数理解析研究所講究録
第586巻 1986年 196-223

Strategic Bug Location Method for Functional Programs
関数型プログラムの戦略的バグ検出法

Naohisa TAKAHASHI and Satoshi ONO
高橋　直久　・　小野　諭

NTT Electrical Communication Laboratories,
3-9-11 Midoricho Musashino-shi Tokyo 180 Japan
NTT電気通信研究所
(〒180 東京都武蔵野市緑町3-9-11)

Abstract

This paper proposes a new algorithmic debugging method for functional programs, Strategic Bug Location (SBL). SBL produces a search graph, Strategically Projected Graph (SPG), using the dynamic execution history in conjunction with the static structure of the program. The method generates queries to the programmer and applies such operations as transformation and selection to the SPG. The repetition of such query and operations leads the programmer to the location of a bug. SBL improves the reduction of the search graph by incorporating bug location strategies in the transformation mechanisms of the search space. Furthermore, SBL facilitates grasping the abstract structure of a program during the bug location and also facilitates following the sequence of the queries using the hierarchical search graph.

1. Introduction

Functional programming languages have many excellent features which facilitate writing short and clear programs, as well as understanding and verifying these programs using clean mathematical semantics.[1-5] It has also been pointed out that

parallelism is easier to detect in a functional program than in an imperative one.[5] These points of view have led many researchers to study both dataflow machines[6-9] and parallel reduction machines[10-11] in an effort to find ways to execute functional programs more efficiently. Furthermore, the propagation of a bug in a functional program is simple to trace, because of the referential transparency accompanying functional languages. This has led the authors to study methodologies for debugging a functional program executed by a highly-parallel computer.[12,13]

By analyzing the execution history and the programmer's answers to the queries automatically generated by the debugging system, the debugging system mechanically detects a bug in the program.[12-14] Using procedures that select a query from the search space made from the execution history as well as reduce the search space through the query-and-answer procedure, these systems make it mechanically and efficiently possible to locate a bug even when a considerable amount of execution history is produced.

All bugs incorporated in the program text will appear in the execution results. Therefore, it is desirable to search for bugs using strategies derived from both the properties of bugs and the analysis of the static program text as well as from the execution history. However, most of the bug location systems proposed to date [12,14] analyze only the execution history without looking into the function dependencies produced by the static analysis of the program itself. Consequently, cases exist in which redundant questions are generated repeatedly for the same part of the program text. This problem is especially serious when a recursive function is called in many times. Furthermore, when an

illegal value is found in function's parameters, such bug location systems can not constrain the search space to the relevant execution history wherein the value was computed.

This paper proposes a new bug location method, called the Strategic Bug Location (SBL) method as a solution to this problem. SBL produces a search graph, named the Strategically Projected Graph (SPG), using both the dynamic execution history and the static structure of the program. The method iteratively applies such operations as transformation and selection to the SPG.

2. Background

2.1 Bug Location Method for Functional Programs

Using a functional program, many operations can be executed in parallel independently of the sequence of such operations in a program text. Therefore, once a bug is found to exist in a program, it is difficult to debug using conventional debugging tools like tracers, memory dumpers, and so on.[16] In addition, a dataflow machine, which is one of the promising vehicles for executing functional programs in parallel, uses tokens to carry data as well as eliminates the concepts of stack and memory.[6-9] Locating a bug by monitoring tokens is difficult, however, because of the complexity of the token arrival sequence during highly-parallel processing.

To solve this problem, we proposed a bug location method based on the analysis of data dependency.[12] Figure 1 illustrates a system employing the method which consists of a debugger and a database. The system stores dependency graphs in the database, which are produced by the analysis of both the

static program text and the execution history. The debugger updates the database by using the analysis results of the programmer's answer to the query which is automatically generated by the debugger. Through the iterations of such a query-and-answer procedure, the debugger successively narrows down the search space until it eventually locates the bug.

2.2 Dependency Graphs for Debugging Functional Programs

The static dependency in a program and its execution history are expressed by using directed graphs. The notations and data structures of the dependency graphs are as follows.

(1) Directed Graphs

A directed graph, G, is a tuple (Ng,Ag,Ig) where "Ng" is a set of nodes, "Ag" is a set of arcs connecting two nodes in "Ng", and "Ig" is an initial node. Suppose the existence of graph G and node $v \in Ng$. A predecessor graph of v is a maximal subgraph of G, which consists of nodes reachable to v by tracing arcs in G, and whose initial node is Ig. (In this paper, a predecessor graph of v is assumed to include itself.) A successor graph of v is a maximal subgraph of G, which consists of nodes reachable from v by tracing arcs in G, and whose initial node is v. A subordination graph of v is a maximal subgraph of G, which consists of such w's as w is a node in a successor graph of v and has no path from Ig to itself without going by way of v. An initial node of a subordination-graph of v is v.

(2) Instance Dependency Graph

A tuple (f,x,y), in which f is a function, x is an input parameter list and y is a result value list, is called an instance value. A pair of an instance value and its identifier is called an instance. When an instance value of an instance,

"i", is computed with an instance, "j", "j" is called a child instance of "i". An instance dependency graph is a directed graph which consists of all nodes, generated during execution, and arcs drawn from each instance to its child instance. Its initial node is an instance generated by executing a function at the start of the program.

(3) Instance Value Dependency Graph

The identifier of an instance is not essential for debugging pure functional programs. From this point of view, an instance value dependency graph is generated by forming all nodes having the same instance value in an instance dependency graph into one node. When an instance value, "i", is computed with an instance value, "j", "j" is called a child instance value of "i". We introduce a function, Ff, which returns the function part, "f", when it is applied to the instance value, v=(f,x,y).

(4) Function Dependency Graph

In a function dependency graph, a node corresponds to a function in a source program. An initial node corresponds to a function which is executed at the start of the program. The function dependency graph has two classes of arcs, namely call-arcs and parameter-arcs. Call-arcs represent the relationship of the reference between two functions. When some expressions in a function, "f", use the return value of the function, "g", a call-arc is drawn from a node, "f", to "g" with an arrow, i.e., f→g. For example, the function dependency graph for Example 1 below is shown in Fig. 2(a), because "f" calls "g" and "g" uses only primitive functions in this example.

[Example 1]:

```
f(x) = g(x)+1;
g(x )= x-2;
```

Parameter-arcs represent the relationship between the definition of an input parameter and its reference. When the input parameter for a function, "g", is computed with the value of a function, "f", a parameter-arc is drawn from a node "f" to "g" with a dotted arrow, i.e., f-->g. Fig. 2(b) illustrates a function dependency graph for the program Example 2 below in which a parameter for g uses the result of a function, "f".

[Example 2]:

```
h(x) = g(f(x));
g(x) = x*x;
f(x) = x+1;
```

Predecessor graphs, successor graphs and subordination graphs are defined for each class of arcs in a function dependency graph. For example, a call-successor graph of "v" is a maximal subgraph of G which consists of nodes reachable to "v" by using only the call-arcs in G. On the other hand, a successor graph of "v" is a maximal subgraph of G which consists of nodes reachable to "v" by using both the call-arcs and parameter-arcs in G. Similar subgraphs can also be defined in terms of predecessor graph and subordination graph.

2.3 Bugs in a Functional Program

Suppose that a function "f" is defined in a program and that "v" is an arbitrary instance value, (f,x,y). When "x" includes an unexpected (or illegal) input parameter for "f", v is called undefined. When x is a list of correct input parameters, v is true if f(x)=g in the specification, while v is false if not. When a function returns an erroneous result for correct input parameters, at least one bug exists in it, since a functional program is not history sensitive. Therefore, if a tuple of (f,x,y) is an instance value, "b", which satisfies both of the

following conditions, there exists a bug in the expressions in a function "f", by which f(x) is actually computed:

(1) An instance "b" is false,

(2) All child instance values of "b" are true.

The above "b" is called the "source" of a bug. A procedure, with which a debugger locates the source of a bug from the execution history, is named a bug location algorithm.

2.4 Problems in Conventional Bug Location Algorithms

Bug location algorithms proposed to date create the search space using only execution history(e.g. using an instance dependency graph for a PROLOG program[14] or using an instance value dependency graph for a functional program[12]). When using these algorithms, some cases exist in which redundant questions for inspecting the same part of the program text are generated repeatedly.

This problem is especially serious when a recursive function is called in many times. For example, consider a program which consists of a single erroneous recursive function. Here, a bug exists either in the recursive part or in the termination part. However, the above algorithms iteratively check instance values derived from the recursive part before they check any instance value derived from the termination part, since they select a node to separate the execution history into two parts. Such a selection mechanism does not work well when a bug exists in the termination part. It should be also pointed out that the above algorithms can not constrain the search space to relevant instances, when erroneous parameters are found to be applied to a function (e.g when an unsorted list is applied instead of a sorted list).

3. Strategically Projected Graph and Its Reduction

To solve the problems described in the previous section, it is important to debug a program using not only the execution history but also data reflecting the static structure of the program. The search space in the Strategic Bug Location method (SBL) is a strategically projected graph (SPG) which is produced by relating a function dependency graph to an instance value dependency graph. SBL transforms and selects the SPG and identifies the source of a bug according to the strategies derived from both the program structure and the properties of the bugs themselves. Consequently, SBL should require less queries of a programmer than the existing algorithms since SBL distinguishes instance values in the termination part and in the recursive part of a recursive function as well as analyzes the direction of bug propagation in greater detail. In this section, the SPG are defined as well as the operations for its reduction.

3.1 Strategically Projected Graph

Assume a function dependency graph, G=(Ng,Ag,Ig), and an instance value dependency graph, V=(Nv,Av,Iv), for a program P. Suppose an arc, "a" ($a \in Av$), exists from node v1 to v2. Also suppose that the function parts of v1 and v2 are f1 and f2, respectively. Therefore, a call-arc, "c" ($c \in Ag$), exists from node f1 to f2. In addition, Ff(Iv) becomes Ig. Suppose an homomorphic graph of V named H that is created by projecting nodes having the same function part to one node. Therefore, H is a sub-graph of G. Hence, we define a function, Fd, as

$$Fd(f) \equiv \{v \mid v \in Nv \text{ and } Ff(v)=f\}, \text{ where } f \in Ng.$$

The function, Fd, accepts a function name, f, and gives a set of instance values computed using f. We call Fd(f) a total

instance value set (TIVS) for f, and the union of TIVSs for all nodes of graph, S, is named a TIVS for S.

The SPG, S=(Ns,As,Is), generated from graphs G and V is either the initial graph, S0, defined below, or a graph obtained by applying an arbitrary number of operations described in Sections 3.2 and 3.3 to the graph S0.

The initial Graph, S0=(Ns0,As0,Is0), is a subgraph of G, and is defined as

* Ns0 $\equiv$ { f | f $\in$ Ng and (Fd(f) is not null) },
* As0 $\equiv$ { a | a $\in$ Ag and
(both the start/end points of "a" are in Ns0)}
* Is0 $\equiv$ Ig

In the next section, structured nodes are introduced. In contrast, nonstructured nodes are named basic nodes. For the basic nodes of an SPG, the function, Fd, is defined similarly as in the case of the function dependency graph. Apparently, Fd(S0)=Nv.

To constrain the search space, we introduce a function, Fs, which specifies the eligible nodes in TIVS. For the basic nodes of an SPG, the function, Fs, is then defined as

$$Fs(f) \equiv \{ v \mid v \in Nv \text{ and } (Fa(v) \cap Fd(f) = \{v\}) \},$$

where Fa(v) stands for a node set of the predecessor graph of "v". This definition implies that each node included in Fs(f) has no parent instances whose function parts are "f".

Fs(f) is named an eligible instance value set (EIVS) for "f". For any node "f" of an SPG, $Fd(f) \supseteq Fs(f)$. The union of EIVSs for all nodes of graph S is called an EIVS for S.

SBL first selects a node f of the SPG. It then selects an arbitrary instance value from Fs(f), and queries the programmer whether it is true, false or undefined. According to the

programmer's answer, it reduces the SPG using transformation/selection operations discussed in the following sections.

3.2 Transformation Operations for an SPG

Assume a function dependency graph, G, and an instance value dependency graph, V=(Nv,Av,Iv), for a program, P. Suppose that S=(Ns,As,Is) is an SPG generated from graphs G and V.

(1) Instantiation operation

The operation which takes the instance value, "j", and appends it to the node, Ff(j), in S, is called the instantiation operation. Suppose $g \in Ns$ and $j \in Fs(g)$. The instantiation of "g" is then defined as

$$Fd(g\{j\}) \equiv Fd(g) - \{j\} \text{ and } Fs(g\{j\}) \equiv Fs(g) - \{j\} .$$

The node g{j} is called an instantiated node. If i Fs(g{j}) and "i" is appended to node "g{j}", it is notated as "g{i,j}". Other values can be appended in the same way.

As examples, consider the SPG on the left in Fig. 3(a) and assume $(g,3,1) \in Fs(g)$. The graph obtained by appending "(g,3,1)" to node "g" is then shown on the right.

If Fs(f) is not null, node "f" is said to be instantiable.

(2) Expansion operation

Assume the function "f" is recursively defined in Program P. Then, multiple nodes, appear which have "f" as a function part, can be appeared in a path of graph V from node "Iv" to any leaf. The set Fs(f) includes only the node nearest to the "Iv" in the path. To make other nodes selectable, the concept of the recursive node is defined as follows.

We first define an auxiliary function, Fm, as

$$Fm(j,f) \equiv \max_{p \in Pj} (Fe(p,f)),$$

where $j \in Nv$ and Pj is a set of paths from node "j" to leaves of V, and Fe(p,f) is the number of instances on path "p" whose function parts are "f". Note that function "Fe" is defined independently of graph S.

For recursive nodes, Fd and Fs are defined as

$$Fd(f.1^*) \equiv Fd(f) \text{ and } Fs(f.1^*) = Fs(f).$$

For any non-negative natural number, m,

$$Fd(f.m) \equiv \{ j \mid j \in Fd(f.m^*) \text{ and } Fm(j,f)=m \},$$

$$Fd(f.m+1^*) \equiv \{ j \mid j \in Fd(f.m^*) \text{ and } Fm(j,f)>m \},$$

$$Fs(f.m) \equiv Fd(f.m),$$

and

$$Fs(f.m+1^*) \equiv \{ j \mid j \in Fs(f.m^*) \text{ and } Fm(j,f)>m \}.$$

Hereafter, a basic node "f" is treated as f.1*. Apparently, $Fd(f.m^*) = Fd(f.m) \cup Fd(f.m+1^*)$ $(m \geq 1)$.

A node f.m* is said to be <u>expansible</u> if $Fd(f.m+1^*)$ $(m \geq 1)$ is not null, and S is said to be <u>expansible</u> if S includes an expansible node. Suppose graph S contains an expansible node f.m*. Then an SPG, T=(Nt,At,It), obtained by expanding the node f.m* is a graph that is computed in the following way:

* Split the node f.m* to f.m and f.m+1*. Draw arcs the same as in graph S except for the case they come to or go from f.m*.
* If an arc comes to/goes out from node f.m* in graph S, then make the arc come to/go from both nodes f.m and f.m+1*. Subsequently, remove an arc in T if it goes from f.m and comes to itself.
* If f.m* = Is then "It" is f.m+1* else "It" is the same as "Is".

The expansion of node f.m* is shown as an example in Fig. 3(b).

(3) Folding/Unfolding operation

The folding operation corresponds to configuring a

hypothesis wherein a function is defined correctly and to removing a set of instances of the function from the search space. The unfolding operation corresponds to the retraction of the hypothesis and to the return of the instances, which are removed by the preceding folding operation, to the search space.

Assuming that $g \in Ns$ and $g \neq Is$. The following operations can then be performed when the folding operation is applied to "g".

* For each node "f" that has a call-arc to node "g", change the node name to "f[g]".
* Remove all call-arcs that go out from node "g".
* Suppose X and Y to be node sets that have parameter-arcs from/to node g, respectively. Then connect a parameter-arc from all nodes in X to all nodes in Y.
* Remove node "g" and the related call/parameter-arcs.

A node like "f[g]" is called a folded node, for which functions "Fd" and "Fs" are defined as

$$Fd(f[g]) \equiv Fd(f) \cup Fd(g),$$

and

$$Fs(f[g]) \equiv Fs(f).$$

Folded node "f[g]" is conversely changed into two identical nodes "f" and "g" when the unfolding operation is applied to it. An example of the folding/unfolding operation is shown in Fig. 3(c).

3.3 Selection Operations for an SPG

Assume a function dependency graph, G, and an instance value dependency graph, V=(Nv,Av,Iv), for a program, P. Suppose that S=(Ns,As,Is) is an SPG generated from graphs G and V. The following three operations are defined to obtain a subgraph of S (in other words, to remove some nodes of S). For each node f in

the resultant subgraph, the values, Fd(f) and Fs(f), remain unchanged.

(1) Inner-removal operation

Let $p \in Ns$ and $p \neq Is$. Then, the graph, obtained by applying inner-removal operation to node "p", is a maximal subgraph which is obtained by removing call-subordination graph of "p" from S and whose initial node is "Is".

(2) Outer-removal operation

Let $p \in Ns$, and $p \neq Is$. Then, the graph, obtained by applying outer-removal operation to node "p", is a call-successor graph of "p" whose initial node is "p".

(3) Non-input removal-operation

Let $p \in Ns$, L be a set of nodes in a call-predecessor graph of "p", M be a set of nodes in a parameter-predecessor graph of each node in the node set L, and N be a set of all nodes in call-successor graph of each node in M. Union of L and N is named an input decision node set of "p". The graph, obtained by applying non-input-removal operation to node "p", is then a maximal subgraph, whose node set is an input decision node set of "p", and whose initial node is "Is".

Figure 4 shows examples of each of these selection operations. The shaded areas represent the graphs removed by each selection operation.

4. Strategic Bug Location Method

4.1 Overview

The strategic bug location method (SBL) consists of two parts, i.e., a program analyzer and a bug locator. The program analyzer statically traces a source program in terms of function

dependencies, and transforms the execution history into a Strategically Projected Graph (SPG), which reflects the structure of the source program. Furthermore, it reduces the SPG to enable the bug locator to more easily manipulate the global structure of the program. The program analyzer replaces a subgraph, which consists of nodes for strongly related functions, with one node, that is, all instances of functions strongly related to each other are projected into a single node in the SPG. The program analyzer iteratively reduces SPG as shown in Fig. 5. As a result, the SPG becomes a single node which represents a hierarchical function structure of the source program.

The bug locator also iteratively "reduces" the SPG through the analysis of the query-and-answer sequence to the programmer. This SPG reduction is achieved from two points of view: unfolding/expanding/detailing the SPG to decrease the number of instances projected into the same node in the SPG, and folding/selecting the SPG to decrease the number of nodes in the SPG. The bug locator selectively applies the transformation/selection operations to the SPG according to the strategies derived from the properties of bug propagation in the functional programs. The iterations of the transformation/selection of the SPG result in the SPG becoming a node which contains the source of a bug as was shown in Fig. 5.

4.2 Program Structure Analysis Using Intervals

When the number of functions in a program is large, it is important to examine the SPG hierarchically. On the other hand, the function dependency graph, which determines the construct of an initial SPG, does not reflect the program's hierarchical structure. In this paper, we propose a procedure that transforms

the SPG into a single node reflecting a hierarchical program structure before reducing it through bug location strategies. This procedure is based upon the use of intervals [15] defined as follows.

Let G0 = (Ng0,Ag0,Ig0) be a directed graph. Given a node, h Ng0, called the header node, an interval, I(h), is the maximal, single entry subgraph in which h is the only entry node and in which all closed paths in I(h) contain the node h. An interval can be expressed in terms of the nodes in it: I(h)=(n1,n2,n3,...,nm), where h$\in$I(h). For this interval, Fd(I(h)) and Fs(I(h)) are defined as $Fd(I(h))=\bigcup_{i=1}^{m} Fd(ni)$ and Fs(I(h))=Fs(h).

By selecting the proper set of header nodes, {h1,h2,...}, in SPG and finding intervals with the algorithm in the literature [15], we can partition the SPG into a unique set of disjointed intervals, and can obtain a new SPG, S1=(Ns1,As1,Is1), where Is1=I(Is0) and Ns1={I(h1),I(h2),...}.

The program analyzer successively applies the following procedure to SPG Si until it becomes a single node. Here, Si=(Nsi,Asi,Isi) is the SPG which is derived from the i-th application of the procedure and S0 is the SPG constructed using the function dependency graph.

(1) Find intervals in Si-1 and make each of them into one node in Si.

(2) Make each interval exit arc into one arc in Si.

(3) Make Isi=I(Isi-1).

This operation is called <u>abstraction</u>, and <u>detail</u> operation gives the interval from the node.

Figure 6 illustrates an SPG sequence produced through the successive applications of the abstraction operation. A node in

SPG Si represents all nodes in an interval in SPG Si-1, and a node in SPG Si-1 represents all nodes in an interval in SPG Si-2, and so on. Eventually, the node produced by the program analyzer reflects the hierarchical structure of the source program.

Although not all graphs can be reduced to a single node by successive transformation, methods for "splitting" (copying) certain nodes in such an irreducible graph produce an equivalent graph reducible to a single node.[15]

4.3 Strategic Bug Location Method

An SPG, S, has the three particular properties relative to bugs.

[Property P1] When node "f" is not instantiable, an erroneous function exists outside the call-subordination graph of "f" under the condition that all instantiated values of "f" are true.

[Property P2] When node "f" has a false instance value, an erroneous function exists in the call-predecessor graph for "f".

[Property P3] When node "f" has an undefined instance value, an erroneous function exists in the input-decision node set of "f".

We hypothesize the relationship between bugs and program structures as follows.

[Hypothesis H1] If a function "f" has at least one true instance value, there is a high probability that "f" is correct.

[Hypothesis H2] Suppose that both "p" and "q" are instance values produced by executing a function "f" (i.e., Ff(p)=Ff(q)=f) and that "p" is true. If the same set of expressions in "f" are executed to produce "q" as those executed to produce "p", there is a high probability that "q" is also true.

[Hypothesis H3] The more functions exist in a program, the more bugs tend to be in it.

[Hypothesis H4] If a function, which is one of the mutual recursive functions, has a bug, there is a high probability that an erroneous result is propagated to all instance values for such functions.

The following strategies are derived from the above hypotheses.

[Strategy S1] When an instance value of a function "f" is found to be true, all instance values of "f" are assumed to be true according to the hypothesis H1, and all instance values, that are in call-subordination graph for "f", are temporally removed from EIVS by folding "f". However, a contradictory assertion, which tells all functions are true, is induced if all functions are folded during the bug location. Therefore, the hypothesis is retracted and the folded graph is recovered by unfolding "f".

[Strategy S2] If the SPG is reduced to one expansible node "f", it is expanded so that the instance values nearest to the leaf can be included in the EIVS. This strategy specifies that the termination part of "f", followed by the recursive part of "f", is checked initially.

[Strategy S3] A node "f" in SPG for instantiation is selected as follows. Suppose that T(f), F(f) and U(f) are the numbers of nodes derived by applying S1, P2 and P3 to SPG in cases that the instance value of "f" is true, false and undefined, respectively. Select node "f" such that the maximum values of T(f), F(f) and U(f) are the smallest in those "f's" which minimize (T(f)+F(f)+U(f)). Assuming that all probabilities, where the instance value of "f" is true, false and undefined, are equal (i.e. 1/3). Then, the expected number of nodes in S, which is obtained by applying the selection operations to "f", is minimal.

[Strategy S4] The SPG is checked by using the unit of interval.

An interval is not to be detailed while other nodes exist in the SPG. When an interval I(h) is to be checked, the instance value j is checked where j Fs(I(h))=Fs(h). In checking I(h), I(h) is true if the header h is true.

Based on the above strategies and their properties, the bug locator iteratively applies operations to the SPG until termination A9 as shown in Fig. 7. In this figure, |S|" is the number of nodes in SPG S, and labels A1 to A9 are identifiers for the operations to be applied. The names of the properties and strategies in the parentheses tagged to the operators give the reasons for applying the operation. The source of bug is detected as follows.

When $|S| \geq 2$, the strategy S3 selects a node "f" from S and tries to instantiate it. Here, if "f" is not instantiable, the inner-removal operation is applied to "f" according to the property P1. On the other hand, if "f" is instantiable, any node "v=(f,x,y)" in EIVS of "f", i.e. Fs(f), is appended to "f" through the instantiation of "f", and the question concerning "v" (i.e. whether f(x) is true, false or undefined.) is generated to the programmer. The answer to the question, which tells whether "v" is true, false or undefined, activates the operation for "f", i.e. , either the folding, outer-removal or non-input-removal operation is applied according to the answer obtained. When |S|=1, S is expanded if possible. When |S|=1 and S is not expansible, S is unfolded if it consists of a folded node. If S is abstracted, S is then detailed. Otherwise, S consists of a single node which contains the source of the bug.

4.4 Examples

We will focus here on how to locate a bug using SBL.

(1) Calculating Fibonacci Numbers

The following function "f" is for calculating Fibonacci numbers.

[Example 3]:

f(n)={if n≤2 then 1 else f(n-1)+f(n-2)}.

We consider a buggy function "fr" of "f", which has a bug in its recursive part, as follows.

[Example 4]:

fr(n)={if n≤2 then 1 else fr(n-1)*fr(n-2)}.

Suppose that we want the tenth Fibonacci number. We find that "fr" is incorrect, since fr(10) returns to 1. The initial SPG is a single node which consists of "fr". SBL iteratively applies the transformation/selection operations to the SPG and eventually points out that the instance value fr(3)=1 is the source of the bug as shown in Fig. 8. As a result, we find that there is a bug in the recursive part of "fr", i.e., in the expressions which consist of n≤2 and fr(n-1)*fr(n-2).

Notice that the sequence of the operations is independent of n, i.e., the three queries lead to the location of the bug even if a considerable execution history is generated by large n. Furthermore, it can be said that SBL is suitable for debugging a functional program using a parallel computer like a data flow machine, since a programmer is only required to answer the query which is independent of the complex behavior in parallel execution.

(2) Sorting a List using Mergesort Algorithm

A simple sorting program using a mergesort algorithm consists of three functions: "split", "merge" and "mergesort" as follows.

[Example 5]

```
split(x)={if null(x) then (nil,nil)
          elsif null(cdr(x)) then (x,nil)
           else {(u,v)=split(cddr(x));
                 return(cons(car(x),u),cons(cadr(x),u))}},

merge(x,y)={if null(x) then y
            elsif null(y) then x
             elsif car(x)<car(y)
              then cons(car(x),merge(cdr(x),y))},

mergesort(x)={if null(cdr(x)) then x
              else {(u,v)=split(x);
                    return(merge(mergesort(u),mergesort(v)))}.
```

These functions are for spliting a list into two lists, for merging two sorted lists into a sorted list and for sorting a list by calling both "split" and "mergesort", and by calling itself. The following function is a buggy version of the above "mergesort" function.

[Example 6]

```
mergesort(x)={if null(cdr(x)) then nil
              else {(u,v)=split(x);
                    return(merge(mergesort(u),mergesort(v)))}
```

The function dependency graph of the above program (i.e., the initial SPG) is a single interval which consists of three nodes as shown in Fig. 9. The program analyzer creates a node "f" through the abstraction of the interval. The bug locator iteratively transforms/selects "f" as shown in the figure, when the instance value dependency graph is provided by executing mergesort((8 2 1 6 4 7 5 3)). As a result, we find that a bug exists in the termination part of "mergesort", i.e., in the expressions of null(cdr(x)) and nil.

5. Conclusion

This paper has presented a new bug location method for functional programs called Strategic Bug Location (SBL). SBL

produces a search graph, named the Strategically Projected Graph (SPG), using both the dynamic execution history and static structure of the program. The method iteratively applies such operations as transformation and selection to the SPG. There are three principal features of SBL. First, SBL is suitable for debugging a functional program using a parallel computer like a data flow machine, since a programmer is only required to answer the query which is independent of the complex behavior present in parallel execution. Second, bug location strategies are incorporated into the transformation mechanisms to accelerate the reduction of the search graph by discarding redundant questions. Third, SBL uses a hierarchical search graph to facilitate grasping the abstract structure of a program during bug location and to facilitate following the query sequence.

Considerable work remain to clarify the effectiveness of SBL. The present experimental debugging system for data flow programs [12] will be extended to enable practical functional program debugging through SBL. Furthermore, efforts will be made to compare SBL with other bug location methods in terms of the number of queries, CPU time and the amount of memory required to debug such practical functional programs.

[Acknowledgements]

The authors would like to thank Dr. Makoto Amamiya, head of the Second Section of Information Science Department at NTT Basic Research Laboratories, for his guidance and encouragement. They also wish to thank the members of the data flow machine research group in the Second Section for their helpful discussion.

References

[1] Darlington,J., Henderson,P. and Turner,D.A. (eds.) : "Functional Programming and its Application," Cambridge University Press, (1982).

[2] Turner,D.A. : "A New Implementation Technique for Applicative Languages," Software Practice and Experience, Vol. 9, pp.31-49 (1979).

[3] Keller,R.M. : "FEL (Function-Equation Languages) Programmer's Guide," AMPS Technical Memorandum No. 7, University of Utah, (April 1982).

[4] Amamiya,M, Hasegawa,R. and Ono,S. : "Valid, A High-Level Functional Programming Language for Data Flow Machine," Rev. ECL Vol. 32, No. 5, pp. 793-802, NTT (1984).

[5] Backus,J. : "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," Comm. ACM, Vol. 21, No. 8, pp. 613-641 (1978).

[6] Arvind and Kathail,V. : "A Multiple Processor Dataflow Machine That Supports Generalized Procedures," Proc. the 8th Annual Symposium on Computer Architecture, pp. 291-302 (1981).

[7] Gurd,J. and Watson,I. : "Data Driven System for High-speed Parallel Computing (1 & 2)," Computer Design, Vol. 9, No. 6 & 7, June & July, pp. 91-100 & 97-106 (1980).

[8] Takahashi,N. and Amamiya,M. : "A Data Flow Processor Array System : Design and Analysis," Proc. the 10th Annual Symposium on Computer Architecture, pp. 243-250 (1983).

[9] Amamiya,M., Hasegawa,R., Nakamura,O. and Mikami,H. : "A List-Processing-Oriented Data Flow Machine Architecture,"

Proc. 1982 National Computer Conference, AFIPS, pp. 143-151 (1982).

[10] Keller,R.M., Lindstrom,G. and Patil,S. : "A Loosely Coupled Applicative Multiprocessing System," Proc. the 1979 National Computer Conference, AFIPS, Vol. 49, pp. 613-622 (1979).

[11] Darlington,J. and Reeve,M. : "ALICE : A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages," Proc. the 1981 ACM/MIT Conference on Functional Programming Language and Computer Architecture, pp. 65-75 (1981).

[12] Takahashi,N., Ono,S. and Amamiya,M. : "Parallel-Processing-Oriented Algorithmic Bug Location Method for Functional Programming Languages," (in Japanese) Proc. Meeting on Software Foundation, 11-4 (1984).

[13] Takahashi,N., Ono,S. and Amamiya,M. : "A Bug Location Method for Functional Programs Using Function Dependency Graph Transformation," (in Japanese) Proc. Meeting on Software Foundation, 12-3 (1985).

[14] Shapiro,E.Y. : "Algorithmic Program Debugging," MIT Press (1983).

[15] Allen,F.E. and Cocke,J. : "A Program Data Flow Analysis Procedure," Comm. ACM, Vol. 19, No. 3, pp. 137-147 (1976).

[16] Johnson,M.S. : "A Software Debugging Glossary," SIGPLAN Notices, 17, 2, pp. 53-70 (1982).

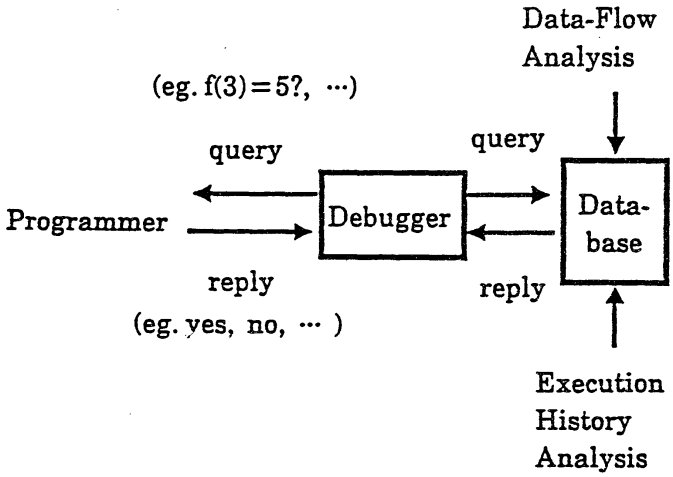


Fig. 1 Program Diagnosis System

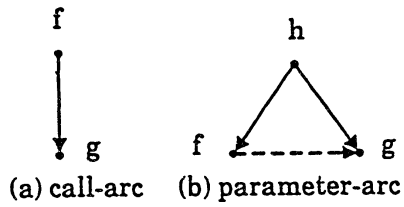
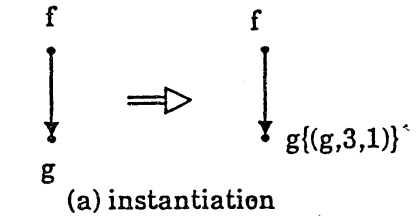
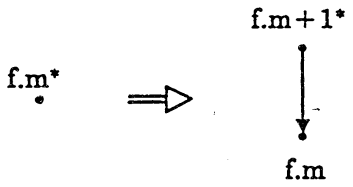


Fig. 2 Function Dependency Graph





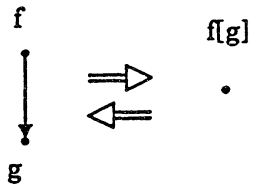


Fig. 3 Transformation Operators for Strategically Projected Graph

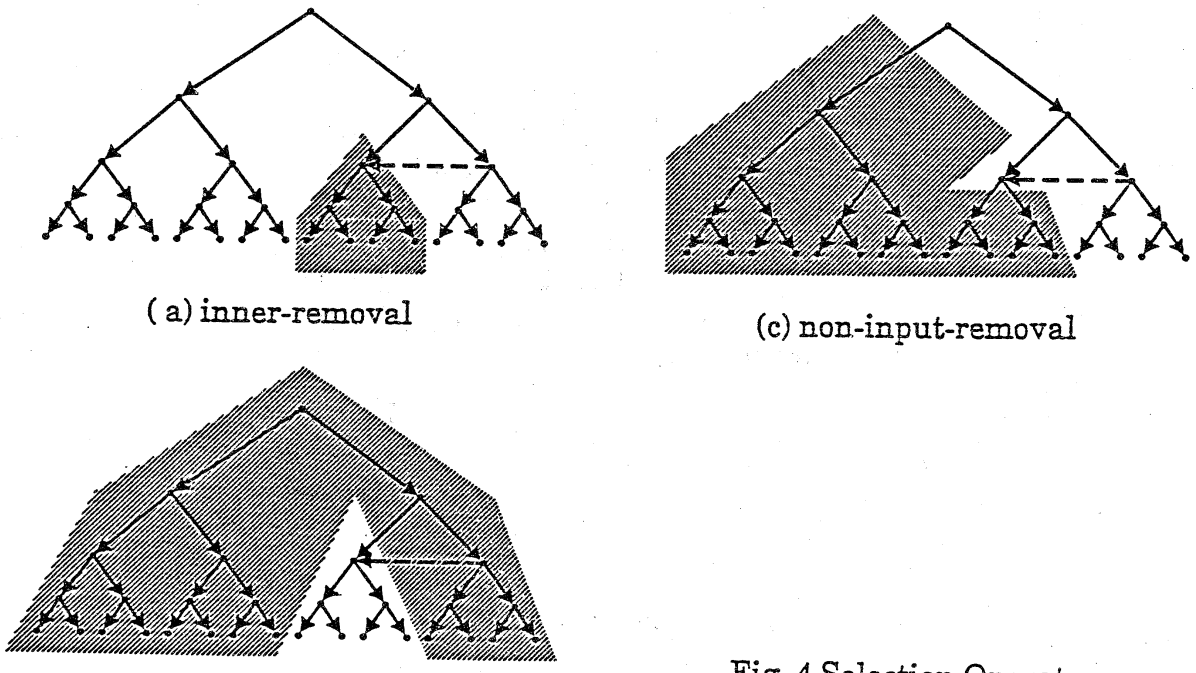


Fig. 4 Selection Operators
for Strategically Projected Graph

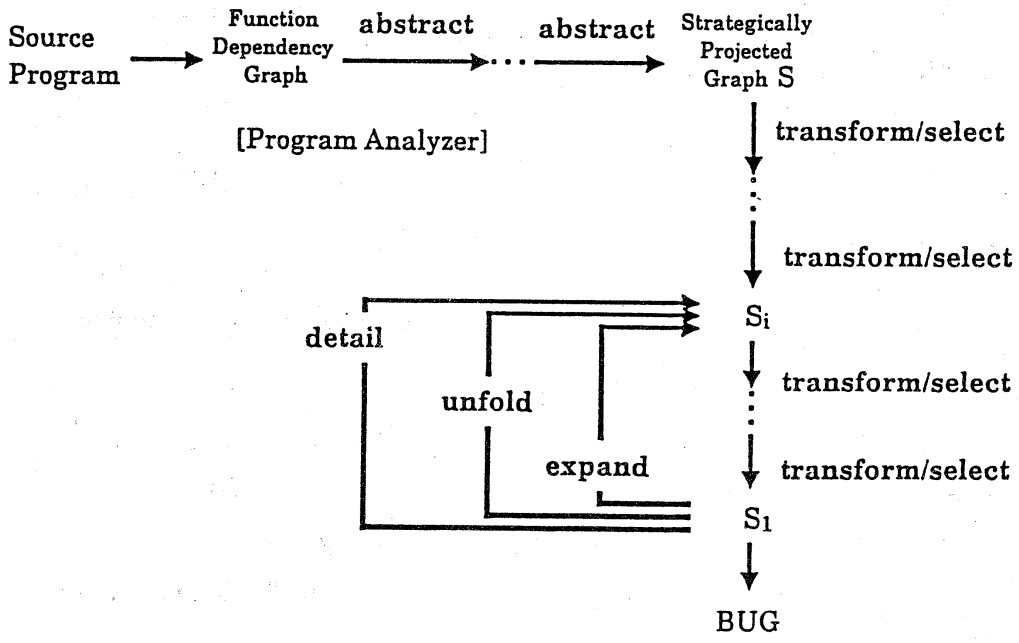


Fig. 5 Outline of Strategic Bug Location

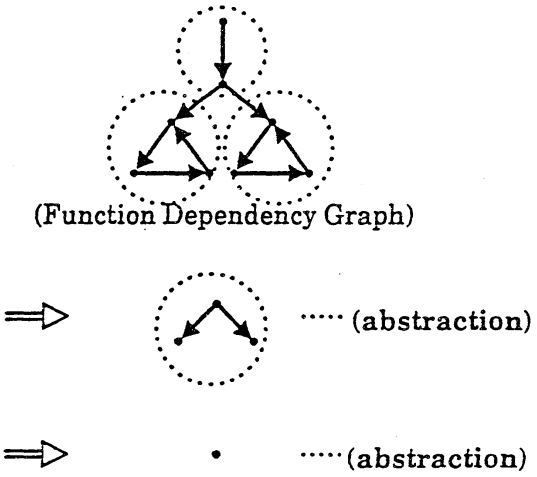


Fig. 6 Abstraction of Intervals

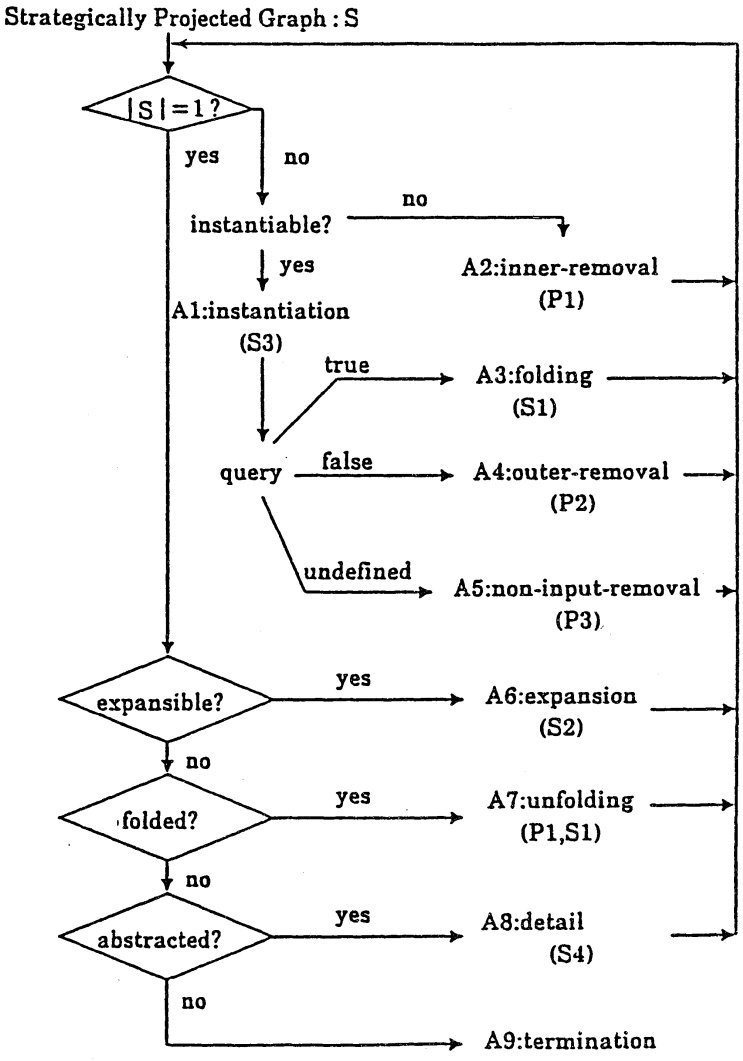


Fig. 7 The SBL Method

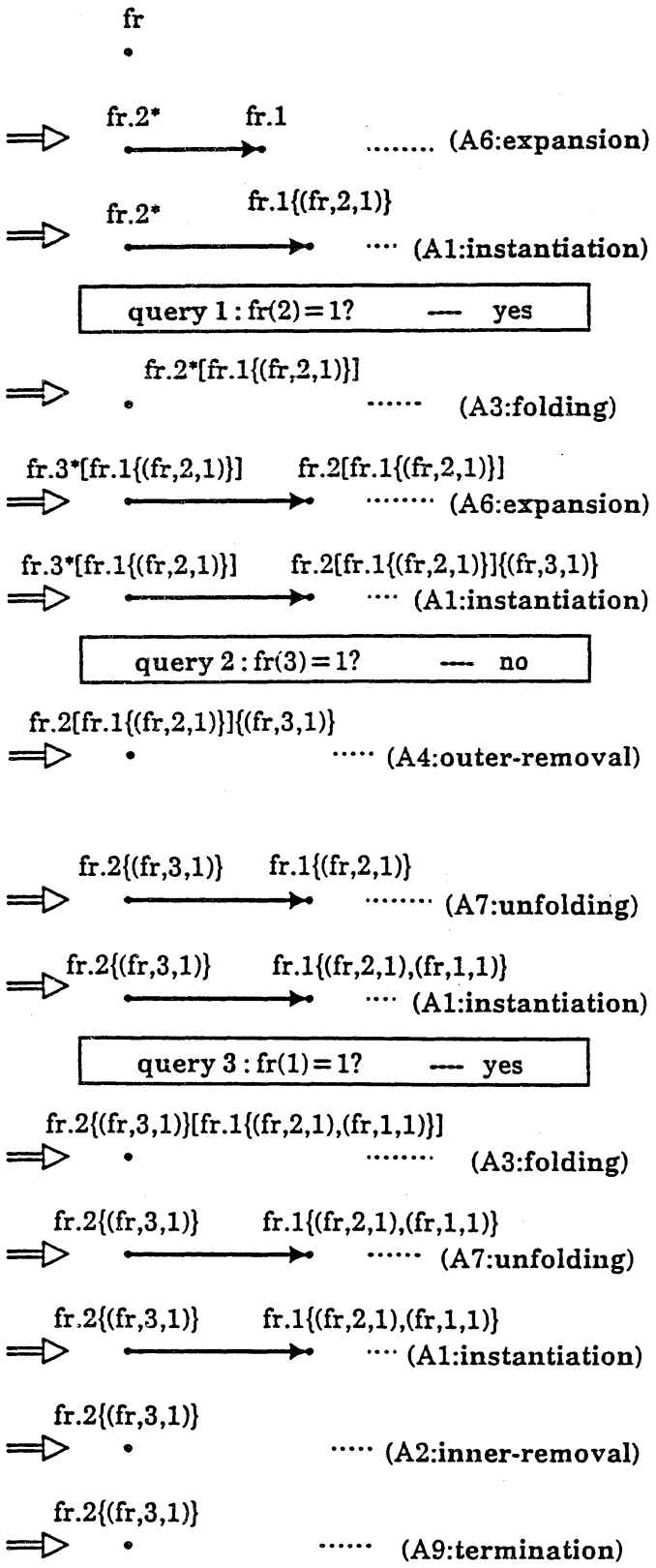


Fig. 8. Bug Location of Erroneous Fibonacci Program Using the SBL Method

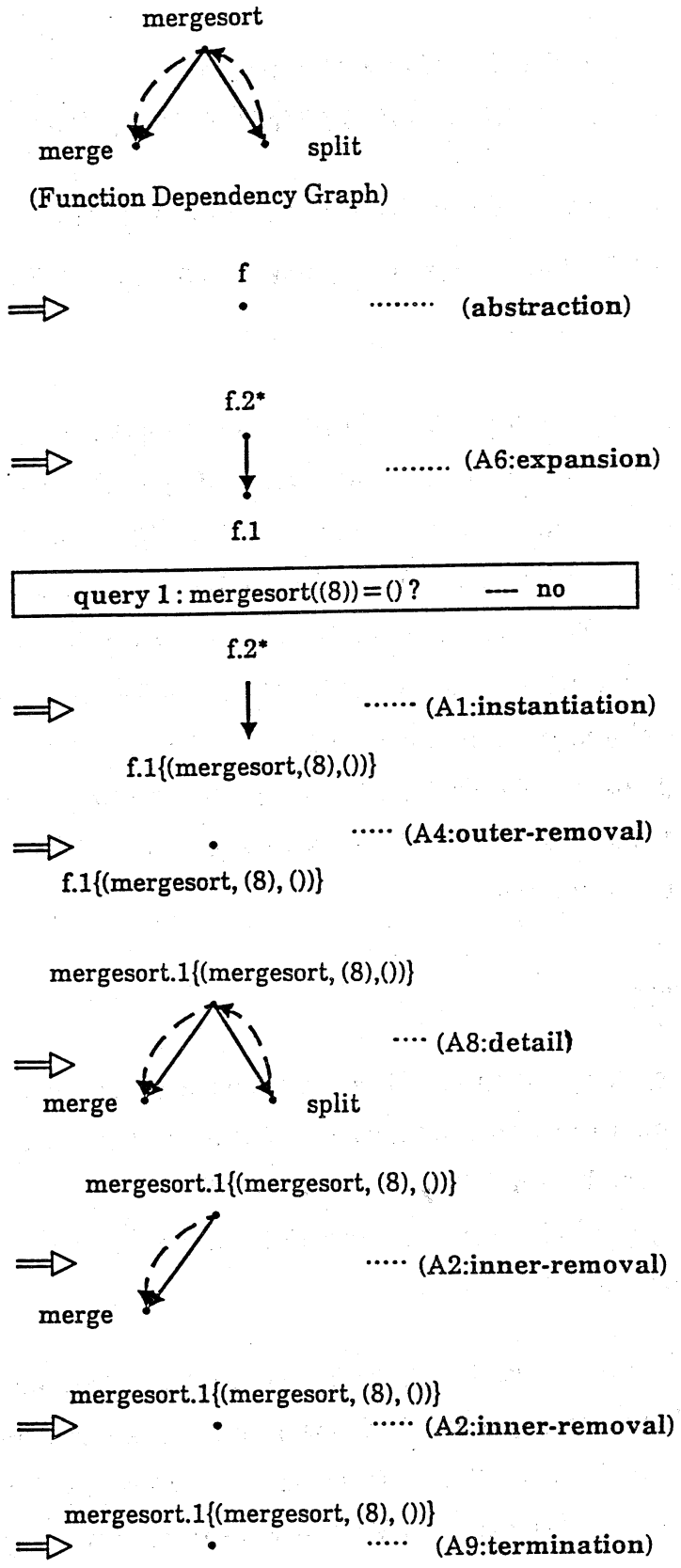


Fig. 9. Bug Location of Erroneous Mergesort Program Using the SBL Method