ソフトウェア基礎論 30−4
(1989. 9. 8)

Design and Implementation of a Declarative Debugging System for Functional Programs

関数型プログラムの宣言的デバッグシステムの設計と実現

Naohisa TAKAHASHI and Satoshi ONO
NTT Software Laboratories
3-9-11 Midoricho, Musashino-shi, Tokyo 180 Japan

高橋　直久　　小野　諭
(NTTソフトウェア研究所)

Atstract The declarative debugging system for functional programs, DDS, presented in this paper is an interactive system which has an elaborated diagnoser and a new partial execution mechanism called a filter. The advantages of the diagnoser in DDS are as follows: The programmers are required only to find some available assertions on input-output relations of functions among the intermediate results of program diagnosis issued by the diagnoser. In addition, the diagnoser, where the heuristics on debugging of functional programs are incorporated, examines the computation tree in conjunction with the results of the static analysis of the program, resulting in both requiring the programmers to give less assertions and allowing us to use the filter so as not to waste a great deal of space and time.

The filter controls the execution of the program being debugged and saves a portion of the computation tree in a small, bounded buffer efficiently. To avoid computation and recomputation unnecessary for diagnosis, the filter "freezes" a portion of the computation, and makes it accessible as a special value to the computation requiring it. This freezing mechanism has advantages for constructing computation trees which include such troublesome expressions as will produce a lot of intermediate results, invoke functions infinitely, suspend execution by incurring run-time errors like type-error-operations.

The proposed approaches allow us to construct a unified framework for program diagnosis, in that DDS can diagnose not only large programs terminating with incorrect values but also non-terminating programs. A prototype of DDS, which has been implemented using Common Lisp on VAX/VMS, demonstrates the usefulness of declarative debugging of functional programs.

あらまし　宣言的デバッグシステムDDS(Declarative Debugging System)では、プログラマはプログラムに期待する実行結果や途中結果を与え(宣言的に定義し)、それに基づいてシステムがプログラムのテキストと実行履歴を解析しバグ探索空間を絞り込む。本稿では、DDSの設計課題を考察し、実現上重要な3つの機構、すなわち、バグ発生源の判定機構、プログラムの部分実行機構、プログラムの診断機構について議論する。被デバッグプログラムに対して診断に不要な部分の実行を抑止する"計算の凍結"機能が重要であることを示し、その実現法とデバッグへの適用法を明らかにする。さらに、プログラム構造の静的な解析とバグ検出のヒューリスティクスを用いてプログラマの宣言と部分実行結果を解析する手法を提示する。最後に、VAX/VMS上に作成したDDSのプロトタイプを用いたデバッグ例を示し、関数型プログラムのデバッグにDDSが有効であることを示す。

1 Introduction

Functional programming languages have many excellent features which facilitate the writing of clear, concise programs, and the understanding and verification of these programs using clean mathematical semantics.[1,2] Furthermore, freedom from side-effects in functional programs makes it possible to adopt algorithmic program debuggers[3−8] whose diagnosers can isolate an error mechanically when given a program and an input on which the program behaves incorrectly. The diagnosers examine the computation tree of the program to diagnose the error by using programmer-provided answers to queries concerning the correctness of input-output relations of functions

(i.e., the correctness of the intermediate results obtained through the execution of the program).

These diagnosers are expected to free programmers from the problems which are inevitable with commonly used debugging methods,[9] namely the having to find when and where bugs occurred and how they should be fixed. However, most diagnosers have problems concerning efficiency and man-machine interactions, because the major research objectives of these diagnosers were not to implement a practical debugger, but to construct a logical framework for algorithmic program debugging. The declarative debugging system for functional programs, DDS, presented in this paper is an interactive system which has an elaborated diagnoser and a new partial execution mechanism called a filter. The advantages of the diagnoser in DDS are as follows.

- The diagnoser does not assume programmers to be able to find answers to all questions being issued. Instead, programmers are required only to find some available assertions on input-output relations of functions among the intermediate results of program diagnosis issued by the diagnoser.
- The diagnoser, where the heuristics on debugging of functional programs are incorporated, examines the computation tree in conjunction with the results of the static analysis of the program to make error isolation efficient.

The filter controls the execution of the program being debugged and saves a portion of the computation tree in a small, bounded buffer efficiently. From the viewpoint of implementation techniques, the filter is an elaboration of optimized evaluation techniques like tabulation[10] and applicative caching,[11] in that the diagnoser can specify functions whose values are to be retained and also specify the portions of a program to be executed (or not executed). In order to avoid computation and recomputation that is unnecessary for diagnosis, the filter "freezes" a portion of the computation, and makes it accessible as a special value to the computation requiring it. This freezing mechanism has advantages for constructing computation trees which include such troublesome expressions as will produce a lot of intermediate results, invoke functions infinitely, suspend program execution by incurring run-time errors like type-error-operations.

The proposed approaches allow us to construct a unified framework for program diagnosis, in that DDS can diagnose not only large programs terminating with incorrect values but also non-terminating programs. This paper discusses three essential mechanisms for implementing DDS; identification of the source of a bug, partial execution of a program being debugged, and program diagnosis. In addition, a prototype of DDS, which has been implemented using Common Lisp on VAX/VMS, demonstrates the usefulness of declarative debugging of functional programs.

2 Overview of declarative program debugging

A prototype of the declarative debugging system, named DDS, is now operational on Common Lisp under VAX/VMS. A simple debugging session of the prototype provides an overview of declarative program debugging.

A programmer writes a program in a pure functional subset of Common Lisp[12] and debugs it using a screen editor named VEX (Visual Evaluation eXecutive). Figure 1 shows terminal screen changes occurring during debugging of a simple program named "mergesort program." All symbols inside the dotted line were typed by a programmer, while other symbols were displayed by DDS in Fig. 1 and the figures described below.

The mergesort program consists of three functions named *mergesort*, *xmerge* and *split*. The $mergesort(x)$ is expected to sort the elements of the list x, while $xmerge(x, y)$ is expected to merge two sorted lists x and y. The $split(x)$ is expected to split the list x into two sublists. Figure 1(a) illustrates an initial screen which shows the definitions of the three functions and an expression for initiating the diagnoser. The expression, "(*diagnoser* '(*mergesort* (3 7 2 5 6 1 8 4)," tells the diagnoser to start locating bugs.

Evaluations of both function definitions and expression lead to the next screen, shown in Fig. 1(b). This screen displays the first query and all instances of the function *split* which are selected by the diagnoser to be examined. The programmer will enter "*yes*", which asserts that the expression (*split* '(3 7 2 5 6 1 8 4)) is a pair of lists (3 2 6 8) and (7 5 1 4). As a result, both the second query and all instances of *xmerge* are displayed with the reasons the diagnoser selected them to be displayed as shown in Fig. 1(c). The reasons mean the following issues. The diagnoser deduced, from the programmer's assertion, a hypothesis that the function *split* has no bug. As a result, the function *split* is treated as a part of the called function (FOLDING: SPLIT) and will not be examined, as if it were a primitive function, until the hypothesis is disproved. The diagnoser selected the function *xmerge* to examine and displayed its instances.

The programmer will enter "(*false* 14 18)" to assert that the expressions "(*xmerge* '(2 3 6 8) '(1 4)) = (1 2 3 4)" and "(*xmerge* '(6 8) nil) = NIL" are not true. The diagnoser concludes that a bug exists in the function *xmerge* and that the bug appears when the *xmerge* is applied to '(6 8) and *nil*. Consequently, it displays the buggy definition of the function *xmerge* and the expressions which are executed in the instance as shown in Fig. 1(d). The programmer will fix the bug by changing the expression "(if (null y) y ...)" into "(if (null y) x ...)" in the definition of *xmerge* through the VEX window.

```
(defun mergesort (x)
  (if (null (cdr x)) x
      (multiple-value-bind (u v) (split x)
         (xmerge (mergesort u) (mergesort v)))))

(defun xmerge (x y)
  (if (null x) y
      (if (null y) y
          (if (< (car x) (car y)) (cons (car x) (xmerge (cdr x) y))
              (cons (car y) (xmerge x (cdr y)))))))

(defun split (x)
  (if (null x) (values nil nil)
      (if (null (cdr x)) (values x nil)
          (multiple-value-bind (u v) (split (cddr x))
             (values (cons (car x) u) (cons (cadr x) v))))))

(diagnoser '(mergesort (3 7 2 5 6 1 8 4)))

———— SHELL evaluator (show) File DEMO.LSP ————
```

(a) Initial screen

```
Query : (SPLIT '(3 7 2 5 6 1 8 4))=(3 2 6 8); (7 5 1 4); True?

0 : (MERGESORT '(3 7 2 5 6 1 8 4))=(1 2 3 4);
 13 : (SPLIT '(5 4))=(5); (4);
  5 : (SPLIT nil)=NIL; NIL;
 12 : (SPLIT '(7 1))=(7); (1);
  5*: (SPLIT nil)=NIL; NIL;
 10 : (SPLIT '(7 5 1 4))=(7 1); (5 4);
  11 : (SPLIT '(1 4))=(1); (4);
   5*: (SPLIT nil)=NIL; NIL;
 9 : (SPLIT '(2 8))=(2); (8);
  5*: (SPLIT nil)=NIL; NIL;
 8 : (SPLIT '(3 6))=(3); (6);
  5*: (SPLIT nil)=NIL; NIL;
 6 : (SPLIT '(3 2 6 8))=(3 6); (2 8);
  7 : (SPLIT '(6 8))=(6); (8);
   5*: (SPLIT nil)=NIL; NIL;
 1 : (SPLIT '(3 7 2 5 6 1 8 4))=(3 2 6 8); (7 5 1 4);
  2 : (SPLIT '(2 5 6 1 8 4))=(2 6 8); (5 1 4);

———— (Information Window) ————
(SPLIT '(3 7 2 5 6 1 8 4))=(3 2 6 8); (7 5 1 4); True?   [yes]
```

(b) Query and answer (1)

```
Query : (XMERGE '(2 3 6 8) '(1 4))=(1 2 3 4); True?

Assertion : (SPLIT '(3 7 2 5 6 1 8 4))=(3 2 6 8); (7 5 1 4);
FOLDING : SPLIT
Examine  : XMERGE
0 : (MERGESORT '(3 7 2 5 6 1 8 4))=(1 2 3 4);
 14 : (XMERGE '(2 3 6 8) '(1 4))=(1 2 3 4);
  15 : (XMERGE '(2 3 6 8) '(4))=(2 3 4);
   16 : (XMERGE '(3 6 8) '(4))=(3 4);
    17 : (XMERGE '(6 8) '(4))=(4);
     18 : (XMERGE '(6 8) nil)=NIL;
 12 : (XMERGE '(1) '(4))=(1 4);
  13 : (XMERGE nil '(4))=(4);
 10 : (XMERGE '(5) '(4))=(4);
  11 : (XMERGE '(5) '(nil))=NIL;
 8  : (XMERGE '(7) '(1))=(1);
  9 : (XMERGE '(7) nil)=NIL;
 5  : (XMERGE '(3 6) '(2 8))=(2 3 6 8);
  6 : (XMERGE '(3 6) '(8))=(3 6 8);

———— (Information Window) ————
(XMERGE '(2 3 6 8) '(1 4))=(1 2 3 4); True?   [(false 14 18)]
```

(c) Query and answer (2)

```
Assertion : (XMERGE '(2 3 6 8) '(1 4)) /=(1 2 3 4);
OUTER-REMOVAL : XMERGE.5
Assertion : (XMERGE '(6 8) nil) /=NIL;
OUTER-REMOVAL : XMERGE.1
INNER-REMOVAL : XMERGE
There is at least one bug in the instance :
(xmerge '(6 8) nil)=nil;
where
(defun xmerge (x y)
  (if (null x)
   y
   (if (null y)
    y
    (if (< (car x) (car y))
     (cons (car x) (xmerge (cdr x) y))
     (cons (car y) (xmerge x (cdr y)))))))
······ evaluated expressions ······
   1  :  (null '(6 8))=nil;
   2  :  (null nil)=t;

———— (Information Window) ————
Result value:    "Buggy instance detected."
```

(d) Final screen

[- - -] Data typed by a programmer
others Data issued by DDS

Figure 1: An Example of Declarative Program Debugging

3 Design issues in declarative debugging systems

The DDS reduces the search space of a bug by the analysis of both the program text and the execution history according to the declarations issued by the programmer as shown in Fig. 2.

Through repetition of the following sequence, the DDS identifies a buggy instance, by which we mean an instance which generates an incorrect value and propagates it to other instances through function applications.

- It executes the program being debugged to collect instances of functions.
- It diagnoses the instances and displays both a query and an intermediate result.
- It analyzes replies given by the programmer to localize the search space for a buggy instance.

The above sequence requires us to efficiently implement three essential mechanisms: identification of buggy instances, partial execution of the program being debugged, and program diagnosis.

(1) Identification mechanism

The mechanism for identification of buggy instances should make it possible to construct a unified framework for program diagnosis, in that DDS can diagnose not only large programs terminating with incorrect values, but also non-terminating programs.

(2) Partial execution mechanism

From a practical point of view, it is important to be able to execute only a portion of the program being debugged and to retain only a small number of instances that are necessary for program diagnosis so as not to waste a great deal of space and time.

(3) Program diagnosis mechanism

It is expected to make program diagnosis efficient by using both the results of static analysis of the program being debugged and heuristics on debugging of recursive functions. However, the problem is how we should incorporate both the static analysis and heuristics with the analysis of instances in a unified framework for program diagnosis.

The design and implementation of the above mechanisms will be described in the following sections.

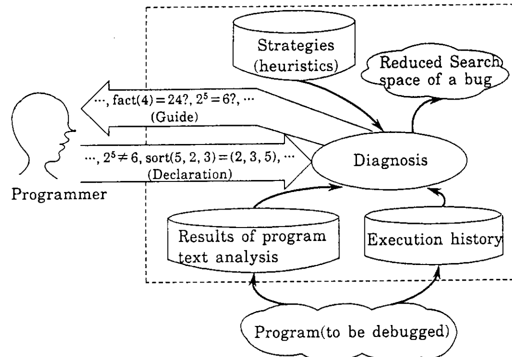

Figure 2: **Declarative Debugging System DDS**

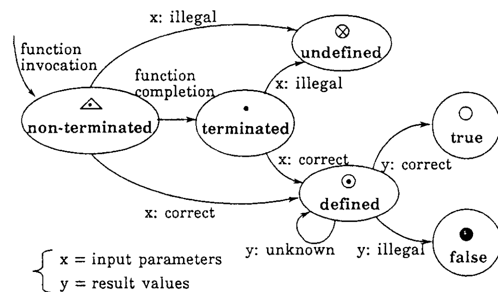

Figure 3: **Transition Diagram of Instance Attributes**

4 Identification mechanism for buggy instances

4.1 Instance attributes and assertions

An instance consists of an instance-value and an attribute. An instance-value is the triplet (f, x, y), in that f, x and y are a function name, an input-argument list and a result-value list, respectively (they are simply called function, argument and result-value below). An attribute is determined according to the properties of x and y as shown in Fig. 3. It is initialized to $non-terminated$ when the function is invoked and changed to $terminated$ when the function is terminated. It is also changed to $undefined$ if DDS found a run-time error such as a primitive operation incurring a type-error. Programmers can assert the correctness of x, the correctness of y or the correctness of the invocation of $f(x)$ in terms of an instance value (f, x, y) so as to change the attribute of the instance.

4.2 Identification of a buggy instance

Freedom from side-effects localizes an error in a program, in the sense that we can isolate a buggy instance within certain instances from the observation of input/output relations of instances. In other words, an error can propagate only through the passing of input/output arguments in function applications. An instance which is invoked with correct arguments is identified to be a buggy instance if and only if one of the following conditions is satisfied.

(1) Although correct results were received from all of the instances being invoked, it returned incorrect result values.

(2) It computed incorrect arguments to be applied to a function by using the correct arguments given by the instance which invoked it and the correct results given by the instances which it invoked.

pattern a pattern b pattern c pattern d

Figure 4: **Bug Patterns in Functional Programs**

(a) Instance-dependency graph created at program termination (b) Detection of a bug pattern

Figure 5: **Identification of a Buggy Instance**

Bug pattern **a** in Fig. 4 illustrates the above case (1) while bug patterns **b** to **d** illustrate case (2). An instance is identified to be a buggy instance if it becomes a root of a tree representing one of four bug patterns.

A simple example is shown in Fig. 5 to describe how a buggy instance is identified. Figure 5(a) illustrates an instance-dependency graph which is provided at the termination of the program execution. Attributes of instances are iteratively changed from *terminated* to new values when the programmer gives assertions. Suppose that the instance-dependency-graph eventually changed into the graph shown in Fig 5(b). The root of a sub-tree being inside the dotted circle is identified to be a buggy instance because the sub-tree represents bug pattern **a**.

5 Partial execution in program diagnosis

5.1 Partial execution control using filter

The referential transparency in functional languages[1] leads to the realization of singular evaluations[11] by saving instance values in a buffer accessible to the computations that need them so as to avoid recomputation. Several techniques of singular evaluation have been proposed so as not to waste a great deal of space and time.[10,11] However, these techniques cannot be adopted for the buffer which instance values are retained for diagnosis since they select functions whose instance values are to be retained from the point of optimization, i.e. the programmer or compile-time optimization specifies which recomputation of a function is to be avoided. The filter presented in this paper is an elaboration of singular evaluation techniques in that the diagnoser can specify functions whose instance values are to be retained and also specify the portions of a program to be executed (or not executed). The filter has the following features.

(1) It retains only instance values whose function has been specified by the diagnoser (this function is called a filtered function) in the execution of a program being debugged.

(2) It also retains instance values that have been specified by the diagnoser not to be invoked in the next execution of the program (this operation is called freezing of an instance value). It "freezes" a portion of the computation, and makes it accessible as a special value to the computation requiring it. Freezing of instance values is done to suppress the invocations of instances which are guaranteed to be unnecessary in the finding of a buggy instance by the diagnoser.

(3) It enables the diagnoser to analyze a program that has a run-time-error or infinite invocations by freezing an instance which will disturb the continuation of the analysis.

(4) It detects an infinite loop by observing a cycle in the dependencies among the instance values being retained. It is efficient because the filter may observe the dependencies only when it finds two instances whose values are equal to each other and the attribute of one of them is *non − terminated*.

fibo(n)=if n≤2 then 1
else fibo(n−1)+fibo(n+2)

n≤8 [Declaration]

[Freezing] n=10
[Freezing] n=9
[Freezing] Detection of a cycle

Figure 6: **Freezing of Unexpected Computaions and Finding of a Cycle**

Figure 6 showes an example of the detection of an infinite loop during the execution of a buggy function *fibo* to compute fibonacchi numbers. The potions of computation of *fibo*(10) and *fibo*(9) are freezed into special values if the programmer asserts that the argument of *fibo* is less than or equal to 8 during the computation of *fibo*(8).

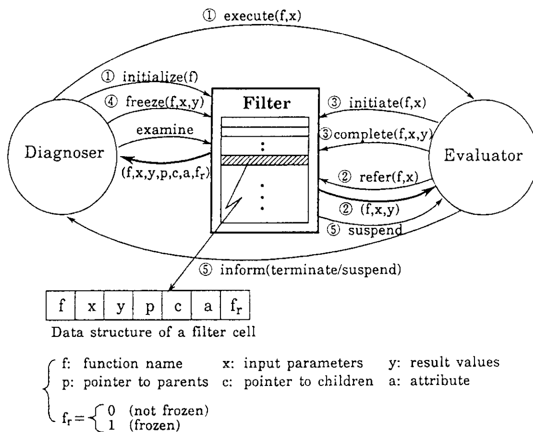

Figure 7: **Filter Mechanism**

5.2 Implementation of filter

The filter includes a set of cells whose fields consist of an instance value (f, x, y), pointers (p, c), an attribute (a) and a flag (f_r), as shown in Fig. 7. The mechanism of the filter is illustrated through the manipulation of the instance value field in a cell as follows.

(1) Initialization: An expression is executed to be diagnosed after a function name selected by the diagnoser is set to the filtered function (the selection of the filtered function will be described in 6.3) and all cells are cleared except for frozen expressions (see (4)).

(2) Computation/reference of an expression: The filter tries to find a cell whose instance value is (f_0, x_0, y_0) when an expression $f_0(x_0)$ appears for computation. It returns y_0 as the result of the computation if it succeeds. Otherwise, it starts a function invocation (see (3)).

(3) Initiation/termination of a function invocation: The filter examines whether f_0 is a filtered function at the function invocation of $f_0(x_0)$. If so, the instance value (f_0, x_0, ω) is written in an empty cell. The symbol ω means an initial value representing an undefined value. The ω will be changed to y_0 which is the result of the computation $f_0(x_0)$ at the termination of the computation. The symbol ω will be returned without the computation of $f_0(x_0)$ independently of f_0 if x_0 includes ω.

(4) Freezing of an expression: If an instance value (f_0, x_0, y_0) or (f_0, x_0, ω) is frozen, it will be referred in (2) without any function invocations.

(5) Suspension of the computation: The DDS suspends the computation of the program being diagnosed if it finds a run-time error, filter overflow, time over or stack overflow. It freezes an instance of a primitive function which will incur an error and sets the attribute of the instance to *undefined* when it finds a run-time error. DDS also freezes an instance incurring a cycle in dependencies among the instances being retained as follows. If the filter finds cells whose instance value is (f_0, x_0, ω) at the initiation of the function application, $f_0(x_0)$, it examines the dependencies among the instances being retained and the function application to be initiated. It decides that the function application will incur infinite function invocations if the dependencies comprise a cycle.

6 Program diagnosis mechanism

6.1 Program diagnosis using filter

In principle, a buggy instance can be identified by using the identification mechanism described in 4.2 if the programmer gives assertions sufficient for finding a bug pattern. DDS allows the programmer to easily give assertions, which effectively reduces the number of assertions required. It diagnoses the program being debugged by using the filter in the following manner.

First, the diagnoser selects a filtered function and initializes the filter as described in 5.2 (1). Next, it asks for the attribute of an instance which is selected from the filter after the program execution (the selections of both a filtered function and an instance to be asked for will be described in 6.3). All instance values being retained in the filter are also issued as the intermediate results of program diagnosis. The programmer will give an answer to the question if he can judge the correctness of the input-output relation of the instance concerned. He may select any instances from the intermediate results and may assert their attributes. Lastly, the diagnoser analyzes the programmer's inputs and reduces the search space for buggy instances by using methods which will be described in 6.2.

The diagnoser identifies a buggy instance when it finds a bug pattern through repetition of the above sequence. This implies that DDS is required to repeatedly execute the program being debugged. The diagnoser freezes instances which will not need to be diagnosed in later repetitions or which will cause suspension of program execution. Identification of those instances will be discussed in 6.2

6.2 Search space reduction based on the properties of bug propagation

The diagnoser analyzes an instance-dependency graph in conjunction with a function-dependency graph representing calling-called relations among functions which are obtained by the static analysis of program text. Suppose that S_F is a search space from which a filtered function is to be selected and that S_I is a search space from which an instance is to be used for a query. An initial S_I is the whole instance-dependency graph representing dependencies among all instances which will be created in program execution while initial S_F is the whole function-dependency graph. Note that the S_I is a virtual graph whose nodes are not actually retained. The diagnoser reduces S_I (or S_F) by replacing it

with its minimal subset, in which the presence of at least one buggy instance (or one function whose instance is a buggy instance) is guaranteed, as follows.

(1)Reduction of S_I

Given the attribute of the instance i, S_I is reduced to its subtree in the following manner.

- The diagnoser will initially execute the instance i in the next partial execution of the program being debugged if the attribute of i is *false*. As a result, a subtree of S_I, whose root is i, is set as a new S_I.
- A subtree of S_I, whose root is i, is removed from S_I by freezing i if the attribute of i is *true* or *undefined*.

(2)Reduction of S_F

Suppose $S_I(f)$ to be a subset of S_I, whose instances are those of the function f. Given the attribute of an instance j, which is an arbitrary element of $S_I(f)$, S_F is reduced to its subgraph in the following manner.

- A subgraph of S_F, whose nodes are reachable from f by using the arcs among nodes (including f), is set as a new S_F if the attribute of j is *false*.
- A set of nodes, which are reachable from the root of S_F via f, is termed an f-dependent set. An f-dependent set is removed from S_F if the attributes of all instances belonging to $S_I(f)$ are *true* or *undefined*.

6.3 Strategic selection methods for filtered functions and queries

It is expected that the following heuristics will lead to the realization of an efficient diagnosis.

- The more functions a portion of a program contains, the higher its diagnosis priority should be.
- If a function f has returned a correct result for an input, it is assumed that f will return correct results for every other input. The higher probability of correctness a function has in the program being debugged, the lower its diagnosis priority should be.
- A recursive function has two groups of instances to be diagnosed individually. These groups can be distinguished simply by observing the behaviour of their function invocations (i.e., some instances invoke the same function recursively while others not).

The filtered function f and the instance q_i to be queried are determined according to the following strategies which are derived from the above heuristics.

- Assuming in turn the various answers the programmer can give, the diagnoser weights the functions in S_F for each individual assumed answer as follows. It computes the number of nodes remaining in S_F after reduction according to the assumed answer. Additionally, it weights the function with the expectation value of the results of the computations for each assumed answers. The function with the largest weight will be selected as f. As a result, the diagnoser selects the function such that expected remaining nodes in S_F becomes least.

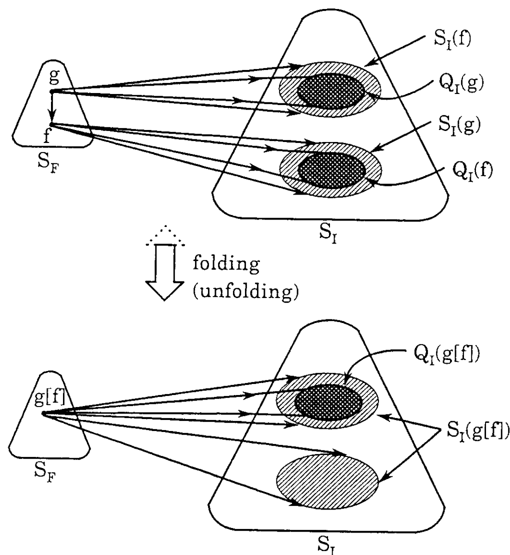

Figure 8: **Folding and Unfolding Operations**

- Finding an instance of the function f, whose attribute is *true* or *undefined*, the diagnoser folds f into the caller of f as if f were a primitive function which would not be diagnosed. The folding of f implies that the diagnoser should remove the f-dependent set from S_F temporarily to focus the other functions because at least one function incurring a buggy instance is assumed to exist. The f will be unfolded to be added to S_F again if the assumption is disproved. Figure 8 illustrates the folding and unfolding operations. In this figure, the $S_I(n)$ is a subset of S_I consisting of instances whose function name is n in S_F, and $Q_I(n)$ is a subset of $S_I(n)$ consisting of instances from which q_i is selected if n becomes a filtered function.
- Suppose p is the instance of f which is the nearest to the root of S_I among the instances being retained in the filter. The node representing f in S_F is expanded into two nodes. These nodes represent a recursive part and a termination part as if they were different functions from each other as illustrated in Fig. 9 if f is a recursive function and p is the root of S_I. The expansion operation implies that the instances of f are selected to be q_i from the leaf of S_I in turn. The p is simply selected to be q_i if f is not a recursive function

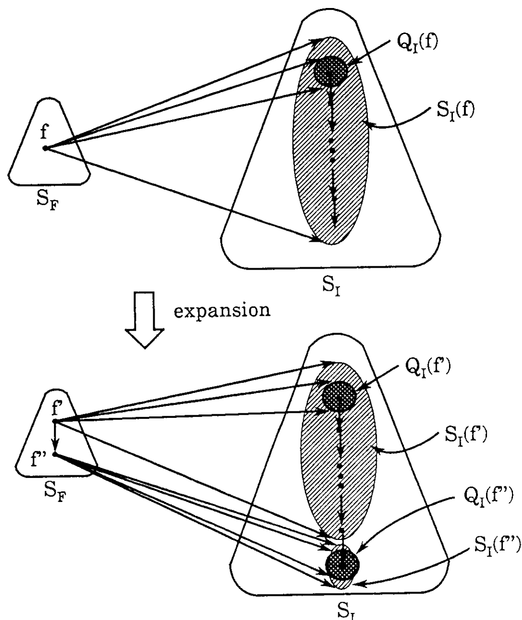

Figure 9: **Expansion Operation**

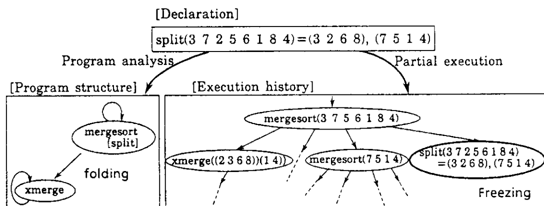

Figure 10: **An Example of Serach Space Reduction by Freezing an Instance and Folding a Function**

or p is not the root of S_I.

Figure 10 illustates the reduction of the search space using the example described in section 2 (See figure 1(b)).

Details of the bug-locating algorithm that is the basis of this diagnosis mechanism are described in [7,13].

7 Examples of debugging with DDS

A simple debugging session of DDS has been described in the Section 2. This section describes other examples of debugging sessions to illustrate the usefulness of the filter in program diagnosis.

7.1 Debugging of a program suspended by a run-time error

Figure 11 illustrates terminal screen changes occurring during debugging of another "mergesort program" which also consists of the functions *split*, *xmerge* and *mergesort*. Figure 11(1) illustrates an initial screen which shows the definitions of the three functions and an expression for initiating a diagnoser. After being initiated, the diagnoser displayed that it froze an expression of (car 'x) because it found a type error in a primitive operation, and that it continued to diagnose the program as shown in Fig. 11(2). The value of (car 'x) was set to ω which was propagated through function applications as described in 5.2 (3). The diagnoser also displayed a query and instances being retained in the filter in order to diagnose *split*. Figure 11(3) illustrates that the diagnoser froze the instance of (*split*, (3 1 5 2), (3 5) (1 2)) and set the attribute of the instance to *true* because it obtained the assertion telling that (*split* (3 1 5 2)) makes the two lists of (3 5) and (1 2). Furthermore, the diagnoser removed *split* from S_F and examined the instances which had relation to the creation of the input argument of (car 'x) to find a buggy pattern. Consequently, a buggy instance was displayed on the terminal screen as shown in Fig. 11(4).

7.2 Debugging of a program incurring infinite invocations

Figure 12 illustrates terminal screen changes occurring during debugging of another "mergesort program" whose function definitions are slightly different from those of the program described in 7.1 as shown in (1) of the figure. Figures (2) and (3) illustrate that the program was suspended by the filter because it did not terminate within a specified execution time. The figures also show the query on the instance which was selected from the filter. The attributes of the instances being asked in Fig. 12(2) and (3) were set to *true* when the programmer answered "yes" to both queries. Freezing both of the instances, the diagnoser executed the program being debugged to diagnose the instances of *mergesort*. As a result, it displayed the occurrence of an infinite loop as shown in Fig. 12(4) because it found a cycle in instances being retained in the filter.

8 Conclusion

This paper presented the design and implementation of a declarative debugging system for functional programs, named DDS. The advantages of DDS are as follows.

- The programmer can diagnose a bug declaratively, in the sense that the programmer need not devise how to find the bug, but need only declare what he intended to compute.

①
```
(defun split (x)    "Split a list X into two sub-lists"
  (cond
    ((null x) (values nil nil))
    ((null (cdr x))(values x nil))
       (T (multiple-value-bind (u v) (split (cddr x))
            (values (cons (car x) u) (cons (cadr x) v))))))
(defun xmerge (x y) "Merge X in Y where X and Y are sorted-lists."
  (cond
    ((null x) y)
    ((null y) x)
    ((< (car x) (car y))
     (cons (car x) (xmerge (cdr x) y)))
    (T (cons (car y) (xmerge 'x (cdr y))))))
(defun mergesort (x)  "Sort a list X"
  (cond
    ((null (cdr x)) x)
    (T (multiple-value-bind (u v) (split x)
          (xmerge (mergesort u) (mergesort v))))))
(diagnoser '(mergesort (3 1 5 2)))
```
SHELL evaluator (show) File DEMO.LISP

⇩

② Query : (SPLIT '(3 1 5 2))=(3 5);(1 2);True?

```
Examine SPG : (SPLIT XMERGE MERGESORT)
Examine : SPLIT
Error while evaluating the function CAR (Level ERROR)
Argument must be a list: X
Selected outermost form is as follows: (CAR 'X)
Freeze (CAR 'X) and Continue evaluation: (MERGESORT '(3 1 5 2))
Examine : SPLIT
0 : (MERGESORT '(3 1 5 2))=**UNDEFINED**;
 99#: (CAR 'X)=**UNDEFINED**;
 5 : (SPLIT '(1 2))=(1); (2);
  3 : (SPLIT nil)=NIL; NIL;
 4 : (SPLIT '(3 5))=(3); (5);
  3*: (SPLIT nil)=NIL; NIL;
 1 : (SPLIT '(3 1 5 2))=(3 5); (1 2);
  2 : (SPLIT '(5 2))=(5); (2);
  3*: (SPLIT nil)=NIL; NIL;
```
(Information Window)

(SPLIT '(3 1 5 2))=(3 5); (1 2); True? [yes]

⇩

③ Query : (XMERGE 'X '(2))
should be evaluated in
(MERGESORT '(3 1 5 2))
: True?

```
Assertion : (SPLIT '(3 1 5 2))=(3 5); (1 2);
FOLDING : SPLIT
Examine : XMERGE
0 : (MERGESORT '(3 1 5 2))=**UNDEFINED**;
 5 : (XMERGE '(3 5) '(1 2))=**UNDEFINED**;
  6 : (XMERGE X '(2))=**UNDEFINED**;
   99#: (CAR 'X)=**UNDEFINED**;
 3 : (XMERGE '(1) '(2))=(1 2);
  4 : (XMERGE nil '(2))=(2);
 1 : (XMERGE '(3) '(5))=(3 5);
  2 : (XMERGE nil '(5))=(5);
  98#: (SPLIT '(3 1 5 2))=(3 5); (1 2);
```
(Information Window)

See the top of the window and Answer the query [no]

⇩
⋮
⇩

④ INNER-REMOVAL : XMERGE

```
There is at least one bug in the instance :
(xmerge '(3 5) '(1 2))=**undefined**;
where
(defun xmerge (x y)
  (cond ((null x) y)
        ((null y) x)
        ((< (car x) (car y) (cons (car x) (xmerge (cdr x) y)))
        (t (cons (car y) (xmerge 'x (cdr y))))))
 ······· evaluated expressions ·······
  (null x): (null '(3 5))=nil;
  (null y): (null '(1 2))=nil;
  (< (car x) (car y)): (< (car '(3 5) (car '(1 2)))=nil;
     (car x): (car '(3 5))=3;
     (car y): (car '(1 2))=1;
  (cons (car y) (xmerge 'x (cdr y))):
       (cons (car '(1 2)) (xmerge 'x (cdr '(1 2))))=**undefined**;
     (cdr y): (cdr '(1 2))=(2);
```
(Information Window)

Figure 11: **Debugging a Program Suspended by a Run-Time Error**

①
```
(defun split (x)
  (if (null x)
     (values nil nil)
     (if (null (cdr x))
        (values x nil)
        (multiple-value-bind (u v) (split (cddr x))
           (values (cons (car x) u) (cons (cadr x) v))))))
(defun xmerge (x y)
  (if (null x)
     y
     (if (null y)
        x
        (if (< (car x) (car y))
           (cons (car x) (xmerge (cdr x) y))
           (cons (car y) (xmerge x (cdr y)))))))
(defun mergesort (x)
  (if (null (cdr x))
     x
     (multiple-value-bind (u v) (split x)
        (mergesort (xmerge u v)))))
(diagnoser '(mergesort (4 2 1 6)))
```
SHELL evaluator (show) File DEMO LSP

⇩

② Query : (SPLIT '(1 2 4 6))=(1 4);(2 6);True?

```
Examine SPG : (SPLIT MERGESORT XMERGE)
Examine : SPLIT
Count over! Filter closed. 300
0 : (MERGESORT '(4 2 1 6))=**UNDEFINED**;
 7 : (SPLIT '(1 2 4 6))=(1 4); (2 6);
  6 : (SPLIT '(4 6))=(4); (6);
   3 : (SPLIT nil)=NIL; NIL;
 5 : (SPLIT '(2 1 4 6))=(2 4); (1 6);
  6*: (SPLIT '(4 6))=(4); (6);
 4 : (SPLIT '(2 4 1 6))=(2 1); (4 6);
  2 : (SPLIT '(1 6))=(1); (6);
   3*: (SPLIT nil)=NIL; NIL;
 1 : (SPLIT '(4 2 1 6))=(4 1); (2 6);
  2*: (SPLIT '(1 6))=(1); (6);
```
(Information Window)

(SPLIT '(1 2 4 6))=(1 4); (2 6); True? [yes]

⇩

③ Query : (XMERGE '(1 4) '(2 6))=(1 2 4 6);True?

```
Assertion : (SPLIT '(1 2 4 6))=(1 4); (2 6);
FOLDING : SPLIT
Examine : XMERGE
Count over! Filter closed. 300
0 : (MERGESORT '(4 2 1 6))=**UNDEFINED**;
 11 : (XMERGE '(1 4) '(2 6))=(1 2 4 6);
  12 : (XMERGE '(4) '(2 6))=(2 4 6);
   10 : (XMERGE '(4) '(6))=(4 6);
    4 : (XMERGE nil '(6))=(6);
 99#: (SPLIT '(1 2 4 6))=(1 4); (2 6);
 8 : (XMERGE '(2 4) '(1 6))=(1 2 4 6);
  9 : (XMERGE '(2 4) '(6))=(2 4 6);
   10*: (XMERGE '(4) '(6))=(4 6);
 5 : (XMERGE '(2 1) '(4 6))=(2 1 4 6);
  6 : (XMERGE '(1) '(4 6))=(1 4 6);
   7 : (XMERGE nil  '(4 6))=(4 6);
 1 : (XMERGE '(4 1) '(2 6))=(2 4 1 6);
  2 : (XMERGE '(4 1) '(6))=(4 1 6);
```
(Information Window)

(XMERGE '(1 4) '(2 6))=(1 2 4 6); True? [yes]

⇩

④
```
(mergesort '(1 2 4 6))=**undefined**;
where
(defun mergesort (x)
  (if (null (cdr x))
    x
    (multiple-value-bind (u v) (split x) (mergesort (xmerge u v)))))
 ······· evaluated expressions ·······
Cyclic Path Detected! Filter closed.
 (MERGESORT '(1 2 4 6))··· (MERGESORT '(1 2 4 6)
0 : (MERGESORT '(1 2 4 6))=**UNDEFINED**;
 3 : (MERGESORT '(1 2 4 6))=(IINFINITE-LOOP TO INSTANCE 0);
 98#: (XMERGE '(1 4) '(2 6))=(1 2 4 6);
 99#: (SPLIT '(1 2 4 6))=(1 4); (2 6);
 2 : (NULL '(2 4 6))=NIL;
 1 : (CDR '(1 2 4 6))=(2 4 6);
```
(Information Window)

Figure 12: **Debugging a Program Incurring Infinite Invocations**

- DDS facilitates declaration of the intended properties of a program by effectively guiding the programmer.
- The diagnoser in DDS is implemented so efficiently in terms of both the amount of memory and CPU time that it can be adopted to interactive debugging for a variety of programs.

This paper discussed three essential mechanisms for implementing DDS: identification of buggy instances, partial execution of a program being debugged, and program diagnosis.

A new partial execution mechanism, called a filter, was proposed to control the execution of the program being debugged and to save a portion of the computation tree in a small, bounded buffer efficiently. To avoid computation and recomputation unnecessary for diagnosis, the filter "freezes" a portion of the computation, and makes it accessible as a special value to the computation requiring it. The filter is an elaboration of optimized evaluation techniques like tabulation and applicative caching, in that the diagnoser can specify functions whose instance values are to be retained, and also specify the portions of a program to be executed (or not executed).

This paper also presented a diagnoser whose bug-locating method has the following features.

- The heuristics on debugging of recursive functions makes bug locating efficient.
- The static analysis of a source program being debugged helps the diagnoser to deduce the location of a bug.

Finally, a prototype of DDS which has been implemented using Common Lisp on VAX/VMS demonstrated the usefulness of declarative debugging of functional programs.

Acknowledgments

The authors would like to thank Dr. Katsuji Tsukamoto, Director of NTT Software Research Laboratories, and Masaru Takesue for their guidance and encouragement. They also wish thank to Dr. Makoto Amamiya, Professor of Kyushu University, for his useful discussions and helpful comments.

References

[1] Henderson, P., "Functional Programming, Application and Implementation," Prentice-Hall, 1980.

[2] Takahashi, N., "Support Systems for Functional Programming," *J. IPSJ*, **29**, 8, pp.872-880, 1988 (in Japanese).

[3] Shapiro, E.Y., "Algorithmic Program Debugging," MIT Press, 1983.

[4] Plaisted, D.A., "An Efficient Bug Location Algorithm," *Proc. 2nd Int. Logic Programming Conference*, pp.151-157, 1984.

[5] Takeuchi, A., "Algorithmic Debugging of GHC Programs and its Implementation in GHC," *ICOT Tech. Rept.*, TR-185, ICOT, 1986.

[6] Takahashi, N., Ono, S. and Amamiya, M., "Parallel-Processing Oriented Bug-Location Method for Functional Programs" *Trans. IPSJ*, **27**, 4, pp.425-434, 1986 (in Japanese).

[7] Takahashi, N., Ono, S. and Amamiya, M., "An Algorithmic Bug-Location Method for Functional Programs Using Strategic Graph Transformation System," *Trans. IPSJ*, **27**, 9, pp.869-876, 1986 (in Japanese).

[8] Takahashi, N. and Ono, S., "Strategic Bug-Location Method for Functional Programs," *Proc. RIMS Symposium on Mathematical Methods in Software Science and Engineering*, RIMS, Kyoto University, pp.196-223, 1986.

[9] Fairley, R., "Software Engineering Concepts," McGraw-Hill Book Company, 1985.

[10] Bird, R.S., "Tabulation Techniques for Recursive Programs," *ACM Computing Surveys*, **12**, pp.403-418, 1980.

[11] Keller, R.M and Sleep, M.R., "Applicative Caching," *ACM Trans. on Programming Languages and Systems*, **8**, 1, pp.88-108, 1986.

[12] Steele, G.L. et al., "Common LISP," Digital Press, 1984.

[13] Takahashi, N. and Ono, S., "A Declarative Debugger for Functional Programs," *Proc. of Data Flow Workshop, IEICE*, pp.255-262, 1987 (in Japanese).