数理解析研究所講究録
396巻 1980年 131-169

Interactive Debugging for Functional Recursive Programming

by

Morio Nagata

Dept. of Administration Engineering

Faculty of Engineering, Keio University

3-14-1 Hiyoshi, Yokohama 223, JAPAN

ABSTRACT

This paper presents a method for constructing reliable recursive programs written in a functional style and an interactive support system based on that method. This approach accepts the specifications of data types and their properties. Whenever a fragment of a recursive program is given, its properties of logical relations and data types are investigated. Using automatic theorem proving techniques, this method indicates conditions which are necessary for the intended program. The method is also concerned with the termination of functional recursive programs.

1. INTRODUCTION

From theoretical and practical points of view, the functional programming style and recursive programming techniques are useful concepts in computer science. McCarthy's Lisp [McC62], Landin's Iswim [LAN66] and Backus's FP system [BAC78] are examples of functional programming languages; above all, Lisp is widely implemented throughout the world. In recent years, in order to overcome the difficulties of program writing, there have been many arguments on programming methodology, and as a result of them, the functional style of programming has come to be recognized as one of the most promising methodologies. This paper gives the first step of our theoretical and practical studies on debugging functional recursive programs.

We do not aim to present a new programming languages. Our approach, called KANSUU (Keio AdvaNced approach for SUpporting fUnctional programming), presents notations representing programs in functional styles. If a programmer writes his programs in the notations as specified in KANSUU, logical errors in the programs can be detected automatically. The termination of certain types of programs can also be confirmed automatically. KANSUU assumes that a program consists of functions which have been already defined, control structures of recursion, McCarthy's conditional expression (or if-then-else construct) and functional composition.

In functional programming, to write programs is defining functions by using the above components. KANSUU is not bound to a particular programming language. In KANSUU, a particular programming language can be specified by data types and primitive functions which have been built in the language. The programmer can write reliable programs in the Lisp

language or other functional languages by using this approach. An interactive support system based on KANSUU, called KSR (Keio Support system for functional Recursive programming), has been implemented in the Lisp language on minicomputers PDP 11/21 and PDP 11/34 which are installed in our department [NAG80b].

The main features of KANSUU are listed below;

(1) Using automatic theorem proving techniques, KANSUU detects not only the exsitence of errors but also provides information for the correction of inconsistent programs.

(2) It is not necessary that the user understand the specific terminologies and techniques of mathematical logics on which KANSUU is founded.

(3) KANSUU does not require that any assertion added to programs.

(4) KANSUU can be applied to both top-down and bottom-up programming.

(5) An automatic mechanism confirming the termination of functional recursive programs is embedded in KANSUU.

(6) KANSUU is useful for the implementation of an interactive programming system.

2. DETECTION OF LOGICAL ERRORS

we show an automatic method for detecting logical errors. The correctness of this method is verified by propositional calculus [NAG80a].

2.1 Syntax of Our Approach

We shall explain some notations for illustrative examples of KANSUU,

and give the syntax of an F-program and other notations in KANSUU.

2.1.1 The Notations for Illustrative Examples

When writing programs in a functional style, we use two basic components, primitive functions and data types, of the programming language. In KANSUU, we have to declare those components before entering the program.

Showing list processing programs as examples, we introduce the basic notions of list structures at first. The notions correspond to those of the Lisp language in a straightforward way. However they are useful for the presentation of some ideas behind our work without reference to a particular programming language.

There are two basic kinds of data, atoms and lists. An atom is a string of alpha-numeric characters which should be taken as a whole and should not be split into individual characters. A number is an atom, and as a first step, we consider only integers as numbers. Excepting integer, the first character of the atom must be alphabetic characters. A, ATOM and 1980 are examples of atoms.

The fundamental structure of a list is a b-list which is defined as follows (cf. [ALL78]):

1. An atom or ε is a b-list.
2. If b_1 and b_2 are b-lists, then $(b_1 : b_2)$ is a b-list.
3. The only b-lists are those given by 1 and 2.

Here ε, the null list, may be written as (), and regarded as a special atom. A proper b-list is a b-list of the form

$(b_1:b_2)$,

where b_1 and b_2 are b-lists. b_1 and b_2 in the definition of the proper b-list $(b_1:b_2)$ are called the first element and the second element of the proper b-list respectively.

The other representation of a list, a k-list, is defined as:

1. ε is a k-list of length 0.

2. If x is a k-list and y is a b-list, then (y:x) is a k-list of length $|x|+1$, where $|x|$ is the length of x.

3. The only k-lists are those given by 1 and 2.

$(x_1:(x_2:(\ldots:(x_n:())\ldots)))$ will often be abbreviated as $(x_1,x_2,\ldots,x_n)$. We sometimes use a list as a b-list.

Now, in general there are data types and primitive functions which have already been provided in each functional programming language. We shall describe such primitive functions which are assumed in later discussions.

There are three primitive functions which process lists as follows.

1) hd(x) selects the first element of a b-list x.

2) tl(x) selects the second element of a b-list x.

3) cons(x,y) constructs a b-list whose first element is x and second is y. The value is (x:y)

The following Boolean valued functions are added to primitive functions.

4) If x is an atom, then atom(x) is true, else the value is false.

5) x and y are atoms. If x equals y then eq(x,y) is true, else the value is false.

When x is a proper b-list, cons(hd(x),tl(x)) is x. In addition to 1)-5), Boolean valued functions $\vee$(or), $\wedge$(and), $\neg$(not) are used.

At first we must declare data types provided by the language. For example, it is assumed that

Boolean, integer, string

are declared. We assume that Boolean values, integers and character strings are atoms. By the difinition of a b-list, an atom is an element of the set of b-list and the atom may be an element constructing the b-list. This hierarchical relationship between atom and list is represented as 'atom$\prec$list' in KANSUU.

Primitive functions on the data should be listed next. We may add conditions which we assume to hold when evaluating the function. For example, we must assume that hd and tl operate only on lists that are not atoms. The value is regarded as undefined when hd is operated on an atom. In this case, $\neg$atom is one of the conditions which should hold for the arguments of hd. We call such a condition a guard predicate of the function (cf. Dijkstra's guarded command [DIJ76]).

2.1.2 Syntax of F-program and Specification

In this section, all symbols, i.e. constants, variables, function and predicate symbols are specified by a particular programming language and the user's program.

We shall list the syntax of the conditional in BNF.

<term> ::= <variable>|<constant>|<function symbol>(<arglist>)|
<logical term>

<arg> ::= <term>|<proposition>

<arglist> ::= <arg>|<arglist>,<arg>

<logical term> ::= <logical variable>|<Boolean constant>|
<predicate symbol>(<arglist>)

<expression> ::= <term>|<the conditional>

<prime proposition> ::= <logical term>|(<proposition>)

<factor proposition> ::= <prime proposition>|¬<prime proposition>

<term proposition> ::= <factor proposition>|
<term proposition>∧<factor proposition>

<proposition> ::= <term proposition>|
<proposition>∨<term proposition>

<fragment> ::= ⟨<expression> -> <expression>⟩

<fragment sequence> ::= <fragment>;<fragment>|
<fragment sequence>;<fragment>

<the conditional> ::= ⟦<fragment sequence>⟧

The left-hand side of '->' of a fragment is called the condition part, the right-hand side is called the expression part of the fragment. The condition part is usually a proposition. A fragment sometimes has its value. If the expression part is a proposition too, the fragment < C -> E > is called a propositional fragment and the value is ¬C∨E.

The value of the conditional in a valuation is the value of the expression part of a fragment whose condition part is true. If an expression, i.e. the conditional or term, is given, the level of the

expression on this expression is defined as 0. That is, the level of given expression on the expression itself is 0. If the conditional of level n on an expression has the form

$$[\![< C_1 \text{ -> } E_1 >;\ldots;< C_m \text{ -> } E_m >]\!],$$

then each expression of $C_1,\ldots,C_m,E_1,\ldots,E_m$ is of level n+1.

When the conditional is given and there exists an expression whose level is more than 1, we say that it has a nesting structure. McCarthy's conditional expression,

$$[C_1 \text{ -> } E_1;\ldots;C_k \text{ -> } E_k] ,$$

is equivalent to the conditional,

$$\begin{aligned}[\![& < C_1 \text{ -> } E_1 >; \\ & < \neg C_1 \wedge C_2 \text{ -> } E_2 >; \\ & \quad\vdots \\ & < \neg C_1 \wedge \neg C_2 \wedge \ldots \wedge \neg C_{k-1} \wedge C_k \text{ -> } E_k >]\!].\end{aligned}$$

Now we shall list the syntax of an F-program.

<guard term> ::= <predicate symbol>(<arg list>)

<guard primary> ::= <guard term>|(<guard predicate>)

<guard factor> ::= <guard primary>|¬<guard primary>

<guard term predicate> ::= <guard factor>|

<guard term predicate>∧<guard factor>

<guard predicate> ::= <guard term predicate>|

<guard predicate>∨<guard term predicate>

<guard predicate list> ::= <guard predicate>|

<guard predicate list>,<guard predicate>

<spec arg> ::= <variable>.<data type>

<spec arg list> ::= <spec arg>|<spec arg list>,<spec arg>

<specification part> ::= <function symbol>(<spec arg list>) =>
<data type>;|

<function symbol>(<spec arg list>) =>
<data type>;<guard predicate list>

<specification> ::= {<specification part>}

<F-program> ::= {<specification part> <= <the conditional>}

The F-program representing a function, and its definition is represented in the conditional. The specification specifies the domain and range of the function. Examples of the specifications of our discussion are as follows:

Data types: {((Boolean,integer,string) = atom) < list }

Primitive functions: {hd(x.list) => list; ¬atom(x)},
{tl(x.list) => list; ¬atom(x)},
{cons(x.list,y.list) => list },
{ atom(x.list) => Boolean},
{eq(x.atom,y.atom) => Boolean }

An example of the F-program is as follows:

```
{equal(x.list,y.list) => Boolean;
  <= [[ <atom(x)/\atom(y) -> eq(x,y)>;
        <atom(x)/\¬atom(y) -> false>;
        <¬atom(x)/\atom(y) -> false>;                    (2.1)
        <¬atom(x)/\¬atom(y)
             -> equal(hd(x),hd(y))/\equal(tl(x),tl(y))> ]] }
```

We introduce the rule for the use of symbols in later sections. Boolean constant is true or false,. Data types are represented by strings beginning with d. Constants are represented by alpha-numeric strings beginning with a, b or c. Variables are strings beginning with x, y or z, and logical variables are represented by strings beginning with s, t or u. Function symbols are strings beginning with f, g or h, and predicate symbols are strings beginning with p, q or r.

2.2 Algorithms for Detecting Logical Errors

In our discussion, the following properties of fragments in the conditional should be verified.

(1) Each condition part is not identical to true or false.

(2) Disjunction of all condition parts is always true. (Exhaustiveness)

(3) Conjunction of condition parts of distinct fragments are always false. (Exclusiveness)

Algorithms for detecting logical errors of F-programs, which are based on these properties, will be described

2.2.1 Automatic Theorem Proving and Trivial Fragment

We shall use a formal system which is a subset of Gentzen's LK [GEN34]. The formal system is useful for describing above properties and verifying our algorithms. In this paper, we use the notations of the formal system which are described by Kleene [KLE52].

In KANSUU, a sequent is used as an internal representation for

detectiing logical errors. The sequent is the same as Gentzen's sequent, that is, a sequent is a formal expression of the form

$$A_1,\ldots,A_\ell \rightarrow B_1,\ldots,B_m$$

where $l,m \geq 0$ and $A_1,\ldots,A_\ell,B_1,\ldots,B_m$ are propositions. The part $A_1,\ldots,A_\ell$ is the antecedent, and $B_1,\ldots,B_m$ the succedent of the sequent. When $l,m \geq 1$, the sequent

$$A_1,\ldots,A_\ell \rightarrow B_1,\ldots,B_m$$

has the same interpretation as

$$A_1 \wedge \ldots \wedge A_\ell \text{ implies } B_1 \vee \ldots \vee B_m.$$

The interpretation extends to the cases where $l=0$ or $m=0$ by interpreting $A_1 \wedge \ldots \wedge A_\ell$ for $l=0$ (the 'empty conjunction') as true and $B_1 \vee \ldots \vee B_m$ for $m=0$ (the 'empty disjunction') as false. Note that the sequent is utilized only when KANSUU detects logical errors and confirms the termination of F-programs, and '→' of the sequent is different from '->' of the fragment.

Logical axioms of KANSUU are:

$$\Gamma_1,A,\Gamma_2 \rightarrow \Delta_1,A,\Delta_2,$$

$$\Gamma \rightarrow \Delta_1,\text{true},\Delta_2$$

and

$$\Gamma_1,\text{false},\Gamma_2 \rightarrow \Delta,$$

where A is any proposition, and Greek capitals represent zero or more propositions.

Rules of inference of KANSUU, which are included in LK, are as

follows:

Rules of Inference:

Let P and Q be arbitrary propositions, and Greek capitals be zero or more propositions, then logical rules of inference are the following.

$$(\text{left-}\wedge) \qquad \frac{\Gamma_1, P, Q, \Gamma_2 \to \Delta}{\Gamma_1, P \wedge Q, \Gamma_2 \to \Delta}$$

$$(\text{right-}\wedge) \qquad \frac{\Gamma \to \Delta_1, P, \Delta_2 \quad \Gamma \to \Delta_1, Q, \Delta_2}{\Gamma \to \Delta_1, P \wedge Q, \Delta_2}$$

$$(\text{left-}\vee) \qquad \frac{\Gamma_1, P, \Gamma_2 \to \Delta \quad \Gamma_1, Q, \Gamma_2 \to \Delta}{\Gamma_1, P \vee Q, \Gamma_2 \to \Delta}$$

$$(\text{right-}\vee) \qquad \frac{\Gamma \to \Delta_1, P, Q, \Delta_2}{\Gamma \to \Delta_1, P \vee Q, \Delta_2}$$

$$(\text{left-}\neg) \qquad \frac{\Gamma_1, \Gamma_2 \to P, \Delta}{\Gamma_1, \neg P, \Gamma_2 \to \Delta}$$

$$(\text{right-}\neg) \qquad \frac{P, \Gamma \to \Delta_1, \Delta_2}{\Gamma \to \Delta_1, \neg P, \Delta_2}$$

A sequent is provable if it is an axiom or the result of applying a rule of inference to sequents which are already known to be provable. From the computer science point of view, a provable sequent means a statement

which is capable of being proved by an automatic theorem prover which works in accordance with the formal system.

If a sequent to be proved is given, the above procedure is applied in KANSUU. When a complete proof tree can be constructed, the sequent is a provable sequent. Every node of the tree is a sequent. A node is transformed into its son or sons by applying one of rules of inference which is relevant. Every terminal node of the complete proof tree satisfies the logical axiom. When the sequent is not a provable sequent, an incomplete proof tree is constructed. In this tree, there exist terminal nodes which are not provable sequents and have no logical connectives. We call these nodes non-provable terminal nodes of the tree.

Concerning property (1), we define a trivial fragment as follows.

Definition 2.1: Trivial Fragment
When a condition part of a fragment has a value which is either always true or always false, the fragment is called a trivial fragment.

The following algorithm tests whether a fragment, < C -> E >, is a trivial fragment.

Algorithm 2.1:
Try to prove two sequents

C → and → C,

where 'C → ' and ' → C' are equivalent to 'C → false' and 'true → C' respectively. If one of them is a provable sequent, then return "It is a trivial fragment.", else return "It is not a trivial fragment!".

144

Example 2.1:

A fragment

$$< \text{atom}(x) \wedge \neg \text{atom}(x) \text{ -> } \underline{\text{false}} >$$

is a trivial fragment.

2.2.2 Exhaustiveness and Exclusiveness

Property (2) and (3) will be shown by the notions of propositional calculus in this section. We shall describe algorithms for detecting logical errors by using these properties.

Definition 2.2: Exhaustiveness

The condition parts of the conditional $[\![<C_1 \text{ -> } E_1>;\ldots;<C_n \text{ -> } E_n>]\!]$ $(n \geq 2)$ are exhaustive, iff

$$C_1 \vee C_2 \vee \ldots \vee C_n \quad \text{is always } \underline{\text{true}}.$$

Definition 2.3: Exclusiveness

The condition parts of the conditional are exclusive, iff for every $i \neq j$,

$$C_i \wedge C_j \text{ are always } \underline{\text{false}},$$

If the conditional

$$[\![< C_1 \text{ -> } E_1 >;\ldots;< C_n \text{ -> } E_n >]\!]$$

is given, the following algorithm tests whether the condition parts are exclusive.

Algorithm 2.2:

Try to prove sequents

$$C_i \wedge C_j \rightarrow$$

for every $i \neq j, 1 \leq i, j \leq n$. If all sequents are provable, then return "exclusive!", else return "C_i and C_j are not exclusive!".

On the other hand, the following algorithm tests whether the condition parts are exhaustive.

Algorithm 2.3:

Try to prove a sequent

$$\rightarrow C_1 \vee C_2 \vee \ldots \vee C_n.$$

If it is a provable sequent, return "exhaustive!", else apply Algorithm 2.2.

If the condition parts are found exclusive but not exhaustive, the following algorithm suggests a proposition which may be lacking.

Algorithm 2.4:

If a sequent

$$A_1, A_2, \ldots, A_k \rightarrow A_{k+1}, \ldots, A_n$$

is the terminal node of the incomplete proof tree of

$$\rightarrow C_1 \vee C_2 \vee \ldots \vee C_n,$$

and does not satisfy the logical axiom, then propose the following proposition as the missing condition part.

$$A_1 \wedge \ldots \wedge A_k \wedge \neg A_{k+1} \wedge \neg A_{k+2} \wedge \ldots \wedge \neg A_n,$$

where A_i is a logical term.

Example 2.2:

Consider the following condition parts of three fragments.

C_1 : atom(x)$\wedge$atom(y)

C_2 : atom(x)$\wedge\neg$atom(y)

C_3 : $\neg$atom(x)$\wedge\neg$atom(y)

In this case,

$$\rightarrow C_1 \vee C_2 \vee C_3$$

is not a provable sequent, and the sequent

$$\text{atom(y)} \rightarrow \text{atom(x)},$$

is not provable in the proof. We can find that a fragment whose condition is

$$\neg\text{atom(x)}\wedge\text{atom(y)}$$

should be added.

2.3 Verification of Algorithms

In this section, we shall verify the algorithms which have been shown in 2.2. The notions, i.e. trivial fragments, exclusiveness and exhaustiveness have been defined by using true or false, while the algorithms, have been presented by using 'provable sequents'. Thus it is necessary to verify that the algorithms based on automatic theorem proving

techniques assure property (1), (2) and (3). Especially, algorithm 2.4 is not so trivial that it is reasonable to demand to verify that the proposed proposition is an adequate one.

2.3.1 Verification of Algorithms for Detecting Errors

Algorithms 2.1, 2.2 and 2.3 will be verified. These algorithms seem to be trivial, but it seems justified to show explicitly that each notion of the fragment or the conditional can be assured by each algorithm which is executed on a computer.

In order to verify them, we use 'validity' of logics. The notion is defined as follows. If a sequent

$$\Gamma \rightarrow \Delta$$

is given, and

$$\text{truth value of } \Gamma \leq \text{truth value of } \Delta$$

always holds, then the sequent is called a valid sequent, where false≤false, false≤true and true≤true.

It is well known that Gentzen's LK is complete, i.e.

every valid sequent is provable.

Since our formal system is a subset of LK, its completeness can be easily shown in the same way as the proof of the completeness of LK.

Now, it is known that LK is plausible, i.e.,

every provable sequent is valid.

Our formal system is plausible.

By the completeness, plausibility and the definition of the trivial fragment, the following theorem can be easily shown.

Theorem 2.1:

C is a condition part of a trivial fragment, iff

$C \rightarrow$ or $\rightarrow C$

is provable.

The meaning of the theorem is described as follows. If a given fragment is a trivial fragment, then KANSUU always detects that it is a trivial fragment. Conversely, if KANSUU detects that a fragment is a trivial fragment, then it is certainly a trivial fragment.

By the completeness, plausibility and the definition of exclusiveness, we can easily obtain the following theorem which assures Algorithm 2.2.

Theorem 2.2:

Let $[\![<C_1 \text{ -> } E_1>;\ldots;<C_n \text{ -> } E_n>]\!]$ $(n\geq 2)$ be the conditional. If the condition parts of the conditional are exclusive, then for every $i\neq j$,

$$C_i \wedge C_j \rightarrow$$

are provable sequents, where $1\leq i,j\leq n$, and vice versa.

This theorem assures that Algorithm 2.2 can detect fragments whose condition parts are not exclusive.

By the definition of exhaustiveness, we can obtain the following theorem.

Theorem 2.3:

Let $[\![<C_1 \rightarrow E_1>;\ldots;<C_n \rightarrow E_n>]\!]$ $(n \geq 2)$ be the conditional. If those condition parts are exhaustive, then

$$\rightarrow C_1 \vee C_2 \vee \ldots \vee C_n$$

is a provable sequent, and vice versa.

Theorem 2.2 and Theorem 2.3 show that we have an automatic method to decide whether the condition parts of the conditional are exhaustive and exclusive.

2.3.2 Verification of Algorithm 2.4

Algorithm 2.4 has shown that an automatic method can provide necessary information to make the conditional exhaustive when those condition parts are exclusive but not exhaustive. The algorithm will be verified by the following theorem.

Theorem 2.4:

Let $[\![<C_1 \rightarrow E_1>;\ldots;<C_n \rightarrow E_n>]\!]$ $(n \geq 2)$ be the conditional, and each C_i be the conjunction whose elements are logical terms and negations of logical terms. If those condition parts are exclusive but not exhaustive, and, in the proof of

$$\rightarrow C_1 \vee C_2 \vee \ldots \vee C_n,$$

there exists a terminal node

150

$$A_1, A_2, \ldots, A_k \rightarrow A_{k+1}, \ldots, A_n$$

which is not a provable sequent(i.e. a non-provable terminal node), then, by adding a fragment whose condition part is

$$C_{n+1} : A_1 \wedge A_2 \wedge \ldots \wedge A_k \wedge \neg A_{k+1} \wedge \ldots \wedge \neg A_n ,$$

the condition parts can be made exhaustive and exclusive.

This theorem can be extended as follows.

Corollary:

Let $[\![<C_1 \rightarrow E_1>; \ldots; <C_n \rightarrow E_n>]\!]$ $(n \geq 2)$ be the conditional and condition parts of all fragments of the conditional be exclusive. If, in the proof of

$$\rightarrow C_1 \vee C_2 \vee \ldots \vee C_n ,$$

$$\begin{array}{l} A_{1\,1}, \ldots, A_{1\,k_1} \rightarrow A_{1\,k_1+1}, \ldots, A_{1\,n} \\ A_{2\,1}, \ldots, A_{2\,k_2} \rightarrow A_{2\,k_2+1}, \ldots, A_{2\,n} \\ \qquad\qquad \vdots \\ A_{m\,1}, \ldots, A_{m\,k_m} \rightarrow A_{m\,k_m+1}, \ldots, A_{m\,n} \end{array}$$

are non-provable terminal nodes of the incomplete proof tree, then, by adding fragments whose condition parts are

$$\begin{array}{l} C_{n+1} : A_{1\,1} \wedge \ldots \wedge A_{1\,k_1} \wedge \neg A_{1\,k_1+1} \wedge \ldots \wedge \neg A_{1\,n} \\ C_{n+2} : A_{2\,1} \wedge \ldots \wedge A_{2\,k_2} \wedge \neg A_{2\,k_2+1} \wedge \ldots \wedge \neg A_{2\,n} \\ \qquad\qquad \vdots \\ C_{n+m} : A_{m\,1} \wedge \ldots \wedge A_{m\,k_m} \wedge \neg A_{m\,k_m+1} \wedge \ldots \wedge \neg A_{m\,n} \end{array}$$

respectively, where '$C_i \wedge C_j \rightarrow$ ' are provable for every $i \neq j$, $n < i, j \leq n+m$, the condition parts can be made exhaustive and exclusive.

In this corollary, if there exist fragments such that

$$C_i \wedge C_j \rightarrow$$

are not provable for $i \neq j$, where $n<i,j \leq n+m$, then we can make the exhaustive condition parts of the conditional by selecting a set X such that

$$C_i \wedge C_j \rightarrow$$

are provable for any $C_i \in X$ and $C_j \in X$, where $i \neq j$, $n<i,j \leq n+m$.

Example 2.3:

Let

$$C_1 : \text{atom}(x) \wedge \text{atom}(y) \qquad \text{and} \qquad C_2 : \text{atom}(x) \wedge \neg\text{atom}(y)$$

be condition parts of a given fragment. In this case, in the proof of

$$\rightarrow C_1 \vee C_2,$$

the following sequents are non-provable terminal nodes.

$$\rightarrow \text{atom}(x)$$

$$\text{atom}(y) \rightarrow \text{atom}(x)$$

$$\rightarrow \text{atom}(y), \text{atom}(x)$$

Therefore we have two ways of making the condition parts exclusive and exhaustive. First, C_1, C_2, and $\neg$atom(x) are exclusive and exhaustive; second, C_1, C_2, $\neg\text{atom}(x) \wedge \text{atom}(y)$ and $\neg\text{atom}(x) \wedge \neg\text{atom}(y)$ are exclusive and exhaustive.

If an F-program satisfies all of property (1), (2) and (3), it is called a consistent F-program.

3. TEST OF THE TERMINATION OF F-PROGRAMS

We shall show the way to confirm the termination of certain types of F-programs. The general problems of the termination and efficiency of recursive programs would be too difficult to be implemented on a computer, if possible at all. Therefore our method is restricted to consistent F-programs of the particular forms.

3.1 Termination of Constructed Programs

This section will specify the types of recursive programs whose termination can be confirmed in KANSUU. The basic idea of the algorithm confirming the termination of programs will also be described. Algorithms for verifying the termination in KANSUU enable us to implement the interactive system confirming the termination of some restricted types of recursive programs.

Let us assume that an F-program

```
{ fib(x.non-negative integer) => non-negative integer;
        <= [[ <x=0 -> 1>;                                    (3.1)
             <¬(x=0) -> fib(x-1)+fib(x-2)> ]] }
```

is given by a programmer. This is a consistent F-program. However we can find that the execution of the program can not terminate for $x \geq 1$. In order to detect such errors, we give a semi-automatic approach confirming the termination of consistent F-programs.

Our program is regarded as the definition of a function, and, roughly speaking, it terminates if the function is total on a set, which is given by the data types of variables of the F-program. In order to introduce our

method, let us consider one of the simplest cases as an example. A recursive F-program h with the variable x is defined by:

```
{h(x.type) => type' ;
    <= [[ <p(x) -> e(x)>;                                    (3.2)
          <¬p(x) -> a(h(b(x)))> ]] }
```

where e, a and b are primitive, specified or defined functions. This program can be interpreted as

```
{ h^{n+1}(x.type) => type' ;
      <= [[ <p(x) -> e(x)>;
            <¬p(x) -> a(h^n(b(x)))> ]]},
```

where h^0 is a totally undefined function. Now,

$$Dom(h^k)$$

represents the set such that h^k is defined, i.e.,

$$Dom(h^0)$$

is ϕ, and in this example,

$$Dom(h^1)$$

is the set $\{x|p(x)\}$.

Further define

$$D$$

as the set given by the data type of the variable.

The programmer expects that the program terminates if any element of

the set D is given as a datum, so we call D an expected set. For example, the expected set of fib is the set of non-negative integers. When a program with two or more variables is given, D is the Cartesian product of expected sets which are given by data types.

Then, we conclude that, if

$$D=Dom(h^{\infty}) = Dom(h^{1})\cup Dom(h^{2})\cup\ldots\cup Dom(h^{n})\cup\ldots$$

holds, then the F-program terminates. By the definition of Dom, $Dom(h^{k})$ increases (or does not decrese) throughout the computation, and k may be regarded as the index of the process of the computation.

3.2 Bottom Predicate and Control Variables

In order to show the termination of a given F-program, we have to consider the structure of the expected set D. We define that $(D,\leq)$ as well-founded set which consists of a set of elements of D and where ordering $\leq$ is defined on the elements. KANSUU uses the structure of D and does not need the termination function. For proving the termination of the F-program, we directly use the well-founded set $(D,\leq)$ and the text of the program.

By the definition of the well-founded set, there exist minimum elements in the set. If, in (3.2), p is true on every minimum element of $(D,\leq)$ and false on all other elements, then termination of h is obvious. The predicate p is called a bottom predicate. This is determined by the operation and the relation over the domain.

Definition 3.2: Bottom Predicate

Let $<D,\leq>$ be a well-founded set. A predicate p on D is the bottom

predicate of an operation g iff

For any element d of D excepting all minimum elements, g(d) immediately precedes d, and P is true for all minimum elements and it is false on all other elements of D.

Example 3.1:

If d consists of all non-negetive integers, "<" is 'less than', then $\lambda x.x=0$ is the bottom predicate of $\lambda x.x-1$ (Fig. 3.1).

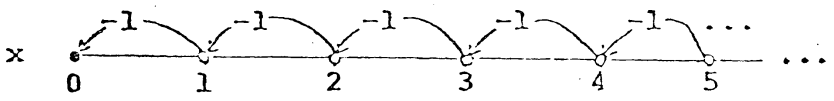

Fig. 3.1 A bottom predicate $\lambda x.x=0$

Example 3.2:

If l is a set of all lists, then atom is the bottom predicate of hd. Fig. 3.2 demonstrates that hd(b) immediately preceeds b for any proper b-list b, here each α_i represents an atom (cf. [SUM77]).

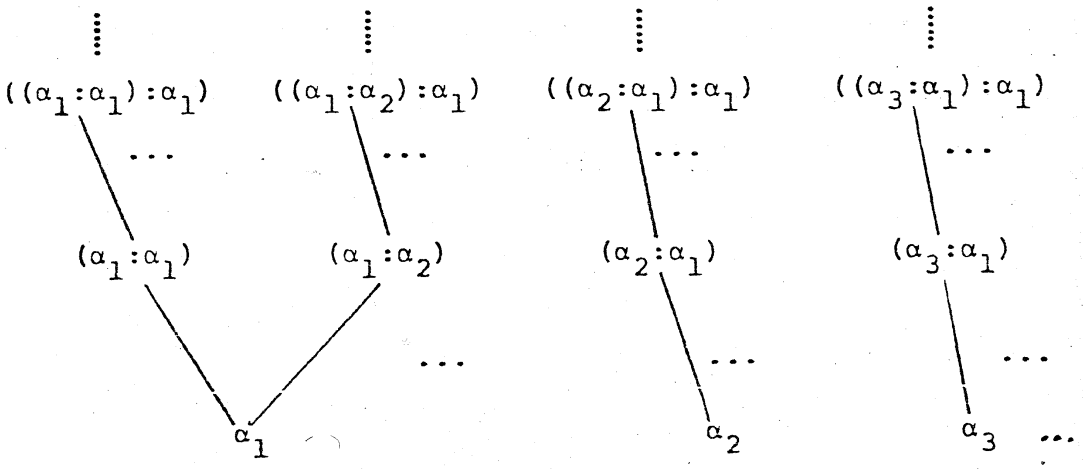

Fig. 3.2 Structure of list

Example 3.3:

If l is a set of all k-lists, then null is the bottom predicate of tl (Fig. 3.3), where null(x) is true if x is ε, and false otherwise. Here represents an arbitrary atom.

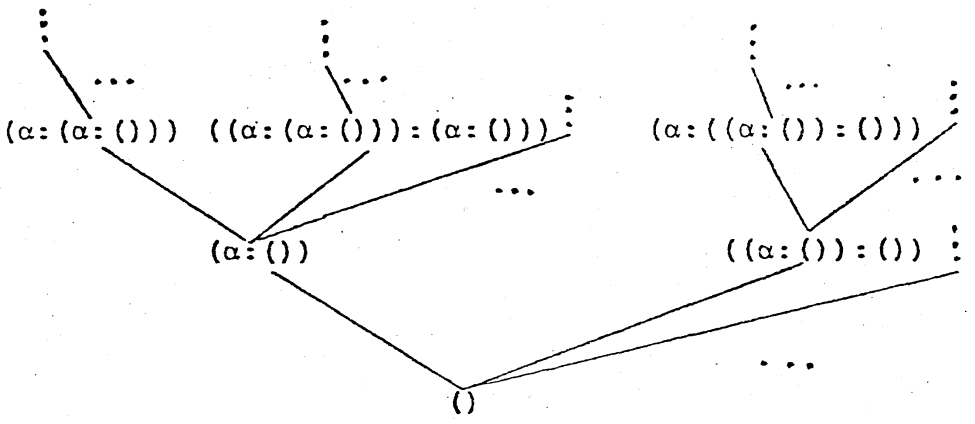

Fig. 3.3 Null and tl for k-lists

Since an F-program may have two or more variables, we introduce the following notions of control variables.

Definition 3.2: Control Variables

Let f be an F-program with n ($n \geq 2$) variables $x_1, x_2, \ldots, x_n$. The control variables of f are those variables which appear in the condition parts of fragments of f.

In order to demonstrate the role of this notion, we show the following example. Both of these programs, frev1 and frev2, reverse k-lists. The program frev2 has two variables x and y, and the control variable is only x. The termination of these programs will be confirmed by the same way (see Algorithm 3.1).

Example 3.4:

Consider the following two programs, frev1 and frev2, both reversing k-lists.

```
{frev1(x.k-list) => k-list;
    <= [[<null(x) -> ε>;
        <¬null(x) -> append(frev1(t1(x)),cons(hd(x),ε)))>]]}
```

where append appends two lists.

```
{frev2(x.k-list,y.k-list) => k-list;
    <= [[<null(x) -> y>;
        <¬null(x) -> frev2(t1(x),append(y,cons(hd(x),ε))))>]]}
```

where frev2(x,ε) returns the reverse of x.

3.3 Test of the Termination

We shall describe the algorithm testing the termination of F-programs. When writing recursive programs, we consider the relation of the program and its expected sets. The execution of a recursive program depends on the structure of expected set given by data types of its variables, therefore the termination of the progaram should be confirmed on the basis of properties of data types. In the following descriptions, algorithm 3.1 will assure the termination of F-programs with one control variable, while Algorithm 3.2 will assure the termination of F-programs with two or more control variables.

Let an exclusive and exhaustive F-program f with a control variable x be given. Let g(x) be the parameter of the recursive call of f, and $q_1(x),\ldots,q_n(x)$ be condition parts of non-recursive fragments of f. We assume that the expected set D and the bottom predicate p on D of the function g are also given.

Algorithm 3.1:

Prove a sequent

$$p(x) \rightarrow q_1(x) \vee \ldots \vee q_n(x).$$

If it is a provable sequent, then the termination is confirmed, else is not confirmed.

The termination of two F-programs, frev1 and frev2, can be be confirmed by this algorithm.

An F-program may include composition of functions as the parameter of the recursive call, and in such a case, the bottom predicate of the

composition can be obtained as follows. Let $p_1,\ldots,p_n$ be bottom predicates of $g_1,\ldots,g_n$ respectively and let the argument of the recursive call be

$$g_n(g_{n-1}(\ldots(g_1(x))\ldots)),$$

then the bottom predicate of the composition function is

$$p_1(x)\vee p_2(g_1(x))\vee\ldots\vee p_n(g_{n-1}(\ldots(g_1(x))\ldots)).$$

Thus, suppose an F-program (3.1) is given, we find that

$$(x=0)\vee((x-1)=0)\rightarrow(x=0)$$

is not a provable sequent. The system will detect that a non-recursive fragment whose condition part is (x-1)=0 is missing.

Therefore the programmer is let to the following corrected F-program by a conversation with the system.

```
{fib(x.integer) => integer; ¬(x<0)
    <=[ < x=0 -> 0 >;
        < ¬(x=0)∧(x=1) -> 1 >;
        < ¬(x=0)∧¬(x=1) -> fib(x-1)+fib(x-2) >] }
```

When the termination of an F-program f with k ($k\geq 2$) control variables is to be confirmed in KANSUU, the following Algorithm 3.2 is applied. Let D_i be an expected set of a control variable x_i, and p_i on D_i be the bottom predicate of g_i ($1\leq i\leq k$), where $g_i(x_i)$ are the parameters corresponding to x_i in the recursive calls of f respectively.

Algorithm 3.2:

Prove a sequent

$$p_1(x_1)\vee\ldots\vee p_k(x_k) \rightarrow q_1(\underline{x}_1)\vee\ldots\vee q_n(\underline{x}_n),$$

where $q_1(\underline{x}_1),\ldots,q_n(\underline{x}_n)$ are the condition parts of non-recursive fragments of f. If it is a provable sequent, then the termination is confirmed, else not confirmed.

Example 3.5:

Let us consider the F-program of equal(x,y) given by (2.1). The control variables of this program are x and y. The bottom predicate of hd on list is atom, and the bottom predicate of tl is atom too. Thus we have to prove a sequent

$$\text{atom}(x)\vee\text{atom}(y) \rightarrow (\text{atom}(x)\wedge\text{atom}(y))\vee(\text{atom}(x)\wedge\neg\text{atom}(y))\vee(\neg\text{atom}(x)\wedge\text{atom}(y)).$$

This is easily proved, therefore we can confirm the termination of (2.1).

3.4 Verification of Algorithms for the Termination

We shall define the termination of F-programs in a formal manner, and verify Algorithms 3.1 and 3.2.

Definition 3.3: Termination of F-programs

If

$$D=\text{Dom}(f^{\infty})$$

holds for an exclusive and exhaustive F-program f with the expected set D

given by the data types of the variables, then we say that f terminates for D.

Algorithm 3.1 is verified by the following lemma.

Lemma 3.1:

Let f be an exclusive and exhaustive F-progrm with a control variable, say x, D be its expected set, g(x) be the parameter which replaces x in the recursive call of f and p on D be a bottom predicate of g. It is assumed that $q_1(x)$, ..., $q_n(x)$ are condition parts of non-recursive fragments. If the sequent

$$p(x) \rightarrow q_1(x) \vee q_2(x) \vee \ldots \vee q_n(x)$$

is provable, then f terminates for D.

In order to confirm the termination of F-programs with two or more control variables, Lemma 3.1 is easily generalized as follows.

Theorem 3.1:

Let f be an exclusive and exhaustive F-program with control variables $x_1,\ldots,x_k$, D_i be an expected set of x_i, $g_i(x_i)$ be the argument of the recursive calls which replaces x_i and p_i on D_i be a bottom predicate of g_i $(1 \leq i \leq k)$. Let $q_1(\underline{x}_1),\ldots,q_n(\underline{x}_n)$ be condition parts of non-recursive fragments, where $\underline{x}$ represents the arguments of q_j. If

$$p_1(x_1) \vee \ldots \vee p_k(x_k) \rightarrow q_1(\underline{x}_1) \vee \ldots \vee q_n(\underline{x}_n)$$

is a provable sequent, then f terminates for D, where $D=D_1 \times \ldots \times D_k$.

4. AN IMPLEMENTATION OF THE SUPPORT SYSTEM

Keio Support system for functional Recursive programming (KSR) based on KANSUU will be presented. If knowledge on the user's intended program has been given to KSR, the system proposes the information for making his program consistent and terminative. Thus the user can write consistent and terminative programs in cooperation with the KSR system. The detail of this system and examples of the real conversation records are shown in [NAG80a,NAG80b].

4.1 Purposes of the KSR System

KSR is not the language processor, and most of its users write programs in a particular language. So the user of KSR gives specifications and bottom predicates of primitive of the language at first. Next he gives a fragment of the intended program, and the system inspects guard predicates and examines whether the fragment is trivial or not. If some conditions are wrong, they are indicated. KSR is provided with a simple text editor capable of correcting a fragment.

Whenever a fragment is added, KSR examines guard predicates, triviality and exclusiveness of the sequents. If an error occurs, the system indicates it and accepts corrections.

When a whole F-program defining a function is given, the exhaustiveness is checked. Having detected logical errors, KSR points out how to correct them. After a consistent F-program is given, the system examines its termination. KSR confirms the termination of the program only when it can be assured. If the user needs to convert it into Lisp program, the system translates it into an efficient Lisp program.

The main purpose of our system is to help one to construct unerring programs in their development process. Algorithms for problem solving are thought out by programmers, not the computer, and KSR only assists them to design software or to develop programs. Besides, the correctness of programs which have been written can be verified by several automatic verifiers [SUZ75]. Especially, since KANSUU is related to our TKP [NAK79], TKP is a suitable tool for proving the correctness of programs written with KSR.

4.2 Organization of KSR-1

Our system has been implemented on minicomputers PDP 11/21 (28K words, 16bits/word) and PDP 11/34 which are installed in Department of Administration Engineering of Keio University. It has been written in the Lisp language, and consists of two parts. The first part, called KSR-1, is concerned in checking data types and logical errors, and the second, called KSR-2, is related to the termination and conversion. There are two monitors controlling all modules of our system, so that they enable us to write programs interactively.

As the first and second parts are separate phases in KSR, there are two monitors for them. We describe the first in this section, the second in Section 4.3.

A man interactively uses KSR with a CRT or teletype terminal connected to PDP 11. An outline of the first part of KSR is shown in Fig. 4.1

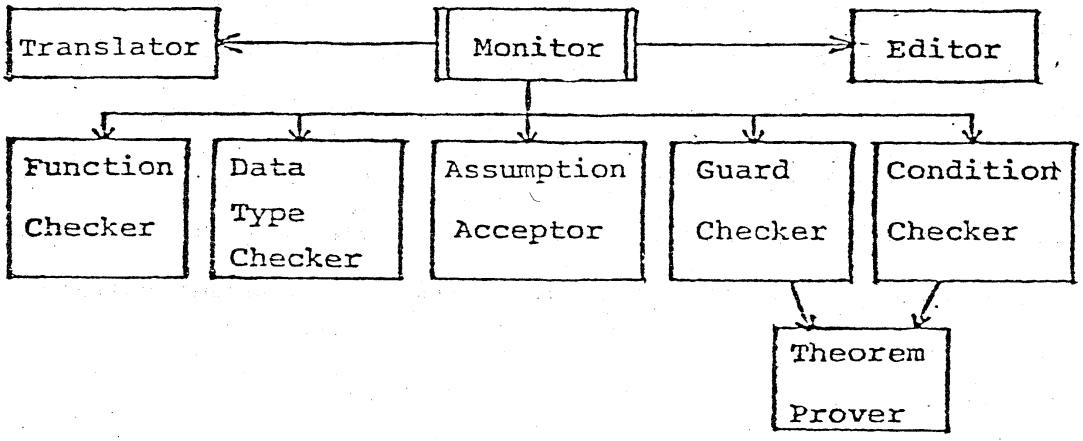

Fig. 4.1 An outline of KSR-1

Monitor analyzes the user's commands, and calls the requisite routines. Our system provides twelve commands, i.e., BYE, PCH, EDIT, PFN, SPEC, FUNC, DEF, INIT, ABORT, AINIT, FINIT and ASP. BYE causes a return to the LISP interpreter. PCH punches some defined functions to the paper tape (PDP 11/21) or the disk device (PDP 11/34). EDIT accepts the corrections of defined functions. SPEC specifies a function, that is, the function name, data types of arguments and the function, and guard predicates are specified. DEF defines a function. ASP is provided for assumptions which are used in Guard Checker and Data Type Checker. PFN specifies primitive functions. INIT initializes the system by cancelling the user input. FINIT and AINIT also initialize it by cancelling the user's F-programs and assumptions respectively. ABORT aborts the user input which is just given.

4.3 Design of KSR-2

An outline of KSR-2, which checks the termination of F-programs and converts them into Lisp programs, will be described. On PDP 11/34, KSR-1 gives F-programs to its disk device, and KSR-2 accepts them from the device

and accomplishes its task.

An outline of KSR-2 is shown in Fig. 4.2

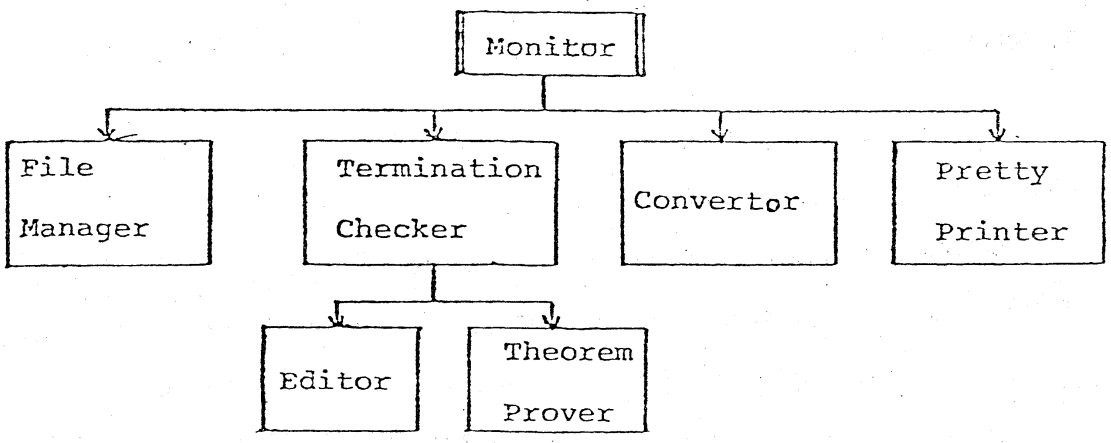

Fig. 4.2 An outline of KSR-2

Monitor analyzes the user's commands and calls the requisite routines. KSR-2 provides five commands, i.e., BYE, FILE, TERM, CONV, INIT, QUIT and PRINT. BYE causes a return to the LISP interpreter. FILE calls File Manager, which reads F-programs from the disk, writes Lisp programs to the device, saves and unsaves a set of bottom predicates to the disk. TERM causes Termination Checker which checks termination of the specified function. It accepts bottom predicates given by the user, and it provides Text Editor for their correction. It also uses Theorem Prover as the same as KSR-1. When the user would like to translate the F-programs into Lisp programs, he types CONV. PRINT prints F-programs and Lisp programs with indentations. INIT initializes the KSR system. QUIT causes a return to the command mode.

By the restriction of memory size of our computer, KSR-1 and KSR-2

work in separate phases. File Manager treats only outputs and inputs of KSR-2. Later part of this section describes on the other routines.

5. CONCLUSIONS

In order to conclude our study, we shall describe the limitations of KSR and comparison with related work.

KANSUU has several limitations on the forms of programs. For example, KANSUU does not accept free variables, functional arguments and mutual recursion. Moreover, an efficient and powerful method of semantic description is to be desired. The method should satisfy the following three conditions. First, it is to be powerful enough to describe semantics of problems and the programming language. Second, a user is to be able to easily describe them with the method. Finally, the method should be subject to effective processing on a computer. KSR is a pilot model realizing KANSUU, and a programming system which provides interactive facilities. In spite of using the LISP interpreter on a minicomputer, KSR spends only an acceptable amount of time for every response to the user. This system accepts both top-down and bottom-up programming. In KSR, semantics of a program is partly given as assumptions. Since the assumption sometimes includes pattern variables, our system has a pattern matching facility.

Lucid [ASH76] is an excellent attempt to realize Dijkstra's discipline [DIJ76], and the user of this system can verify the correctness of his program during its development process. However, the author believes that it is more practical to deal with consistency and correctness separately. If an inconsistent program is given, an automatic verifier fails to prove

the correctness of the program. So that when the verifier fails to prove the correctness, we cannot distinguish the two possibilities of inconsistency and incorrectness. KSR can detect the inconsistency of programs. There exist, on the other hand, many programs which are consistent but incorrect, and KSR can detect no errors of such programs. thus, the verifier should be used for the verification of programs which have been checked by KSR.

Although most existing program verifiers attempts to verify only the correctness of completed programs, some aspects of their techniques have been utilized in our work. For example, the theorem proving techniques of KSR are similar to the provers of the Stanford University verifier [SUZ75] and of our TKP [NAK79]; and Lemma 3.1 has been proved in the similar way to Burstall's structural induction [BUR69].

A verifier as a debugging tool based on Hoare's axiomatic basis and an assertion method has been proposed by Brand [BRA78], and it is useful for proving the correctness of completed programs. However, the author believes that our interactive approach is more practical in times of developing programs.

So far as Lisp programming is concerned, the aim of Wertz's PHENARETE system [WER79] is the same as our system. However KANSUU does not restrict programming language, so that we can write specifications and programs in a uniform way with KSR.

ACKNOWLEDGEMENTS

This paper is a revised version of "A Methodology for Recursive Program Construction, Technical Report 7901, Dept. of Administration

Engineering, Keio Univ., 1979". The author is indebted to Professors Hidetoshi Takahashi, Toshio Nishimura, Shoji Ura and Masakazu Nakanishi for this revision.

REFERENCES

[ALL78] Allen, J.R.: Anatomy of LISP, McGraw-hill, N.Y., 1978

[ASH76] Ashcroft, E.A. and Wadge, W.W.: Lucid - a formal system for writing and proving programs, SIAM J. Comput., Vol. 5, No. 3, 1976, pp. 336-354

[BAC78] Backus, J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, Commun. ACM, Vol. 21, No. 8, 1978, pp. 613-641

[BRA78] Brand, D.: Path calculus in program verification, Jour. ACM, Vol. 25, No. 4, 1978, pp. 630-651

[BUR69] Burstall, R.M.: Proving properties of programs by structural induction, Computer J., Vol. 12, 1969, pp. 41-48

[DIJ76] Dijkstra, E.W.: A Decipline of Programming, Prentice-Hall, N.J., 1976

[GEN34] Gentzen, G.: Unterschungen ueber das logische Schliessen, Mathematishe Zeitschrift, Vol. 39, 1934-35, pp. 176-210, 405-431

[HOA69] Hoare, A.C.: An axiomatic basis for computer programming, Commun. ACM, Vol. 12, No. 10, 1969, pp. 576-580,583

[KLE52] Kleene, S.C.: Introduction to Metamathematics, North-Holland Pub., Amsterdam, 1952

[LAN66] Landin, P.J.: The next 700 programming languages, Commun. ACM, Vol. 9, No. 3, 1966, pp. 157-164

[McC62] McCarthy, J. et al.: LISP 1.5 Programmer's Manual, The M.I.T. Press, Mass., 1962

[NAG80a] Nagata, M.: An Approach to Interactive Debugging for Functional Recursive Programming, Technical Report 8003, Department of Administration Engineering, Keio University, 1980

[NAG80b] Nagata, M., Akiyama, T., and Fujikake, Y.: An interactive supporting system for functional recursive programming, Proc. IFIP Congress 80, North-Holland, Amsterdam, 1980 (to be published)

[NAK79] Nakanishi, M., Nagata, M., and Ueda, K.: An automatic theorem prover generating a proof in natural language, Proc. IJCAI, 1979, pp. 636-638

[SUM77] Summers,P.D.: A methodology for LISP program construction from examples, Jour. ACM, Vol. 24, No. 1, 1977, pp. 161-175

[SUZ75] Suzuki, N.: Verifying programs by algebraic and logical reduction, Proc. of Intern. Conf. on Reliable Software, 1975, pp. 473-481

[WER79] Wertz, H.: Automatic program debugging, Proc. of IJCAI, 1979, pp. 951-953