An Approach to Construction of Functional Programs*

MORIO NAGATA**

An approach to the construction of reliable functional recursive programs is proposed. If a programmer writes his/her programs as specified in our approach, called KANSUU, logical errors in the programs can be detected automatically. The termination of certain types of programs can also be confirmed automatically.

Whenever a fragment of a functional program is given, its logical properties related to other fragments are inspected. Using automatic theorem proving techniques (propositional calculus), KANSUU indicates conditions which are necessary for the intended program. If the basic relations of functions and predicates have been given, then KANSUU may confirm the termination of functional recursive programs by using propositional calculus. Using KANSUU, we have implemented an interactive support system for functional recursive programming and a consultation system for Lisp programming.

KANSUU can provide information for the correction of inconsistent fragments of given functional programs. This approach is different from the viewpoint of automatic program verification or traditional debugging aids. Moreover, KANSUU uses simple and efficient algorithms. Thus, this approach will be useful for building a comprehensive programming system.

1. Introduction

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 the most promising one [3]. Research papers have been presented on programs in the functional style. Many papers are on automatic verification or synthesis of functional programs [4, 13, 14], however, little works has been done on the debugging of such programs [5, 16]. This paper gives a new approach to interactive debugging of functional recursive programs.

Our approach, called KANSUU (Keio AdvaNced approach for SUpporting fUnctional programming), presents notations representing programs in functional styles. If a programmer writes his/her 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. Note that the author does not aim to present a new programming language. We propose a new approach to support the writing of reliable functional programs.

Every technique for automatic program verification is to show formally that a given program satisfies a property expected by the programmer. Such a technique assumes that the program is logically consistent. If the automatic verifier fails to prove the correctness of the program, then we cannot distinguish the two possibilities of inconsistency and incorrectness. Moreover, since logical errors in consistency and/or the termination of programs are apt to be detected during their executions, the cost of debugging is high. Thus, from a practical point of view, we must develop an automatic method for correcting inconsistent functional programs. KANSUU provides such a method by using simple and efficient automatic theorem proving techniques (propositional calculus), that is, KANSUU can automatically detect inconsistencies of functional programs and point out how to improve the programs.

Using the same technique, KANSUU also confirms the termination of certain types of functional recursive programs. If KANSUU and traditional program verifier can be effectively combined, a useful interactive support system for functional recursive programming will be built.

KANSUU is not bound to a particular programming language. In this approach, a particular programming language can be specified by data types and primitive functions which have been built in the language. KANSUU assumes that a program consists of control structures of recursion, McCarthy's conditional expression (or *if-then-else* construct) and functional composition. The main features of KANSUU are listed below:

1) Using automatic theorem proving techniques, KANSUU detects not only the existence 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) An automatic mechanism confirming the termination of certain types of functional recrusive programs is

---

*This work was partly supported by the Science Foundation Grant of the Ministry of Education, Science and Culture of Japan, Grant-in-Aid for Encouragement for Young Scientists, Grant No. 56790033.

**Dept. of Administration Engineering, Faculty of Science and Technology, Keio University, 3-14-1 Hiyoshi, Yokohama 223, Japan.

embedded.

4) KANSUU is useful for the implementation of interactive programming systems.

An interactive support system based on KANSUU, called KSR (Keio Support system for functional Recursive programming), has been implemented in the Lisp language on the PDP 11 minicomputer. We have written many small programs and several medium-sized programs using KSR. Since KANSUU proposes simple and efficient algorithms, KSR attains our purpose even though it is implemented on a minicomputer.

An outline of KSR and its real conversation record are presented in [11]. However, [11] does not describe the algorithms on which KSR is founded. Thus this paper describes the basic idea of our approach and algorithms for detecting logical errors and testing the termination of such programs. The correctness of these algorithms has been verified by using propositional calculus and mathematical induction [10].

Moreover, using KANSUU, we are now implementing a consultation system for noviece Lisp programmers, and the effectiveness of this approach has been demonstrated by our experience. For someone building an interactive programming system, KANSUU will provide useful guidance.

2. Notions for Illustrative Examples

2.1 Basic Notations

Showing a list processing programs as examples, we first introduce the basic notions of list structures. 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. The fundamental structure of a list is a *b-list* which is defined as follows (cf. [1]):

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. b_1 and b_2 in the definition of the b-list $(b_1 : b_2)$ are called the *first element* and the *second element* of the proper b-list respectively. A b-list is sometimes called a list. $(x_1 : (x_2 : (\cdots : (x_k : (\ )) \cdots)))$ will often be abbreviated as $(x_1, x_2, \cdots, x_k)$, and called a *k-list*.

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*.

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 in KANSUU. For example, it is assumed that

Boolean, integer, string

are declared. We assume that Boolean values, integers and character strings are atoms. By the definition 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* κ *list*' in KANSUU.

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(x) is one of the conditions which should hold for the arguments of *hd*. We call such a condition a *guard proposition* of the function.

2.2 Specification and F-program

When writing functional programs, we must give *specifications* of data types and primitive functions in KANSUU. A specification of a function is written as:

$$f(v_1 \cdot d_1, \cdots, v_n \cdot d_n) \Rightarrow d; p,$$

where f is the name of the function, v_i is a variable, d_i is its data type, d is the type of the value of the function and p is a guard proposition. This proposition is optional in the specification. If it is assumed that hd and tl operate on lists not atoms, then examples of the specifications of our discussion are as follows:

Data types: ((*Boolean, integer, string*) = *atom*) κ *list*
Primitive functions: hd(x.*list*)⇒*list*; $\neg$atom(x),
tl(x.*list*)⇒*list*; $\neg$atom(x),
cons(x.*list*, y.*list*)⇒*list*,
atom(x.*list*)⇒*Boolean*,
eq(x.*atom*, y.*atom*)⇒*Boolean*

A component defining a function is of the form

⟨condition part→expression part⟩

in our approach. This is called a *fragment*. The condition part is usually a proposition. If the expression part is a proposition too, the fragment $\langle C \rightarrow E \rangle$ is called a *propotional fragment* and the value is $\neg C \vee E$.

A collection of fragments

$$[\![\text{fragment}_1; \text{fragment}_2; \cdots; \text{fragment}_n]\!]$$

is called *the conditional*. The value of the conditional in a valuation is the value of the expression part of a fragment whose condition part is *true*. McCarthy's conditional expression,

$$[C_1 \rightarrow E_1; \cdots; C_n \rightarrow E_n],$$

is equivalent to the conditional,

$$[\![<C_1 -> E_1>;$$
$$<\neg C_1 \wedge C_2 -> E_2>;$$
$$\vdots$$
$$<\neg C_1 \wedge \neg C_2 \wedge \cdots \wedge \neg C_{n-1} \wedge C_n -> E_n>]\!].$$

An F-program is written as:
{specification⇐the conditional}.
An example of the F-program is as follows:

$$\{\text{equal}(x.list, y.list) => Boolean;$$
$$<= [\![<\text{atom}(x) \wedge \text{atom}(y) -> \text{eq}(x, y)>;$$
$$<\text{atom}(x) \wedge \neg\text{atom}(y) -> false>;$$
$$<\neg\text{atom}(x) \wedge \text{atom}(y) -> false>; \quad (1)$$
$$<\neg\text{atom}(x) \wedge \neg\text{atom}(y)$$
$$-> \text{equal}(\text{hd}(x), \text{hd}(y)) \wedge \text{equal}(\text{tl}(x), \text{tl}(y))>]\!]\}$$

3. Detection of Logical Errors

3.1 Properties of F-programs

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

A: Each condition part is not identical to *true* or *false*.

B: Disjunction of all condition parts is always *true*. (*Exhaustiveness*)

C: Conjunction of condition parts of distinct fragments are always *false*. (*Exclusiveness*)

We easily understand Property A. If a fragment does not satisfy Property A, then the fragment is called a *trivial fragment*. Property B confirms that the value of an F-program consisting of non-recursive fragments is never undefined under its specification. If Boolean constant *true* is used in the condition part, this property always holds. In KANSUU, *true* may be written in a similar manner to Lisp programming. In this case, KANSUU automatically assumes the actual proposition represented by *true*.

If there exist two fragments of an F-program which do not satisfy Property C, then those condition parts are non-deterministic. In McCarthy's conditional expression, every $C_k \to E_k$ depends on all $C_i \to E_i (1 \le i \le k)$. Thus, Property *C* is not always assumed in the expression. However, in the conditional of our approach, all fragments are independent of each other.

Certain types of logical errors of F-programs can be detected by algorithms based on these properties. Suppose that one of fragments of (1) is lacking, then the property *B* is not satisfied. Algorithms for detecting such errors will be described.

3.2 Automatic Theorem Proving and Trivial Fragment

We shall use a formal system which is a subset of Gentzen's LK [7]. The formal system is useful for describing the above properties and our algorithms. In this paper, we use the notations of the formal system which are described by Kleene [8].

In KANSUU, a *sequent* is used as an internal representation for detecting logical errors. The sequent is the same as Gentzen's sequent, that is, a sequent is a formal expression of the form

$$A_1, \cdots, A_l \to B_1, \cdots, B_m$$

where l, $m \ge 0$ and $A_1, \cdots, A_l, B_1, \cdots, B_m$ are propositions. When l, $m \ge 1$, the above sequent has the same interpretation as

$$A_1 \wedge \cdots \wedge A_l \text{ implies } B_1 \vee \cdots \vee B_m.$$

The interpretation extends to the cases where $l=0$ or $m=0$ by interpreting $A_1 \wedge \cdots \wedge A_l$ for $l=0$ (the 'empty conjunction') as *true* and $B_1 \vee \cdots \vee B_m$ for $m=0$ (the 'empty disjunction') as *false*. The number of logical connectives of a sequent is called the *degree* of the sequent.

Logical axioms of KANSUU are:

$$\Gamma_1, A, \Gamma_2 \to \Delta_1, A, \Delta_2,$$
$$\Gamma \to \Delta_1, \text{true}, \Delta_2$$

and

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

where *A* is any proposition, and Greek capitals represent zero or more propositions. Let *P* and *Q* be arbitrary propositions, then the logical rules of inference are the following.

Rules of Inference:

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

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

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

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

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

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

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, that is, 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 axion. Figure 1

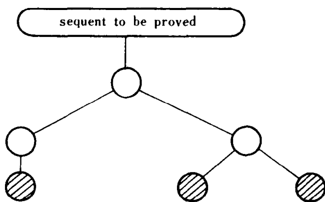

Fig. 1 An example of the complete proof tree.

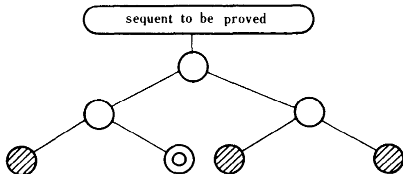

Fig. 2 An example of an incomplete proof tree.

shows an example of the complete proof tree. Each terminal node '●' is a logical axiom.

When the sequent is not a provable sequent, a *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. Figure 2 is an example of the incomplete tree. A terminal node '◎' represents the non-provable terminal node.

There exist predicates (atom, eq etc.) in F-programs, however, when we analyze the logical relations between condition parts, we consider only propositions (atom(x), eq(x, y) etc.) in our approach. For example, we may write atom(x) and atom(y) as P and Q respectively, where P and Q represent propositions (cf. Example 1). Thus, we use only Gentzen-type propositional calculus, not the predicate calculus, as the automatic theorem proving technique in KANSUU. In this technique, the number of logical symbols of a sequent is always reduced by applying the relevant rule of inference. Therefore, if any sequent is given, then we can decide whether it is provable or not with an efficient way, that is, the complete or imcomplete proof tree of the sequent can always be constructed automatically.

Now, the following algorithm tests whether a fragment, $\langle C \to E \rangle$, is a trivial fragment.

Algorithm 1:
Try to prove two sequents

$$C\to \quad \text{and} \quad \to C,$$

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

3.3 Exhaustiveness and Exclusiveness

We shall describe algorithms for detecting logical errors by using Property B and C. If the conditional

$$[\![<C_1 -> E_1>; \cdots; <C_n -> E_n>]\!]$$

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

Algorithm 2:
Try to prove sequents

$$C_i \wedge C_j \to$$

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 3:
Try to prove a sequent

$$\to C_1 \vee C_2 \vee \cdots \vee C_n.$$

If it is a provable sequent, return "exhaustive", else apply Algorithm 4.

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

Algorithm 4:
If a sequent

$$A_1, A_2, \cdots, A_k \to A_{k+1}, \cdots, A_n$$

is the non-provable terminal node of the incomplete proof tree of

$$\to C_1 \vee C_2 \vee \cdots \vee C_n,$$

then propose the following proposition as the missing condition part.

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

where A_i is a proposition.

Example 1:
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, we may write propositions atom(x) and atom(y) as P and Q respectively. Therefore,

$$\to (P \wedge Q) \vee (P \wedge \neg Q) \vee (\neg P \wedge \neg Q)$$

is not a provable sequent, and the sequent of degree 0,

$$Q \to P,$$

that is,

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

is the non-provable terminal node of the incomplete

proof tree. We can find that a fragment whose condition is $\neg$atom(x) $\wedge$ atom(y) should be added.

If there are two or more different non-provable terminal nodes, KANSUU cannot determine which condition parts are missing. It can only proposes several possibilities.

Example 2:
Let

C_1: atom(x) $\wedge$ atom(y) and C_2: atom(x) $\wedge$ $\neg$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 not provable sequents of degree 0.

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

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, $\neg$atom(x) $\wedge$ atom(y) and $\neg$atom(x) $\wedge$ $\neg$atom(y) are exclusive and exhaustive. If an F-program satisfies all of property A, B and C, it is called a *consistent F-program.*

The new idea of the algorithm on the consistency of F-programs is that the algorithm provides information for the correction of inconsistent programs by using simple automatic theorem proving techniques.

4. Test for 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 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 particular forms.

If our approach confirms the termination of certain types of F-programs, then they always terminate. However, when it cannot confirm them, they may or may not terminate.

4.1 Termination of F-programs

Let us assume that an F-program

$$\{\text{fib}(x.\textit{non-negative integer}) = > \textit{non-negative integer};$$
$$< = [\![< x=0 - > 1 > ;$$
$$< \neg(x=0) - > \text{fib}(x-1) + \text{fib}(x-2) >]\!]\} \quad (2)$$

is given by a programmer. We can find that the execution of the program can not terminate for $x \leq 1$. 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:

$$\{h(x.\ \text{type}) = > \text{type}';$$
$$< = [\![< p(x) - > e(x) > ;$$
$$< \neg p(x) - > a(h(b(x))) >]\!]\} \quad (3)$$

where e, a and b are primitive, specified or defined functions. Our approach can be extended to mutual recursive programs, however, it is assumed that this is not a mutual recursive program in this paper.

We shall discuss the confirmation of the termination of F-programs satisfying the following properties:

1) Consistent F-programs
2) Recursive but not mutual recursive F-programs
3) F-programs including only primitive, specified and defined functions

Program (3) can be interpreted as

$$\{h^{n+1}(x.\ \text{type}) = > \text{type}';$$
$$< = [\![< p(x) - > e(x) > ;$$
$$< \neg p(x) - > a(h^n(b(x))) >]\!]\},$$

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

$$\text{Dom}\ (h^k)$$

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.

We conclude that, if

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

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

4.2 Bottom Predicate and Control Variable

In order to show the termination of a given F-program, we make use of a well-founded set (D, $<$) which is defined as a set of the expected set D with a partial ordering $<$ having the property that there can be no infinite descending chain of elements of D. We assume that (D, $<$) is a well-founded set which consists of a set of elements of D and the ordering $<$ defined on the elements. For proving the termination of the F-program, we directly use the well-founded set (D, $<$) and the text of the program.

By the definition of the well-founded set, there exist minimum elements in the set. Suppose that $b(x)<x$ holds in (3). If p of (3) is *true* on every minimum element of $(D, <)$ 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 1:
Let $(D, <)$ 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:
If 1 is a set of all lists, then atom is the bottom predicate of hd. Figure 3 demonstrates that hd(b) immediately preceeds b for any proper b-list b, here each α_i represents an atom (cf. [14]).

Example 4:
If 1 is a set of all k-lists, then null is the bottom predicate of tl (Fig. 4), where null(x) is *true* if x is ε, and *false* otherwise. Here α represents an arbitrary atom.

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

Definition 2:
Let f be an F-program with $n(n\geq 2)$ variables $x_1, x_2,\cdots, x_n$. 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

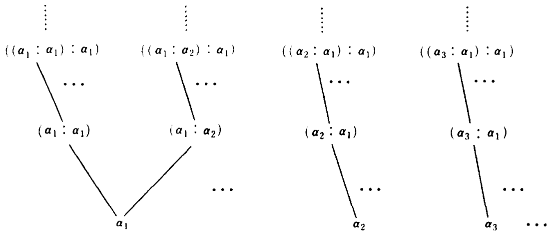

Fig. 3 Structure of b-lists.

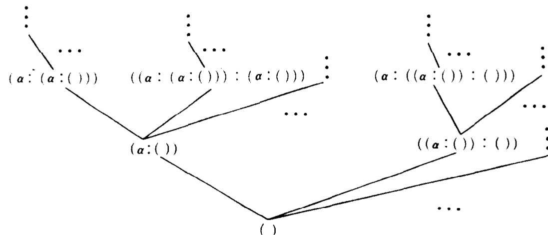

Fig. 4 Null and tl on k-lists.

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 method (see Algorithm 5).

Example 5:
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(tl(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(tl(x), append(y, cons(hd(x), ε)))>⟧}

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

4.3 Test of the Termination

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 the expected set given by data types of its variables, therefore the termination of the program should be confirmed on the basis of properties of data types. In the following descriptions, Algorithm 5 will assure the termination of F-programs with one control variable, while Algorithm 6 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 $q_1(x),\cdots, q_n(x)$ 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 5:
Prove a sequent

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

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 confirmed by this algorithm.

If an F-program does not include composition of functions as the parameter of recursive call, Algorithm 5 is essentially the same method as the termination function approach. But, 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 $p_1,\cdots, p_m$ be bottom predicates of $g_1,\cdots, g_m$ respectively and let the argument of the recursive call be

$$g_m(g_{m-1}(\cdots(g_1(x))\cdots)),$$

then the bottom predicate of the composition function is

$$p_1(x) \vee p_2(g_1(x)) \vee \cdots \vee p_m(g_{m-1}(\cdots(g_1(x))\cdots)).$$

Thus, suppose an F-program (2) is given, we find that

$$P \vee Q \rightarrow P$$

is not a provable sequent, where P and Q are $x=0$ and $(x-1)=0$ respectively. The system will detect that a non-recursive fragment whose condition part is $(x-1)=0$ is missing. Therefore the programmer is led to the corrected F-program.

When the termination of an F-program f with $k(k \geq 2)$ control variables is to be confirmed in KANSUU, the following Algorithm 6 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 6:
Prove a sequent

$$p_1(x_1) \vee \cdots \vee p_k(x_k) \rightarrow q_1(x_1) \vee \cdots \vee q_n(x_n),$$

where $q_1(x_1), \cdots, q_n(x_n)$ are the condition parts of non-recursive fragments of f. If it is a provable sequent, then the termination is confirmed, else not confirmed.

The termination of (1) can be confirmed by Algorithm 6. The basic ideas of Algorithm 5 and 6 are influenced on the termination function approach. However, they use simple and efficient theorem proving techniques. Thus these algorithms enable us to make an automatic method for confirming the termination of recursive programs. The types of programs treated in KANSUU are restricted and bottom propositions must be given, but our algorithms are useful for the writing of programs including compositions of functions as two or more parameters of recursive calls. Moreover, Algorithm 5 and 6 may provide information for the correction of non-terminative programs.

5. The KSR System

If knowledge on the user's intended program has been given to KSR, the system provides information for making his program consistent and for assuring termination. Thus the user can write consistent and terminating programs in cooperation with the KSR system.

At first the user gives specifications and bottom predicates of primitive functions of the language. Next he gives a specification of the intended program. Whenever the user writes a fragment of the program, the KSR system inspects guard propositions and examines the fragment's triviality and exclusiveness. If some conditions are wrong, they are indicated. KSR is provided with a simple structure editor capable of correcting fragments [12].

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.

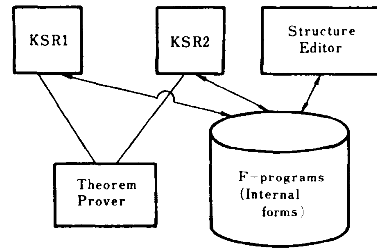

Fig. 5 Structure of KSR.

KSR mainly consists of three parts, i.e., KSR1, KSR2 and the structure editor. These are separately implemented now. KSR1 is concerned with the consistency and KSR2 is related to the termination. Both of them use the same automatic theorem prover. Figure 5 shows the structure of KSR. The user writes programs in M-expression-like forms shown in this paper, but KSR stores them in S-expression-like forms.

KSR is a pilot model realizing KANSUU, and a programming system which provides interactive facilities. In spite of using a LISP interpreter on a minicomputer, every response returns in a few seconds. Thus KSR only spends an acceptable amount of time for every response to the user.

We have written many small programs and several medium-sized programs by using KSR. An example of the real conversation record of writing a small program is shown in [11]. An example of a medium-sized one is a symbolic manipulation program. The program simplifying symbolic polynomials consists of about twenty fragments. Even though KSR is implemented on a minicomputer, we can write such programs with KSR. Moreover, we are now implementing a consultation system for Lisp programming.

6. Conclusions

We shall conclude by comparing our approach with other related works.

Lucid [2] is an excellent attempt to verify the correctness of the user's programs during the development process. However, the author believes that it is practical to deal with consistency and correctness separately. If an inconsistent program is given, an automatic verifier fais 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. KANSUU can detect the inconsistency of programs. There exist, on the other hand, many programs which are consistent but incorrect, and KANSUU can not detect errors of such programs. Thus, the verifier should be used for the

verification of programs which have been checked by KSR.

Algorithms in KANSUU are useful for the detection of logical errors before the written programs are executed Before the execution, KANSUU first checks the consistency and the termination of the program, and a verifier then proves the correctness. The use of these algorithms will save much labor and time of the programmer. Although many existing program verifiers attempt 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 KANSUU are similar to the provers of verifiers [4, 13]. A verifier as a debugging tool based on Hoare's axiomatic basis and an assertion method has been proposed [5], and it is useful for proving the correctness of completed programs. However, the author believes that our interactive approach is practical at the times programs are developed. So far as Lisp programming is concerned, the aim of Wertz's PHENARETE system [16] is the same as our system. But KANSUU does not restrict programming language, so that we can write specifications and programs in a uniform way.

This paper has proposed a new approach to interactive debugging for functional recursive programming. On the other hand, it is necessary that our approach and other methods, i.e. automatic verifiers, traditional debugging aids and new programming tools [6, 9, 15, 17], should be combined. A powerful programming system would be built, if they are effectively combined.

Acknowledgements

The authour is indebted to Professors Hidetosi Takahasi, Toshio Nishimura, Shoji Ura, Yoshio Hayashi and Masakazu Nakanishi for accomplishment of this work.

References

1. Allen, J. R. Anatomy of LISP, McGraw-hill, N.Y. (1978).
2. Ashcroft, E. A. and Wadge, W. W. Lucid-a formal system for writing and proving programs, *SIAM J. Comput.* 5, 3 (1976), 336–354.
3. Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Commun. ACM* **21**, 8 (Aug. 1978), 613–641.
4. Boyer, R. S. and Moore, J S. A Computational Logic, Academic Press, N.Y. (1979).
5. Brand, D. Path calculus in program verification, *Jour. ACM* **25**, 4 (Dec. 1978), 630–651.
6. DoD: Requirements for Ada Programming Support Environment "stoneman" (February 1980).
7. Gentzen, G. Unterschungen ueber das logische Schliessen, *Mathematishe Zeitschrift* 39, 1934–35, 176–210, 405–431.
8. Kleene, S. C. Introduction to Metamathematics, North-Holland Pub., Amsterdam (1952).
9. Lieberman, H. and Hewitt, C. A session with Tinker: Interleaving program testing with program writing, *Conference Record of the 1980 LISP Conference* (Aug. 1980), 90–99.
10. Nagata, M. Interactive debugging for functional recursive programming, RIMS Kokyuroku, No. 396, Kyoto University (Sept. 1980), 131–169.
11. Nagata, M., Akiyama, T. and Fujikake, Y. An interactive supporting system for functional recursive programming, *Information Processing 80*, Lavington, S. (ed.), North-Holland, Amsterdam (October 1980), 263–268.
12. Nagata, M. and Orita, K. A structure editor of the support system for functional programming, WGSYM 15–1, IPSJ, (June 1981), 1–10 (in Japanese).
13. Nakanishi, M., Nagata, M. and Ueda, K. An automatic theorem prover generating a proof in natural language, *Proc. of 6th IJCAI* (August 1979), 663–638.
14. Summers, P. D. A methodology for LISP program construction from examples, *Jour. ACM* **24**, 1 (January 1977), 161–175.
15. Waters, R. C. The programmer's apprentice: Knowledge based program editing, *IEEE Transaction on Software engineering*, SE-8 (January 1982), 1–12.
16. Wertz, H. Automatic program debugging, *Proc. of 6th IJCAI*, (Aug. 1979), 951–953.
17. Wilander, J. An interactive programming system for Pascal, *BIT* 20 (1980), 163–174.

(Received November 24, 1981: revised May 26, 1982)