TR-185

Algorithmic Debugging of GHC Programs and its implementation in GHC

by
A. Takeuchi

June, 1986

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Algorithmic Debugging of GHC Programs and its implementation in GHC

Akikazu Takeuchi

ICOT Research Center
Institute for New Generation Computer Technology
1–4–28, Mita, Minato-ku, Tokyo 108 Japan

Last Revision: June 6, 1986

1. Introduction

As parallel computers become to be widely used, the debugging of programs running in parallel will become important technology for software development. Even in sequential computers, debugging of programs which are written in parallel languages or languages with extended control such as coroutine is large obstacle for software development. In fact, it has been said that debugging of these programs is very hard task, compared with debugging of sequential programs. The reasons why it is difficult are that 1) conceptually several computations are executed in parallel, 2) these computations may interact with each other and 3) there are new types of bugs such as deadlock.

In conventional debugging method, a programmer tries to find a bug by observing execution trace. If incorrect behavior is found at some place, then the bug is detected in its neighborhood. For this style of debugging, it is assumed that a programmer has complete knowledge of program behavior, and it is required that the execution trace is displayed in structural way in order for the programmer to recognize easily flow of computation and to compare trace with his knowledge. The flaw of this style of debugging is its complexity, which increases as a program becomes complicated. As the complexity of the program increases, both the amount of the execution trace and the amount of knowledge a programmer must have increases, so that the programmer gets into danger of misjudging. Concerning with parallel programs, in the reasons mentioned in the first paragraph, the complexity of a program is surmised to increase. Therefore we suspect that debugging parallel programs using execution trace will face to serious problems.

In general, we should distinguish between debugging and understanding of a program behavior. It is essential to clearly display what are happening in computer in order for a programmer to understand data and control flows of a program. Monitoring of a program behavior will help finding a bug, but it forces a programmer to understand a program behavior. It is better if a programmer could debug a program only with more abstract knowledge such as input and output specification of component modules.

Generally speaking, a logic program has two different aspects, logic and control. Logic aspect of a program directly relates to its declarative semantics of the program and control

aspect to its procedural semantics. Owing to these double aspects, a program can be read declaratively and procedurally. This is also true for debugging, that is, debugging can be done declaratively and/or procedurally. Conventional debugging corresponds to debugging using procedural meaning of a program. Debugging using declarative meaning is known as algorithmic debugging first investigated by Shapiro [Shapiro83a]. In algorithmic debugging, a bug is detected by observing the result of computation such as input/output history, rather static information than dynamic behavior of the program. There are several algorithms for locating the bug. Each algorithm guarantees that the debugger can find a bug within finite question-answering. Each question issued by the debugger is isolated, that is, it concerns with the correctness of the result of a procedure. Therefore answering each question is easy no matter how a program is complicated, although the number of questions depends on the complexity of the program. Therefore the algorithmic debugging is a steady and efficient way of debugging, and is a promising method of debugging parallel programs.

The pioneering work of algorithmic debugging is due to Shapiro [Shapiro83a]. Several researchers investigated algorithmic debugging of logic programs further [Ferrand85, Pereira86, Lloyd86]. Algorithmic debugging of functional languages are investigated in [Takahashi85].

In this paper, we present an algorithmic debugger for a parallel logic programming language, GHC [Ueda85]. The debuggers with different debugging strategies, "single stepping" and "divide and query", are shown with their implementation in GHC. The formal framework of our debugger is investigated in the subsequent paper [Lloyd&Takeuchi86].

Readers are assumed to be familiar to parallel logic programming and algorithmic debugging.

2. GHC

A GHC program is a finite set of guarded clauses. A guarded clause has form:

$$H :- G_1, ..., G_n | B_1, ..., B_m.$$

where H, $G_1, ..., G_n$ and $B_1, ..., B_m$ are called the head, the guard part and the body part of the clause, respectively.

For the simplicity, the algorithm developed in this paper only deals with body parts of GHC clauses. Therefore we will restrict GHC programs to be Flat GHC programs, which are only allowed to have system predicates in guard parts.

3. Framework for algorithmic debugging of GHC programs

We summarize the framework of algorithmic debugging of GHC programs following [Lloyd&Takeuchi86].

In order to debug a buggy program, the abstract meaning of a program, which is called the intended interpretation of the program, is required. Shapiro's system uses the minimum model obtained from the program as its intended interpretation. In our system, the intended interpretation of a program has the following definition.

Definition 1 *Given a program S, Msuc and Msus are defined to be the following two sets.*

$$Msuc = \{A : \text{there exists a derivation from} \leftarrow A \text{ which succeeds without further instantiation of } A\}$$

$$Msus = \{A : \text{there exists a derivation from} \leftarrow A \text{ which suspends without further instantiation of } A\}$$

We call *Msuc* and *Msus* together, the "*intended interpretation*" of the program S.

The debugging algorithms are applied to a computation tree of a given goal $\leftarrow A$. The computation tree is a proof tree of the goal and is also a final AND-tree, to which all the substitutions made during the computation have been applied. The advantage of using a final AND-tree is that computations in disjoint subtrees can be regarded as independent computations.

Definition 2 *We say a node $\leftarrow A$ in the computation tree has been executed incorrectly*

if *(a) $\leftarrow A$ succeeds in the tree, but $A \notin Msuc$. or*
(b) $\leftarrow A$ suspends in the tree, but $A \notin Msus$.

Two major errors of GHC programs considered in this paper are defined as follows.

Definition 3 *Result of computation is called termination with incorrect answer if the computation has terminated with incorrect answer. Result of computation is called buggy deadlock if the computation deadlocks where it should not deadlock.*

Two major bugs corresponding to the two major errors are *incorrect clause instance* and *buggy suspension*.

Definition 4 *We say a clause instance,*

$$H :- G_1, ..., G_n | B_1, ... B_m.$$

invoked at some node in the computation tree is incorrect if H has been incorrectly executed, but each goal B_j has been correctly executed, for $j = 1, ..., m$. We say a goal $\leftarrow A$ invoked at some node in the computation tree is buggy suspension if $\leftarrow A$ immediately suspends, but $A \notin Msus$.

Buggy suspension indicates that there is a missing clause in the definition called by the goal. The correlation between errors and bugs are shown in the table 1.

We assume that a programmer knows what literals are included in *Msuc* and what literals in *Msus*, as declarative debuggers of pure Prolog. Therefore, when it is necessary

Table 1 Relation between errors and bugs

Errors\Bugs	Incorrect Clause	Buggy Suspension
Incorrect Answer	∘	×
Buggy Deadlock	∘	∘

∘ indicates that an error is invoked by a bug.
× indicates that an error is not invoked by a bug.

to know whether a node $\leftarrow A$ in a computation tree has been executed correctly or not, a query is issued to the programmer. The query is called an *oracle query* and has two forms depending on the context of the node in the computation tree.

"*succeeded*(A)?"	if $\leftarrow A$ succeeds in the computation tree.
	Meaning: $A \in Msuc$?
"*suspended*(A)?"	if $\leftarrow A$ suspends in the computation tree.
	Meaning: $A \in Msus$?

4. Implementation

There are several strategies for locating a bug. Shapiro proposed two strategies, "single stepping" and "divide and query" [Shapiro83a]. Pereira proposed dependency directed strategy [Pereira86]. In [Lloyd86, Lloyd&Takeuchi86], the top-down strategy was adopted. In this paper, we present GHC implementation of the debuggers based on "single stepping" and "divide and query" strategies. In section 4.1, the debugger for termination with incorrect answer adopting the "single stepping" strategy is presented. Section 4.2 presents the debugger for the same error based on the "divide and query" strategy. In section 4.3, the debugger handling both termination with incorrect answer and buggy deadlock based on the "divide and query" strategy is presented.

4.1 "Single Stepping" algorithm for termination with incorrect answer

When a program terminates with incorrect answer, it can be concluded that there is at least one incorrect clause instance.

The algorithm to find such clause is specified as follows:

Algorithm 1
Input: A goal G that terminates with incorrect answer substitution.
Output: An incorrect clause instance.

Simulate the execution of G; whenever a subgoal Q terminates with Q', check, using an oracle query, whether Q' has been executed correctly. If not, return the clause invoked there as an incorrect clause instance.

The point of this algorithm is that by confirming the result of computation whenever a goal terminates, we can find such clause instance invoked at some node in the computation tree that generates an incorrect answer, but all its body goals have been executed correctly. It is clear that such clause is an incorrect clause instance. The query complexity (the maximum number of queries issued by the debugger) of this algorithm is proportional to the number of goal reductions. GHC implementation of this algorithm is shown below.

```
% single_stepping(Goal,IncClsIns) :-
```
Given `Goal`, *it returns an incorrect clause instance at* `IncClsIns`.

```
single_stepping(Goals,IncClsIns) :- true | tree(Goals,IncClsIns,_).

% tree(Goals,IncClsIns,Ctr) :-
```
Given `Goals`, *it returns an incorrect clause instance at* `IncClsIns`. `Ctr` *is used to abort irrelevant computation.*

```
tree(_,_,abort) :- true | true.
tree(true,X,_) :- true | X=ok.
tree(A,X,_) :- ghcsystem(A) | X=ok, call(A).
tree((A, B),X,C) :- true |
    tree(A,Xa,Ca), tree(B,Xb,Cb), and(Xa,Ca,Xb,Cb,X,C).
tree(A,X,C) :- reduce(A,Body) |
    tree(Body,X1,C), reduction(X1,(A:-Body),X,C).
```

`ghcsystem(A)` is a system predicate checking whether `A` is a system predicate or not. `reduce(A,Body)` is also a system predicate which, given a goal `A`, returns a body part of the clause, the head and the guard of which have been successfully solved. `tree` simulates execution of the goal, at the same time it forms a computation tree as the network of `reduction` and `and`.

```
% reduction(X1,Clause,X,Ctr) :-
```
`reduction` *corresponds to reduction of a goal to a body part.* `Clause` *is an instance of the clause used for the reduction.* `X` *and* `X1` *are used to send an incorrect clause instance to the head and to receive it from the body part, respectively.*

```
reduction(ok,(P:-Q),X,_) :- true |
    query(P,Ans), react(Ans,(P:-Q),X).
reduction((P:-Q),_,X,_) :- true | X=(P:-Q).
reduction(_,_,_,abort) :- true | true.

and(_,Ca,_,Cb,_,abort) :- true | Ca=abort, Cb=abort.
and((P:-Q),_,_,Cb,X,_) :- true | Cb=abort, X=(P:-Q).
and(_,Ca,(P:-Q),_,X,_) :- true | Ca=abort, X=(P:-Q).
and(ok,_,Xb,_,X,_) :- true | Xb=X.
and(Xa,_,ok,_,X,_) :- true | Xa=X.

react(yes,_,Ans) :- true | Ans=ok.
react(no,(P:-Q),Ans) :- true | Ans=(P:-Q).
```

When `reduction(X1,(P:-Q),X,Ctr)` received `ok` at `X1` from the body part, it means the body goals `Q` have been executed correctly. Then it asks a programmer whether `P` has been executed correctly or not by the predicate `query(P,Ans)`. If answer is yes, then it returns `ok` to `X`. If the answer is no, it means that `P:-Q` is an incorrect clause instance. Hence it is returned to `X`. When `X1` is already instantiated to some clause, it means that an incorrect clause instance has been already found. Such clause is passed to `X`.

`and`s make the intelligent communication network for `reduction`s in the form of the binary tree. They carry the information about correctness of reductions to their parent reductions. At the same time, they send `abort` messages to appropriate `reduction`s in order to kill irrelevant computation.

Conceptually the "single stepping" debugger has two phases. In the first phase, it forms the computation tree as a tree of `reduction`'s inter-connected by `and`. In the second phase, for each reduction from leaves to the root, the result of reduction is confirmed using oracle queries. In actual execution, these two phases are appropriately overlapped.

Example 1: Debugging of a program terminating with incorrect answer by "Single stepping" algorithm

Source of buggy quick sort which returns [] with [3,1,2] as input.

```
qsort(Xs, Ys) :- qsort(Xs, Ys, []).

qsort([X|Xs], Ys0, Ys2) :-
%    part(Xs, X, S, L), qsort(S, Ys0, [X|Ys1]), qsort(L, Ys1, Ys2).
     part(Xs, X, S, L), qsort(S, Ys0, Ys1), qsort(L, Ys1, Ys2).
qsort([],     Ys, 0) :- true | 0=Ys.

part([X|Xs], A, S, L) :- A <  X | L=[X|L1], part(Xs, A, S, L1).
part([X|Xs], A, S, L) :- A >= X | S=[X|S1], part(Xs, A, S1, L).
part([],     _, S, L) :- true | S=[], L=[].
```

Log:

```
| ?- ghc single_stepping(qsort([3,1,2],P),L).
succeeded(part([],3,[],[])) ? (yes/no) y.
succeeded(part([2],3,[2],[])) ? (yes/no) y.
succeeded(part([1,2],3,[1,2],[])) ? (yes/no) y.
succeeded(part([],1,[],[])) ? (yes/no) y.
succeeded(part([2],1,[],[2])) ? (yes/no) y.
succeeded(qsort([],_57,_57)) ? (yes/no) y.
succeeded(part([],2,[],[])) ? (yes/no) y.
succeeded(qsort([],_57,_57)) ? (yes/no) y.
succeeded(qsort([],_57,_57)) ? (yes/no) y.
succeeded(qsort([2],_57,_57)) ? (yes/no) n.
succeeded(qsort([],[],[])) ? (yes/no) y.
```

```
L = qsort([2],[],[]):-part([],2,[],[]),qsort([],[],[]),qsort([],[],[]),
P = []

yes
```

4.2 "Divide and Query" algorithm for termination with incorrect answer

Using the "divide and query" strategy, we can improve the query complexity of the algorithm. The basic idea is the binary search for an incorrect clause instance over the computation tree spawned by "reduction"s. Before specifying the algorithm, first we define the weight of reduction in the computation tree.

Definition 5 *Let G be a goal reduced in a reduction R.*

(a) $weight(R) = 0$ *if G is a system predicate.*
(b) $weight(R) = 1 + \sum_{i=1}^{N} weight(R_i)$ *if R invokes N reductions, $R_1, ..., R_N$.*

The weight of a reduction reflects size of the subtree rooted at that reduction.

Algorithm 2
Input: A goal G that terminates with incorrect answer substitution and a (possibly empty) subset M' of $M_{suc} \cup M_{sus}$.
Output: An incorrect clause instance.

Simulate the execution of G, compute w, the weight of the reduction of G modulo M'. If w is 1, then the clause invoked at the root is returned. Otherwise, it finds the heaviest node Q in the computation tree whose weight is less than or equal to $\lceil w/2 \rceil$ and queries oracle whether the goal G_Q reduced at Q has been executed correctly or not. If the oracle answers "yes", then the algorithm is applied to the same computation tree with $M' \cup \{G_Q\}$ as new M'. If the oracle answers "no", then the algorithm is applied to the computation tree rooted at Q with the same M'.

It is known that the maximum number of the queries of this algorithm is $O(log(N))$, where N is the number of reductions.

GHC implementation of this algorithm is shown below. The predicates similar to those in the single stepping algorithm have the same names.

```
divide&query(Goal,IncClsIns) :-
    true |
        tree(Goal,W,I/0),
        cursor([down|In],Out,top(_,_),mid(0,I)),
        oracle(Out,In,[],IncClsIns).
```

`divide&query(Goal,IncClsIns)` is a top level procedure, which invokes three subgoals, `tree`, `cursor` and `oracle`. Roughly speaking, `tree` simulates the computation of the goal `Goal` and forms the computation tree by spawning `reductions` which correspond to reductions. `cursor` points two important nodes in the tree, the root and the middle nodes, and can

communicate with them. **cursor** is also an interface between these two nodes and **oracle**. **oracle** is the manager of oracle queries.

```
% tree(Goals,Weights,Input/Output) :-
```

Given **Goals**, **tree** *simulates the computation of* **Goals** *and forms the computation tree as network of* **reductions**. **Weights** *is the sum of the initial weights of reductions of* **Goals**. **Input** *and* **Output** *are streams from and to outside, respectively.*

```
tree((A,B),Wab,I/O) :-
    true |
        tree(A,Wa,I/Oa), tree(B,Wb,I/Ob),
        Wab := Wa+Wb, omerge(Oa,Ob,O).
tree(A,Wa,I/O) :-
    ghcsystem(A) |
        Wa=0, call(A), O=[].
tree(A,Wa,I/O) :-
    reduce(A,Body) |
        tree(Body,Wb,Ob/Ib),
        Wa := Wb+1,
        reduction(I,O,Ib,Ob,Wa,(A:-Body)).

omerge([middle((Ra,Wa),Ca)|X],[middle((Rb,Wb),mid(I,O))|Y],Z) :-
    Wa>=Wb | Z=[middle((Ra,Wa),Ca)|ZZ], O=[], omerge(X,Y,ZZ).
omerge([middle((Ra,Wa),mid(I,O))|X],[middle((Rb,Wb),Cb)|Y],Z) :-
    Wa<Wb  | Z=[middle((Rb,Wb),Cb)|ZZ], O=[], omerge(X,Y,ZZ).
omerge([A|X],Y,Z) :- A\=middle(_,_) | Z=[A|ZZ], omerge(X,Y,ZZ).
omerge(X,[A|Y],Z) :- A\=middle(_,_) | Z=[A|ZZ], omerge(X,Y,ZZ).
omerge([],Y,Z) :- true | Y=Z.
omerge(X,[],Z) :- true | X=Z.
```

omerges makes the intelligent communication network for **reductions** in the form of the binary tree.

The tree formed by the network of **reduction** and **omerge** is responsible to computing the weights, finding the middle point of the tree and updating the tree and the weights according to oracles.

```
% reduction(I,O,Ib,Ob,W,Clause) :-
```

It represents a reduction in the computation tree. **Clause** *is an instance of the clause invoked there.* **I** *and* **O** *are streams from and to the preceding reduction, respectively.* **Ib** *and* **Ob** *are streams from and to the reductions of the body goals.* **W** *is the weight. Initial weights are computed when the computation tree are formed by* **tree**.

```
reduction([],Op,_,Oc,_,_) :-
    true |
        Oc=[], Op=[].                                              (1)
reduction([down|Mi],Mo,Ic,Oc,1,Clause) :-
    true |
        Mo=[answer(Clause)], Oc=[].                                (2)
reduction([down|Mi],Mo,Ic,Oc,Wa,Clause) :-
    NBD := (Wa+1)/2, Wa=\=1 |
        Oc=[new(NBD)|Oc2],
        reduction2(Mi,Mo,Ic,Oc2,Wa,Clause).                        (3)
reduction([new(NBD)|Ip],Op,Ic,Oc,Wa,Clause) :-
    Wa>NBD |
        Oc=[new(NBD)|Oc2],
        reduction2(Ip,Op,Ic,Oc2,Wa,Clause).                        (4)
reduction([new(NBD)|Ip],Op,Ic,Oc,Wa,Clause) :-
    Wa=<NBD |
        Op=[middle((Clause,Wa),mid(Mo,Mi))|Op2],
        middle(Mi,Mo,Ip,Op2,Ic,Oc,Wa,Clause).                      (5)
reduction(Ip,Op,[update(Wb,_)|Ic],Oc,Wa,Clause) :-
    true |
        Wa1 := Wa-Wb, Op=[update(Wb,Wa1)|Op2],
        reduction(Ip,Op2,Ic,Oc,Wa1,Clause).                        (6)
reduction2(Ip,Op,[middle(M,C)|Ic],Oc,Wa,Clause) :-
    true |
        Op=[middle(M,C)|Op2],
        reduction(Ip,Op2,Ic,Oc,Wa,Clause).                         (7)

% middle(Mi,Mo,Ip,Op,Ib,Ob,Weight,Clause) :-
```

middle(Mi,Mo,Ip,Op,Ib,Ob,Weight,Clause) *is a variant of* reduction(Ip, Op,Ib,Ob,Weight,Clause) *which locates in the middle point of the computation tree.* Mi *and* Mo *are streams from and to* cursor.

```
middle([terminal],Mo,Ip,Op,Ic,Oc,Wa,Clause) :-
    true |
        Op=[update(Wa,0)|Op2], Mo=[], Oc=[],
        reduction(Ip,Op2,_,_,0,Clause).                            (8)
middle([down|Mi],Mo,Ip,Op,Ic,Oc,1,Clause) :-
    true |
        Mo=[answer(Clause)],Oc=[],Op=[].                           (9)
middle([down|Mi],Mo,Ip,Op,Ic,Oc,Wa,Clause) :-
    NBD := (Wa+1)/2, Wa=\=1 |
        Oc=[new(NBD)|Oc2], Op=[],
        reduction2(Mi,Mo,Ic,Oc2,Wa,Clause).                        (10)
middle([],Mo,Ip,Op,Ic,Oc,Wa,Clause) :-
    true |
        Mo=[], reduction(Ip,Op,Ic,Oc,Wa,Clause).                   (11)
```

The middle point of the computation tree is determined in the following way: First reduction at the root of the computation tree receives down message. Let w be the current weight of

the root. Then it computes the $\lceil w/2 \rceil$, sends it downward as **new** message and waits for the reply (3). On receiving **new** message, **reduction** sends it downward further if its current weight is greater than the value (4). Otherwise **reduction** changes to **middle** and returns the clause invoked there, its current weight and input/output streams as **middle** message (5). On receiving **middle** message, **reductions** waiting for reply pass it upward (7). In this algorithm, more than one **middles** are created, however, **omerges** select one which has the greatest weight and the rest of **middles** change back to **reductions** (11).

```
% cursor(In,Out,top(Ti,To),mid(Mi,Mo)) :-
```

cursor points two important **reductions** *in the computation tree, the root and the middle nodes, and can communicate with them.* `Ti` *and* `To` *are streams from and to the root, and* `Mi` *and* `Mo` *are streams from and to the middle point, respectively.*

```
cursor([down|In],Out,top(Ti,To),mid(Mi,Mo)) :-
    true |
        Mo=[down|Mo2], To=[],
        cursor2(In,Out,top(Mi,Mo2)).                              (12)
cursor([up|In],Out,Top,mid(Mi,Mo)) :-
    true |
        Mo=[terminal],
        cursor2(In,Out,Top).                                      (13)
cursor2(In,Out,top([answer(Clause)|Ti],To)) :-
    true |
        Out=[answer(Clause)],To=[].                               (14)
cursor2(In,Out,top([middle(Mid,MChan)|Ti],To)) :-
    true |
        Out=[middle(Mid)|Out2],
        cursor(In,Out2,top(Ti,To),MChan).                         (15)
cursor2(In,Out,top([update(_,W)|Ti],To)) :-
    NBD:=(W+1)/2 |
        To=[new(NBD)|To2],
        cursor2(In,Out,top(Ti,To2)).                              (16)
```

cursor is an interface between the root and the middle point of the tree and **oracle**. Initially **cursor** only points the root of the tree as the middle point and has received the **down** message (see the definition of `divide&query`). The message initiates the computation of the middle point.

Whenever **cursor** receives the new middle point of the tree, it sends the goal reduced in the middle point to **oracle**. **oracle** replies **down** or up to **cursor**, depending on whether the goal has been executed correctly or not. On receiving the **down** message from **oracle**, **cursor** sends the **down** message to the middle point in order to examine the subtree rooted at the middle point and to discard the rest of the tree. The message initiates the computation of the new middle point of the new tree. On receiving the **up** message, **cursor** sends the **terminal** message to the middle point in oder to discard the subtree rooted at the middle

point and to examine the rest of the tree. The message initiates the computation of the new middle point of the updated tree.

```
% oracle(In,Out,QDB,IncClsIns) :-
```

`oracle` *receives a goal from the input stream* `In` *and queries the programmer whether the goal has been executed correctly or not. If reply from the programmer is yes, then the message* `up` *is sent through the output stream* `Out`. *Otherwise the message* `down` *is sent. The queries already asked and their answers are stored in* `QDB`. `QDB` *is used to suppress to issue the same query.*

```
oracle([middle(((P:-Q),W))|In],Out,S,IncClsIns) :-
    true |
        member(P,S,R), branchonr(R,Out,In,P,S,IncClsIns).
oracle([answer((P:-Q))],Out,S,IncClsIns) :-
    true |
        IncClsIns=(P:-Q).
branchonr(yes,Out,In,Query,S,IncClsIns) :-
    true |
        Out=[up|Out2], oracle(In,Out2,S,IncClsIns).
branchonr(no,Out,In,Query,S,IncClsIns) :-
    true |
        Out=[down|Out2], oracle(In,Out2,S,IncClsIns).
branchonr(unknown,Out,In,Query,S,IncClsIns) :-
    query(Query,Ans) |
        branchonr2(Ans,Out,In,[fact(Query,Ans)|S],IncClsIns).
branchonr2(yes,Out,In,S,IncClsIns) :-
    true |
        Out=[up|Out2], oracle(In,Out2,S,IncClsIns).
branchonr2(no,Out,In,S,IncClsIns) :-
    true |
        Out=[down|Out2], oracle(In,Out2,S,IncClsIns).
```

4.3 "Divide and Query" algorithm for both errors

The debugging algorithm for a program which deadlocks can be obtained by slightly extending the debugger above.

First the top level procedure is changed to `divide&query(Goal,BUG,DL)` where `DL` is *deadlock flag* which is uninstantiated during the computation of the `Goal` and is instantiated externally to `deadlock` when the computation deadlocks. It is impossible to implement such deadlock detecting mechanism in GHC itself. Discussion of the implementation of such mechanism is beyond the scope of this paper and we assume that there is such mechanism. `divide&query` invokes three subgoals, `tree`, `cursor` and `oracle`, where `tree` and `oracle` are extended and `cursor` is unchanged.

The basic idea of extension is to make the computation tree include suspended goals upon deadlocking. Once the computation tree including suspended goals is formed, the

debugging algorithm used so far can be applied as before.

```
% divide&query(Goal,BUG,DL) :-
```

Given `Goal`, *it returns an incorrect clause instance or buggy suspension at* `BUG`.

```
divide&query(Goal,BUG,DL) :-
    true |
        tree(Goal,W,I/O,H-[],DL),
        cursor([down|In],Out,top(_,_),mid(O,I)),
        oracle(Out,In,[],BUG).
```

```
% tree(Goals,Weights,Input/Output,H-T,DL) :-
```

Given `Goals`, `tree` *simulates the computation of* `Goals` *and forms the computation tree as network of* `reductions`. *If the simulation deadlocks, owing to external instantiation of DL, it makes the computation tree including suspended goals.* `Weights` *is the sum of the initial weights of reductions of* `Goals`. `Input` *and* `Output` *are streams from and to outside, respectively.* `H-T` *is the difference list of the goals which are included in* `Goals` *and have not terminated upon deadlocking.*

`tree` is completely redefined as follows.

```
tree((A,B),Wab,I/O,H-T,DL) :-
    true |
        tree(A,Wa,I/Oa,H-T1,DL), tree(B,Wb,I/Ob,T1-T,DL),
        Wab := Wa+Wb, omerge(Oa,Ob,O).
tree(A,Wa,I/O,H-T,DL) :-
    ghcsystem(A) |
        call(A,Res,Ctr), filter(Res,A,H-T,DL,Ctr),
        O=[], Wa=O.
tree(A,Wa,I/O,H-T,DL) :-
    reduce(A,Body) |
        tree(Body,Wb,Ob/Ib,H1-[],DL),
        Wa := Wb+1, switch(H1,H-T,A),
        check_termination(H1,I,O,Ib,Ob,Wa,(A:-Body)).
tree(A,Wa,I/O,H-T,deadlock) :-
    true |
        H=[susp(A)|T], Wa=1, reduction(I,O,_,_,_,1,susp(A)).
```

The last clause defining `tree` handles the suspended goal when the entire computation deadlocks. It creates `reduction` with `susp(A)` instead of the clause used in the reduction.

```
% filter(Res,A,H-T,DL,Ctr) :-
```

`filter` *examines the result* `Res` *of the evaluation of the system predicate and unifies* `H` *with* `T` *if it is* `success`. *Otherwise, upon deadlocking,* `susp(A)` *is registered in the difference list* `H-T`.

```
filter(success,_,H-T,_,_) :-
    true |
        H=T.
filter(Res,A,H-T,deadlock,Ctr) :-
    true |
        Ctr=stop, H=[susp(A)|T].

% switch(H1,H-T,A) :-
```

switch examines the difference list `H1-[]` *and registers* `open(A)` *in* `H-T` *if there is at least one element. Otherwise it unifies* `H` *with* `T`.

```
switch([],H-T,_) :-
    true |
        H=T.
switch([_|_],H-T,A) :-
    true |
        H=[open(A)|T].
```

In order to distinguish between the reduction at least one of descendants of which suspends and the reduction all of descendants of which terminate, `check_termination` is introduced. It creates **`reduction`** with `open(Clause)` for the former and **`reduction`** with **`Clause`** for the latter.

```
check_termination([],Ip,Op,Ic,Oc,Weight,Clause) :-
    true |
        reduction(Ip,Op,Ic,Oc,Weight,Clause).
check_termination([A|B],Ip,Op,Ic,Oc,Weight,Clause) :-
    true |
        reduction(Ip,Op,Ic,Oc,Weight,open(Clause)).
```

`oracle` is extended to handle the new terms such as `susp(P)` and `open(C)`. The new definition of `oracle` is shown below.

```
oracle([middle((susp(P),W))|In],Out,S,BUG) :-
    true |
        member(suspended(P),S,R), branchonr(R,Out,In,suspended(P),S,BUG).
oracle([middle((open((P:-Q)),W))|In],Out,S,BUG) :-
    true |
        member(open(P),S,R), branchonr(R,Out,In,open(P),S,BUG).
oracle([middle(((P:-Q),W))|In],Out,S,BUG) :-
    true |
        member(P,S,R), branchonr(R,Out,In,P,S,BUG).
oracle([answer(Clause)],Out,S,BUG) :-
    true |
        BUG=answer(Clause).
```

Example 2: Debugging of a program terminating with incorrect answer by the "Divide and Query" algorithm

Buggy quick sort which terminates with incorrect answer [] for given input [3,1,2].

```
qsort(Xs, Ys) :- qsort(Xs, Ys, []).

qsort([X|Xs], Ys0, Ys2) :-
%    part(Xs, X, S, L), qsort(S, Ys0, [X|Ys1]), qsort(L, Ys1, Ys2).
     part(Xs, X, S, L), qsort(S, Ys0, Ys1), qsort(L, Ys1, Ys2).
qsort([],     Ys, 0) :- true | 0=Ys.

part([X|Xs], A, S, L) :- A <  X | L=[X|L1], part(Xs, A, S, L1).
part([X|Xs], A, S, L) :- A >= X | S=[X|S1], part(Xs, A, S1, L).
part([],     _, S, L) :- true | S=[], L=[].
```

Log:

```
| ?- ghc divide&query(qsort([3,1,2],P),DL).
succeeded(qsort([2],[],[])) ? (yes/no) n.
succeeded(part([],2,[],[])) ? (yes/no) y.
succeeded(qsort([],[],[])) ? (yes/no) y.

answer((qsort([2],[],[]):-part([],2,[],[]),qsort([],[],[]),qsort([],[],[])))

DL = _78,
P = []

yes
```

Since the simulation of the goal does not deadlock in the above case, it is not necessary to instantiate externally DL.

Example 3: Debugging of buggy deadlock by the "Divide and Query" algorithm

Buggy quick sort which deadlocks for given input [3,1,2].

```
qsort(Xs, Ys) :- qsort(Xs, Ys, []).

qsort([X|Xs], Ys0, Ys2) :-
    part(Xs, X, S, L), qsort(S, Ys0, [X|Ys1]), qsort(L, Ys1, Ys2).
qsort([],     Ys, 0) :- true | 0=Ys.

% part([X|Xs], A, S, L) :- A <  X | L=[X|L1], part(Xs, A, S, L1).
part([X|Xs], A, S, [X|L1]) :- A <  X | part(Xs, A, S, L1).
part([X|Xs], A, S, L) :- A >= X | S=[X|S1], part(Xs, A, S1, L).
part([],     _, S, L) :- true | S=[], L=[].
```

Log:

```
| ?- ghc divide&query(qsort([3,1,2],P),DL).

%% It is assumed that DL is instantiated externally to
%% deadlock upon deadlocking of the simulation of the goal.
```

```
suspended(qsort([1,2],_57,[3])) ? (yes/no) n.
suspended(part([2],1,_1452,_1453)) ? (yes/no) n.

answer(susp(part([2],1,_1452,_1453)))

DL = deadlock,
P = _57

yes
```

The answer indicates that `part([2],1,_1452,_1453)` is the buggy suspension and that there is a missing clause in the definition of `part`.

5. Concluding Remarks

We have presented the debugging algorithms for two major errors of GHC programs, termination with incorrect answer and buggy deadlock together with its GHC implementation.

The subsequent paper [Lloyd&Takeuchi86] gives the formal framework of the algorithmic debugging of GHC. It is proved there that the debugger can always find a bug under certain condition. The handling of buggy failure is also presented in that paper.

Although we concentrate on the debugger of GHC programs, the technique developed here will be applicable to other parallel logic programming languages such PARLOG [Clark&Gregory84] and Concurrent Prolog [Shapiro83b].

Following types of bugs can be treated by our debugger with appropriate augmentation.

- Infinite loop
- Bugs in the guard parts.

Following types of bugs are difficult to handle in our framework.

- Bugs without reappearance because of nondeterminism
- Bugs caused by unfairness of scheduling such as starvation
- Termination with correct answer but incorrect behavior

References

[Clark&Gregory84] Clark, K., and Gregory, S., *PARLOG: Parallel Programming in Logic*, Research Report DOC 84/4, Imperial College, 1984.

[Ferrand85] Ferrand, G., *Error Diagnosis in Logic Programming: An Adaptation of E. Y. Shapiro's Method*, Rapport de Recherche 375, INRIA, 1985.

[Lloyd86] Lloyd, J., *Declarative Error Diagnosis*, Technical Report 86/3, Dept. of Computer Science, Univ. of Melbourne, 1986.

[Lloyd&Takeuchi86] Lloyd, J. and Takeuchi, A., *A Framework for Debugging GHC*, ICOT Technical Report TR-186, Institute for New Generation Computer Technology, 1986.

[Pereira86] Pereira, L. M., "Rational Debugging in Logic Programming," In *Proc. Third Int. Conf. on Logic Programming*, Springer-Verlag, 1986.

[Shapiro83a] Shapiro, E., *Algorithmic Program Debugging*, MIT Press, 1983.

[Shapiro83b] Shapiro, E., *A Subset of Concurrent Prolog and Its Interpreter*, ICOT Technical Report TR-003, Institute for New Generation Computer Technology, 1983.

[Takahashi85] Takahashi, N. and Ono, S., "Strategic Bug Location Method for Functional Programs," In *Proc. 6th RIMS Symp. on Mathematical Methods in Software Science and Engineering*, RIMS, Kyoto Univ., 1985.

[Ueda85] Ueda, K., *Guarded Horn Clauses*, ICOT Technical Report TR-103, Institute for New Generation Computer Technology, 1986. Also in *Logic Programming'85*, E. Wada (ed.), Lecture Notes in Computer Science 221, Springer-Verlag, 1986.