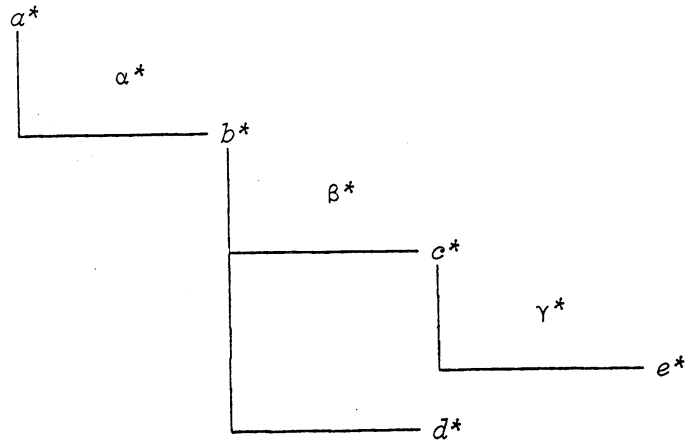


FIG. 3.4 Displayed Form of Fig. 3.3

```
A , A>B -> B
  ↑
  ↑    DEF.OF >
  ↑
  *<<<<< A , \AVB -> B
         ↑
         ↑    LEFT V-ELIM.
         ↑
         *<<<<< A , \A -> B
         ↑       ↑
         ↑       ↑    \-ELIM.
         ↑       ↑
         ↑       *<<<<< A -> A , B
         ↑
         ↑
         ↑
         *<<<<< A , B -> B
```

FIG. 3.5 Proof Figure of A , A>B -> B

This visualizing program is controlled by mainly two functions, *prinsub* and *printree*. *prinsub* controls a list of the proof trees

$$(proof\ tree_1 \quad proof\ tree_2 \quad \ldots \quad proof\ tree_n)$$

It outputs all elements of this list one by one. Output of *prinsub* is just the transmittion of the control to *printree*. *printree* manipulates a proof tree of 3.2.1 (1), (2). *printree* just prints out a sequent in the proof tree with sequent translation, and transmitts the control to *prinsub* with its proof tree list.

In the actual figure drawing, there is a physical limitation with the line printer, the number of characters in a line. In KLISP every line is atmost 100 characters. If we can know that with the structure of stairs the external representation of a sequent is not able to be written in the line, we must control the operation for carriage return and line feed (CRLF). For this, we count the number of characters for tabulation and the external representation of a sequent. If the control is in the second level of the stair, then before CRLF elimination of the blanks, which was inserted for better looking, will be tried. The blank eliminating operation has two levels, elimination of blanks except for the blanks in the both side of "->", elimination of every blanks in the sequent. After the blank eliminating operation if it is not yet enough to print the sequent in the line, or the control is in the deeper level of the stairs, CRLF is tried.

If this CRLF cuts some arcs in the figure, then the connectors of the arcs and the connecting numbers will be added.

Visualizer also prints out the problem itself. First the sequent to be proved, the definitions if there are, assumptions if there are, and then the proof figure if proved. An example will be shown in appendix.

4. *Conclusions*

This processor has been implemented in TOSBAC-3400/30 (24 bits/word 16K words) KLISP interpreter. KLISP interpreter has about 4K cells of free list. The processor has succeeded in proving all examples in [1].

We consider there are some problems about our processor.

(1) Currently we use only static assumptions. We have to extend the processor to accept a kind of dynamic assumptions, say, procedural assumptions.

(2) The processor should have some well known and useful algebraic theorems, and theorems never to be used should be automatically deleted from the system. Theorems prepared should be selected at users' will.

(3) We consider that our theorem prover will be more powerful if the conversational facility is added. We hope to report an implementation of a conversational theorem prover.

REFERENCES

[1] Manna, Z., Ness, S., and Vuillemin, J. Inductive methods for proving properties of programs. *Proc. of an ACM Conference on Proving Assertions about Programs, New Mexico State Univ., New Mexico, January 6-7, 1972, 27-50*

[2] Nishimura,T. Gentzen-type formal system representing properties of functions. *This Publication of the Research Institute for Mathematical Sciences*

Appendix. A proof figure of the associativity of *append*

Problem append[append[x;y];z] -> append[x;append[y;z]]

Definition of *append*

```
append[x;y] = [null[x]->y;
               t->cons[car[x];append[cdr[x];y]] ] ;
```

Assumptions

1. Inductive assumption
2. A property of *cons*

A proof figure of this problem is shown in the next page.

PROVE G(?F(X,Y),Z) -> ?F(X,G(Y,Z)) Associativity of Append

WITH DEFINITION

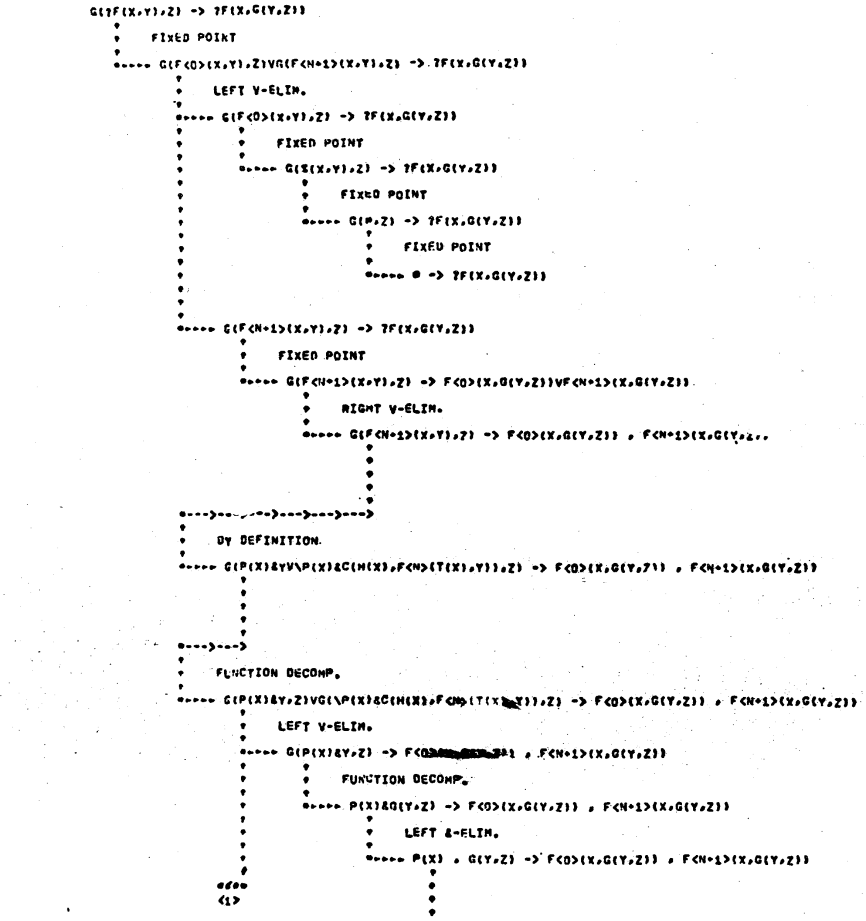
1. F<N+1>(X,Y) = P(X)&∀V\P(X)&C(H(X),F<N>(T(X),Y))

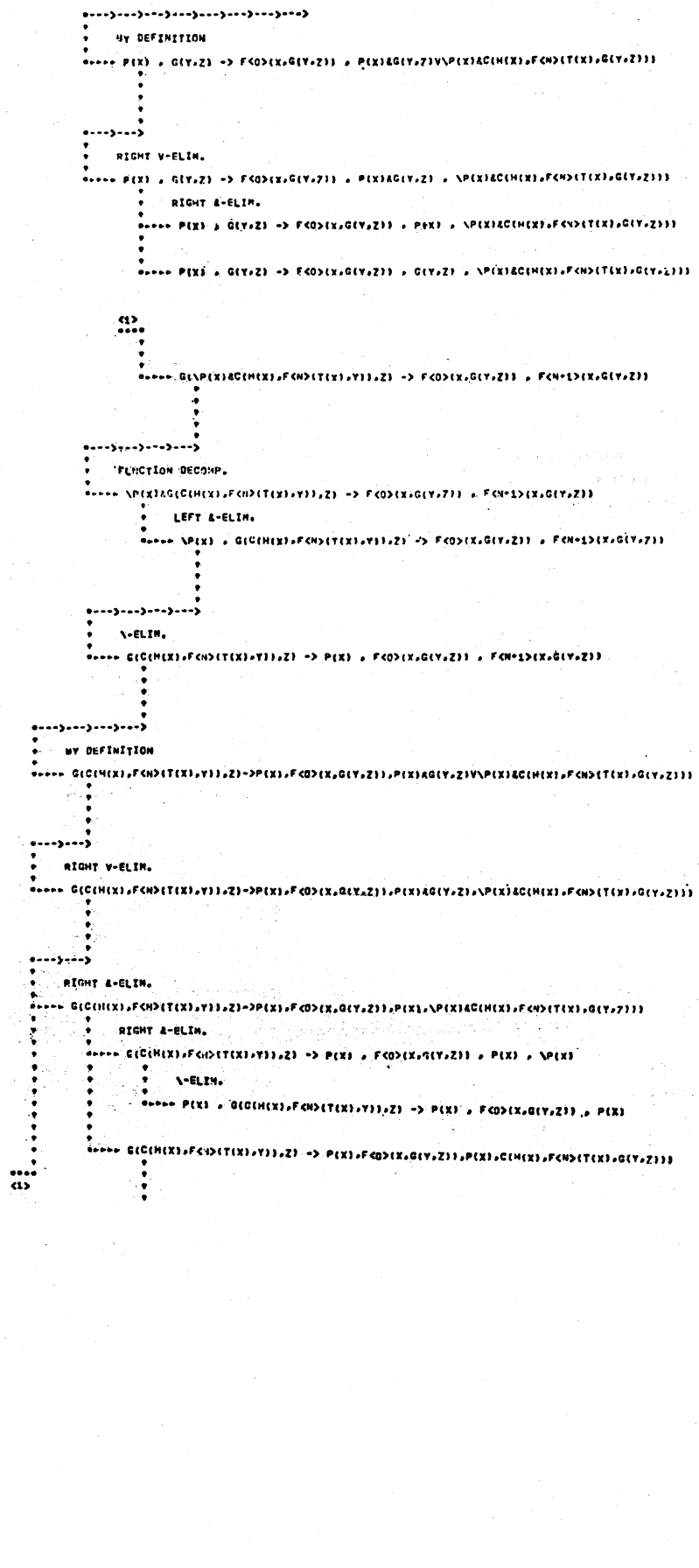
WITH ASSUMPTIONS

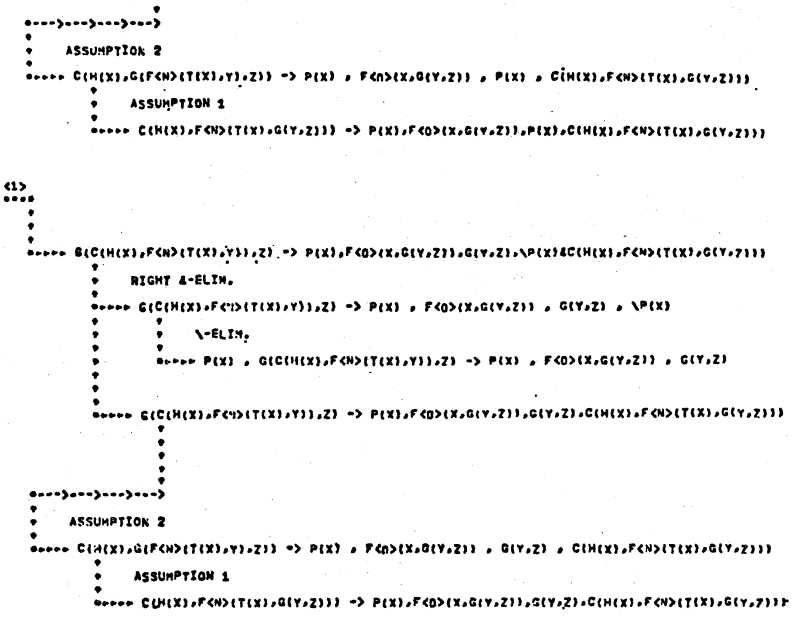
1. G(F<0>(∀X,∀Y),∀Z) -> F<N>(∀X,G(∀Y,∀Z))

2. G(C(H(∀X),∀Y),∀Z) -> C(H(∀X),G(∀Y,∀Z))

PROOF FIGURE




BY DEFINITION
RIGHT V-ELIM.
RIGHT &-ELIM.
FUNCTION DECOMP.
LEFT &-ELIM.
\-ELIM.
BY DEFINITION
RIGHT V-ELIM.
RIGHT &-ELIM.
RIGHT &-ELIM.
\-ELIM.



THIS COMPLETES THE PROOF.