ICOT Technical Report: TR-186

TR-186

A Framework for Debugging GHC

by
J. Lloyd (Melbourne Univ.)
and A. Takeuchi (ICOT)

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

A FRAMEWORK FOR DEBUGGING GHC

J. Lloyd† and *A. Takeuchi*‡
(alphabetical order)

† Department of Computer Science
University of Melbourne
Parkville, Victoria 3052, Australia

‡ 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

Debugging has been important subject of computer science, but has been put in the subject of the heuristic expertise. Initiated by the pioneering work of Shapiro [Shapiro83], several researchers investigated declarative debugging of logic programs [Ferrand85, Pereira86, Lloyd86]. It seems that these works have succeeded in lifting debugging to the scientific subject with a theoretical foundation.

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. Declarative debugging enables to detect a bug using only declarative meaning of a program and no procedural meaning.

GHC is a parallel programming language derived from general model of logic programs. It has primitives for controlling parallel computation. Programming in GHC heavily relies on these primitives. Errors of GHC programs such as deadlock also depends on these control primitives. An algorithmic debugger for a GHC program was implemented by [Takeuchi86]. He defined an intended interpretation of GHC programs and implemented the debugger using "divide and query" strategy in GHC. We examine a framework for debugging GHC used in the above system in more detail. The framework is procedural, but this is probably unavoidable because of given nature of GHC.

We give analogous definitions to declarative debugging of Prolog such as "incorrect clause instance" except here the meaning is procedural — all atoms in body are "executed correctly", but head atom is "not executed correctly".

Readers are assumed to be familiar to declarative debugging and GHC.

2. Termination with incorrect answer and deadlock

GHC is a parallel logic programming language [Ueda85]. A GHC program is a finite set of guarded clauses. A guarded clause has a form:

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

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

The clause can be read declaratively as follows:

For all term values of the variable in the clause,
H is true if both $G_1, ..., G_n$ and $B_1, ..., B_m$ are true.

Declarative reading regards guarded clauses as pure Horn clauses by ignoring operational things such as guards. In declarative debugging of Horn clause programs, the minimum models of the programs play a central role. However, the minimum model of a GHC program obtained by declarative reading of guarded clauses is insufficient for debugging errors heavily depending on control primitives such as guards. Examples are dealock and failure. In deadlock, pieces of a computation suspend, while correctness of suspending computations cannot be determined by the minimum model. In GHC, a computation fails because of incorrect choices of clauses even if there can be a successful computation. For handling these errors, it is necessary to extend the meaning of guarded clause programs.

Definition 2.1 *Given a progam S, Msuc and Msus are defined to be the following two sets.*

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

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

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

[Example 1.1]

```
p(a,Y) :- true | Y=b.
p(X,b) :- true | X=a.
p(c,Y) :- true ! true.
p(c,Y) :- true | q(Y).

q(d) :- true | true.
```

Msuc and *Msus* of the above program are:

Msuc = {p(a,b), p(c,Y), p(c,d), q(d)}
Msus = {p(X,Y), p(b,Y), p(X,a), p(c,Y), p(X,c), p(d,Y), p(X,d), q(X)}

Note that `p(c,Y)` is included in both *Muc* and *Mus*. In general, *Muc* and *Mus* can have non-empty intersection. This is because of the nondeterministic choice mechanism of GHC. *Mus* may include an unsatisfiable atom, which suspends due to insufficient instantiation. Elements of *Mus* such as `p(b,Y)` and `p(X,a)` are such examples.

Although there are several kinds of erroneous results that a bug may cause, in this section we concentrate on the two major ones which programmers often encounter. One is termination with incorrect answer and the other is deadlock. Handling of failure is studies in the next section.

Definition 2.2 *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.*

If $\leftarrow A$ succeeds with incorrect answer, then declarative method can be used. The problem is handling buggy deadlock.

When computation falls into an erroneous result, there is at least one bug in the history of the computation. History of computation is modelled by a computation tree in our framework.

Definition 2.3 *A computation tree of a goal $\leftarrow A$ is a proof tree of the goal.*

A Computation tree is 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 substrees can be regarded as independent computations.

A node in a computation tree corresponds to a procedure call, and its immediate descendants to goals in the right hand side of the clause invoked at the node. Let H be a node in the computation tree, and $G1, \ldots Gn$ and $B1, \ldots Bm$ be immediate descendants of H, where Gi's are guard goals and Bj's are body goals. Then $H :- G1, \ldots Gn|B1, \ldots Bm$ is an instance of the clause used at that node.

Correctness of each node in computation tree is defined with respect to the intended interpretation, *Muc* and *Mus*.

Definition 2.4 *We say a node $\leftarrow A$ in the computation tree has been executed correctly*

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

We assume throughout that guard goals are executed correctly. This is case for Flat GHC where only system predicates are allowed to call in a guard part.

Now we define two types of bugs of GHC programs.

Definition 2.5 *We say a clause instance.*

$$H :- G1, ..., Gn|B1, ...Bm.$$

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 $j = 1, ..., m$.

N.B. An incorrect clause instance can be declaratively correct. (Example shall be given later)

Definition 2.6 *We say a goal $\leftarrow A$ invoked at some node in the computation tree is buggy suspension if $\leftarrow A$ immediately suspends, but $A \notin Msus$.*

Because of our assumption that guard goals are always executed correctly, buggy suspension means that the procedure invoked by the goal is buggy.

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 to know whether a node $\leftarrow A$ in the 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: $A \in Msus$?

From the assumption that every guard is executed correctly, the debugger only queries oracle for nodes not included in any guard computation.

Now we give the description of the debugging algorithm for two types of errors, termination with incorrect answer and buggy deadlock.

Algorithm 2.1
Input: a computation tree whose root node has been incorrectly executed.
Output: an incorrect clause instance or buggy suspension.

Let T be the given computation tree.

1) Query the oracle for a root node of T whether the node has been executed correctly or not. (If T is the entire computation tree, then this step can be skipped.)

   2a) If the root has been executed incorrectly and has no descendant, then it is returned as buggy suspension.

   2b) Otherwise query the oracle for each node of immediate descendants of the root node whether it has been executed correctly or not.

3a) If there is no node that has been executed incorrectly, then the clause instance used here is returned as an incorrect clause instance.

3b) There is at least one node which has been executed incorrectly. Select one such node arbitrarily and go to 1) with the subtree rooted at the node as T.

The algorithm searches for a bug from the root of the computation tree using top-down strategy. It is possible to adapt the algorithm to the "divide and query" strategy.

For this algorithm, the following theorem holds.

Theorem 2.1 *Let P be a GHC program and G be a goal. Let T be the computation tree formed by the execution of the goal G. Suppose the execution of G succeeds or deadlocks and G' is the final form of G on T. If G' has not been executed correctly, then the above algorithm will terminate and return either an incorrect clause instance or a buggy suspension.*

In order to prove the theorem, we need the following lemma stating the basic property of a computation tree.

Lemma 2.1 *Let G' be the root node of the computation tree T. If G' has not been executed correctly, then there exists a bug in T (either an incorrect clause instance or buggy suspension).*

Proof: The proof is by induction on the height of T.

Suppose first that T consists of the single node G. If G succeeds then, since $G \notin \mathit{Mans}$, we have that the unit clause invoked at G is incorrect. If G suspends, then, since $G \notin \mathit{Mans}$, we have that G is a buggy suspension.

Suppose now that T has height $k+1$. If all the immediate descendants of the root node has been executed correctly, then the clause invoked at the root is incorrect. Otherwise, one of the immediate descendants has been executed incorrectly and we can apply the induction hypothesis to the subtree rooted at this immediate descendant which has height $\leq k$. ∎

Proof of Theorem 2.1: Similar induction argument to the proof of the lemma 2.1. ∎

According to the theorem, the algorithm can find a bug as long as the final form of the given goal has been executed incorrectly, that is, the bug is manifest at the root of the computation tree. In actual programs, it seems that the bug is manifest if the result of the computation differs from the intended one. However, there are strange cases in which the final form of the query is not the intended one, but has been executed correctly. Such strange examples are discussed in section 4.

3. Handling buggy failure

First we define the buggy failure

Definition 3.1 *The result of the computation is called buggy failure if the computation fails where it should not fail.*

The framework of debugging buggy failure is same as before, but we need a few additional definitions and a few extensions.

Definition 3.2 *Given a progam S, Mf is defined to be the following set.*

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

Now we call Muc, Mus and Mf together, the "intended interpretation" of a program.

In general, computation may be known to fail during the compuation, but it is necessary to complete the computation even in such cases for debugging buggy failure. The computation tree of failing computation must be a completed proof tree.

An immediately failing node and a failing node in the failing computation tree are defined as follows.

Definition 3.3 *A node in the computation tree is called an immediately failing node*

if *(a) it is incompatible unification, or*
(b) it does not succeed nor suspend, but has no descendant node.

The case (b) corresponds to the case where all the guard parts of the clauses fail.

Definition 3.4 *A node in the computation tree is called a failing node*

if *(a) it is an immediately failing node, or*
(b) it has immediately failing node in the subtree rooted at the node.

We say that a node in the computation tree has failed if and only if the node is a failing node. In the same way, we say that a node has immediately failed if and only if the node is an immediately failing node. The definition of "executed incorrectly" is extended as follows.

Definition 3.5 *(Extension of Definition 2.4)*
We say a node $\leftarrow A$ is the computation tree has been executed incorrectly

if *(a) $\leftarrow A$ suspends in the tree, but $A \notin Mus$, or*
(b) $\leftarrow A$ succeeds in the tree, but $A \notin Muc$, or
(c) $\leftarrow A$ fails in the tree, but $A \notin Mf$.

A new type of a bug is defined which causes computation to fail where it should not fail.

Definition 3.6 *We say a goal $\leftarrow A$ invoked at some node in the computation tree is an uncovered goal if $\leftarrow A$ immediately fails, but $A \notin Mf$.*

Oracle queries are extended as follows.

"*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: $A \in Msus$?
"*failed*(A)?"	if $\leftarrow A$ fails in the computation tree.
	Meaning: $A \in Mf$?

The description of the algorithm is almost the same as the previous one.

Algorithm 3.1
Input: a computation tree whose root node has been incorrectly executed.
Output: an incorrect clause instance, buggy suspension or an uncovered goal.

Let T be the given computation tree.

1) Query the oracle for a root node of T whether the node has been executed correctly or not. (If T is the entire computation tree, then this step can be skipped.)

   2a) If the root has been executed incorrectly and has no descendant, then it is returned as a buggy suspension or as an uncovered goal, depending on whether it suspends or fails.

   2b) Otherwise query the oracle for each node of immediate descendants of the root node whether it has been executed correctly or not.

      3a) If there is no node that has been executed incorrectly, then the clause instance used here is returned as an incorrect clause instance.

      3b) There is at least one node which has been executed incorrectly. Select one such node arbitrarily and go to 1) with the subtree rooted at the node as T.

We have the following theorem.

Theorem 3.1 *Let P be a GHC program and G be a goal. Let T be the computation tree formed by the execution of the goal G. Suppose the execution of G succeeds, deadlocks or fails and G' is the final form of G on T. If G' has not been executed correctly, then the above algorithm will terminate and return either an incorrect clause instance, buggy suspension or an uncovered goal.*

Proof: Proof is almost the same as that of theorem 2.1. ∎

Again what the theorem ensures is that the algorithm can find a bug as long as the final form of the given goal has been executed incorrectly, that is, the bug is manifest at the root of the computation tree. The problem is whether a bug is always manifest at the root of the computation tree or not. In the next, we examine strange cases in which the final form of the query is not the expected one, but is included in the intended interpretation.

4. Discussion

It is assumed so far that the bug will manifest at the root of the final computation tree when the computation of the given goal fails into the result different from the intended one. However, it is not always true. We show several strange examples which violate the assumption.

First we consider the case in which the goal is expected to succeed, but instead succeeds with correct but unexpected answer, suspends or fails. Let $\leftarrow A$ be a goal and $\leftarrow A'$ be an instance of $\leftarrow A$ in the final computation tree.

Case 1: $\leftarrow A$ succeeds with correct but unexpected answer.

> In this case, $A' \in Msuc$, but the computation where $\leftarrow A$ goes $\leftarrow A'$ is in no way included in the intended interpretation. Then, clearly, a bug does not manifest in the root of the final computation tree. The following example illustrates such case.

[Example 3.1]

```
p(X,Y) :- true | X=1, Y=1.        (1)
p(2,Y) :- true | Y=2.             (2)
```

Intension of the program is that `p(1,1)` and `p(2,2)` $\in Msuc$. The goals $\leftarrow$ `p(X,Y)` and $\leftarrow$ `p(2,Y)` are also expected to succeed with p(1,1) and p(2,2) as unique answer, respectively.

Suppose that the clause (1) has been incorrectly written as follows.

```
p(X,Y) :- true | X=2, Y=2.        (1')
```

Then the goal $\leftarrow$ `p(X,Y)` succeeds with p(2,2) while it is expected to succeed with p(1,1) as its unique answer. The final form of the goal, `p(2,2)`, is included in $Msuc$. Consequently the bug does not manifest at the root of the computation tree.

Case 2: $\leftarrow A$ suspends.

> Oracle states that $\leftarrow A$ should not suspend. In such case, it seems natural to assume that an instance of $\leftarrow A$, i.e., $\leftarrow A'$, should not suspend. Under this assumption, $A' \notin Msus$. Then the bug does manifest at root of computation tree. However, for a sufficiently weird program like the program below, the assumption may not hold.

[Example 3.2]

```
p(X) :- true | true.              (1)
p(a) :- true | G.                 (2)
```

where `G` is assumed to always suspend. Intension is that `p(X)` $\in Msuc$, `p(X)` $\notin Msus$, `p(a)` $\in$ $Msuc$ and $Msus$.

Suppose that the clause (1) has been incorrectly written as follows.

```
p(X) :- true | X=a, p(X).                                    (1')
```

Then the goal ← `p(X)` suspends while it is expected to succeed. The final form of the goal is `p(a)`, which is, however, included in *Mauz*. Consequently the bug does not manifest at the root of the computation tree.

Case 3: ← *A* fails.

[Example 3.3]

```
qsort(Xs,Ys) :- true | qsort(Xs,Ys,[]).                      (1)

qsort([X|Xs],Ys0,Ys2) :- true |
     part(Xs,X S,L), qsort(S,Ys0,[X|Ys1]), qsort(L,Ys1,Ys2).  (2)
qsort([],Ys,0) :- true | 0=Ys.                               (3)

part([X|Xs],A,S,L) :- A<X | L=[X|L1], part(Xs,A,S,L1).       (4)
part([X|Xs],A,S,L) :- A>=X | S=[X|S1], part(Xs,A,S1,L).      (5)
part([],_,S,L) :- true | S=[], L=[].                         (6)
```

The program above is GHC version of the well-known quicksort program. Suppose that the clause (1) has been incorrectly written as follows.

```
qsort(Xs,Ys) :- true | Ys=[], qsort(Xs,Ys,[]).               (1')
```

Then the goal ← `qsort([3,2,1],Y)` fails while it is expected to succeed. The final form of the goal is `qsort([3,2,1],[])`, which is, however, included in *Mf*. Consequently the bug does not manifest at the root of the computation tree.

Note that the clause (1') is declaratively correct. The clause describes the specific case, i.e., `Xs = []`. Procedurally incorrect usage of declaratively correct clauses often invokes erroneous result.

It may be able to conclude that the debugging algorithm for buggy failure probably is not very useful in practice because we would often expect a bug not to be manifest on the final computation tree.

Next we consider the case in which the goal is expected to suspend, but instead succeeds with correct answer.

The following example illustrates the case.

[Example 3.4]
Again we use `qsort` program described above.

Here the clause (1) is incorrectly written as follows.

```
qsort(Xs,Ys) :- true | Xs=[2,1], qsort(Xs,Ys,[]).            (1'')
```

Then the goal ← `qsort(X,Y)` succeeds with `qsort([2,1],[1,2])` while it is expected to suspend. The final form of the goal `qsort([2,1],[1,2])` is, however, included in *Msuc*. Consequently the bug does not manifest at the root of the computation tree.

These examples are enough to show the cases where the bug does not manifest at the root of the computation tree when the computation has falled into the result different from the expected one, though we have not consider all the cases.

4. Open Problems

We have presented the framework of debugging GHC programs and shown the debugging algorithm. It has been proved that the algorithm can find a bug as long as a bug manifests at the root of the computation tree. We have noticed the gap between all the buggy computations and the cases the algorithm can handle using several examples. Reducing this gap is the theme of further research. The algorithm for buggy failure should be imporved since it seems that a bug does not often manifest when the computation fails where it should not fail.

If the current algorithm is applied to a computation including a bug in the guard, then the algorithm only detects a bug on the surface of guard computation, i.e., in the level of a clause that has invoked the buggy guard computation. Handling of a bug in guard computation in finer level is another theme of further research.

In order to solve these problems, it appears necessary to look more closely at I/O behaviors during execution of a goal.

Acknowledgement

We would like to thank Kazuhiro Fuchi, Koichi Furukawa and all the other members of ICOT, both for help with this research and for providing a stimulating place in which to work.

References

[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

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

[Shapiro83] Shapiro, E., *Algorithmic Program Debugging*, MIT Press, 1983.

[Takeuchi86] Takeuchi, A., *Algorithmic Debugging of GHC programs and Its Implementation in GHC*, ICOT Technical Report TR-185, Institute for New Generation Computer Technology, 1986.

[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.