Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 174 (2007) 37–54

www.elsevier.com/locate/entcs

Dynamic Reverse Code Generation for Backward Execution

Jooyong Lee

BRICS,
Department of Computer Science,
University of Aarhus,
IT-parken, Aabogade 34,
DK-8200 Aarhus N, Denmark

Abstract

The need for backward execution in debuggers has been raised a number of times. Backward execution helps a user naturally think backwards and, in turn, easily locate the cause of a bug. Backward execution has been implemented mostly by state-saving or checkpointing, which are inherently not scalable. In this paper, we present a method to generate reverse code, so that backtracking can be performed by executing reverse code. The novelty of our work is that we generate reverse code on-the-fly, while running a debugger, which makes it possible to apply the method even to debugging multi-threaded programs.

Keywords: debugging, reverse execution, reverse code generation

1 Introduction

It has been pointed out in a number of papers that enabling backward execution in debuggers would be of great help in a debugging process [1,3,6,7]. A typical debugger-aided bug-finding, where a debugger does not support backward execution, is performed in iterative steps of: (1) guess a problematic point which may cause an unexpected behaviour of a program, and set a breakpoint there (2) restart a debugging session and watch a program state on the breakpoint. This procedure is time-consuming, not only because a user must repeat starting and stopping debugging sessions until finally identifying the cause of the error, but also because guesses made by a user are often not precise. What is worse, as a mainstream language like Java begins to support multi-threading, the traditional debugging procedure often even does not work because one cannot keep the scheduling order between threads the same as before, by only restarting a program. On the other hand, if a debugger

[1] Email: jlee@brics.dk

doi:10.1016/j.entcs.2006.12.028

can run a program backwards, a user can naturally see what happened in the past and, as a result, trace the error back to its source, not depending on an error-prone and time-consuming guess-restart procedure.

There have been several works that aim to support backward execution [2]. However most of them rely on state-saving or checkpointing, a periodic state-saving, and that makes their debuggers suffer from memory blow-up. Recently Akgul and Mooney suggested a way to generate reverse code by static analysis (control/data dependency analysis), and showed the memory efficiency of it [4]. We aim to generate reverse code in the same spirit as [4], but we also want to be able to deal with multi-threaded programs, unlike/in addition to [4]. To achieve the goal, we calculate reverse code on-the-fly, while a debugger is running, based on logged history of transitions, basically pointers to program locations, each of which may require only a few bits of information [3].

After introducing our input language and motivating example in the next two sections, we demonstrate in detail our reverse code generation method (Section 4). In the subsequent two sections (Section 5,6), we also explain auxiliary techniques necessary for reverse code generation. Then related work, discussion and conclusion come in order.

2 Input Language

We assume, as input, an imperative language that supports multi-threading, although as will be discussed in Section 8, we think functional languages can also benefit from dynamic reverse code generation.

The language grammar is described in the Extended Backus-Naur Form (EBNF), where regular expression operators such as ?, $*$ and $+$ are added to the BNF. Double quotation marks enclose keywords, and angle brackets enclose non-terminal symbols. We do not expand non-terminal symbols that are out of our concern, for example ⟨loc-id⟩, ⟨literal-exp⟩, ⟨boolean-type⟩ and so on.

Figure 1 shows the input language we use, which is a simplified form of BIR (Bogor Input Representation) [25]. BIR was originally designed as an intermediate language of a toolset called Bandera [12] which transforms a Java program to the equivalent program written in various kinds of model-checker specific languages such as PROMELA [20] of SPIN [19] and the SMV language [23]. More recently, BIR was revised as an input language to a model checker Bogor [24]. We chose BIR because memory-efficient backtracking will be beneficial not only to debuggers, but also to explicit model checkers.

Although full-fledged BIR is so expressive that a number of modern language features can be expressed seamlessly, at this initial stage of research, we want to focus on as simple a language as possible. In this spirit, we deal with a subset of BIR.

[2] Related work is shown in Section 7

[3] It is easy to decide how many bits are necessary since all program locations are known in static time.

```
          ⟨system⟩  ::=  “system” ⟨system-id⟩
                         “{” ⟨system-member⟩* “}”
   ⟨system-member⟩  ::=  ⟨const⟩ | ⟨global-var⟩ | ⟨fsm⟩
      ⟨global-var⟩  ::=  ⟨var⟩
             ⟨var⟩  ::=  “transient”? ⟨basic-type⟩
                         ⟨var-id⟩ ⟨var-init⟩? “;”
             ⟨fsm⟩  ::=  ⟨thread⟩
          ⟨thread⟩  ::=  (“active” (“[” ⟨num-active⟩ “]”)?)
                         “thread” ⟨thread-id⟩ “(” ⟨params⟩? “)”
                         “{” ⟨var⟩* ⟨body⟩ “}”
          ⟨params⟩  ::=  ⟨basic-type⟩ ⟨local-id⟩
                         (“,”⟨basic-type⟩ ⟨local-id⟩)*
            ⟨body⟩  ::=  ⟨location⟩+
        ⟨location⟩  ::=  “loc” ⟨loc-id⟩ “:” ⟨transformation⟩+
  ⟨transformation⟩  ::=  ⟨guard⟩? “do” ⟨visibility⟩?
                         “{” ⟨action⟩* “}” ⟨jump⟩ “;”
           ⟨guard⟩  ::=  “when” ⟨exp⟩
      ⟨visibility⟩  ::=  “visible” | “invisible”
          ⟨action⟩  ::=  ⟨assign-action⟩ | ⟨assert-action⟩
   ⟨assign-action⟩  ::=  ⟨var-exp⟩ “:=” ⟨exp⟩ “;”
   ⟨assert-action⟩  ::=  “assert” “(” ⟨exp⟩ “)” “;”
            ⟨jump⟩  ::=  “goto” ⟨loc-id⟩ | “return” ⟨local-id⟩
             ⟨exp⟩  ::=  ⟨literal-exp⟩ | ⟨var-exp⟩ | ⟨unary-exp⟩ |
                         ⟨binary-exp⟩ | ⟨paren-exp⟩ |
                         ⟨apply-exp⟩
             ⟨fun⟩  ::=  “fun” ⟨fun-id⟩ “(” ⟨fun-params⟩? “)”
                         “returns” ⟨basic-type⟩ “=” ⟨exp⟩ “;”
      ⟨fun-params⟩  ::=  ⟨basic-type⟩ ⟨fun-local-id⟩
                         (“,”⟨basic-type⟩ ⟨fun-local-id⟩)*
         ⟨var-exp⟩  ::=  ⟨var-id⟩
       ⟨unary-exp⟩  ::=  ⟨unary-op⟩ ⟨exp⟩
      ⟨binary-exp⟩  ::=  ⟨exp⟩ ⟨binary-op⟩ ⟨exp⟩
       ⟨paren-exp⟩  ::=  “(” ⟨exp⟩ “)”
       ⟨apply-exp⟩  ::=  ⟨fun-id⟩ “(” ⟨args⟩? “)”
        ⟨unary-op⟩  ::=  “+” | “-” | “!”
       ⟨binary-op⟩  ::=  “+” | “-” | “*” | “/”
      ⟨basic-type⟩  ::=  ⟨boolean-type⟩ | ⟨integer-type⟩
            ⟨args⟩  ::=  ⟨exp⟩ (“,” ⟨exp⟩)*
```

Fig. 1. Syntax of simplified BIR

To prevent confusion, we note a couple of differences between BIR and usual imperative languages.

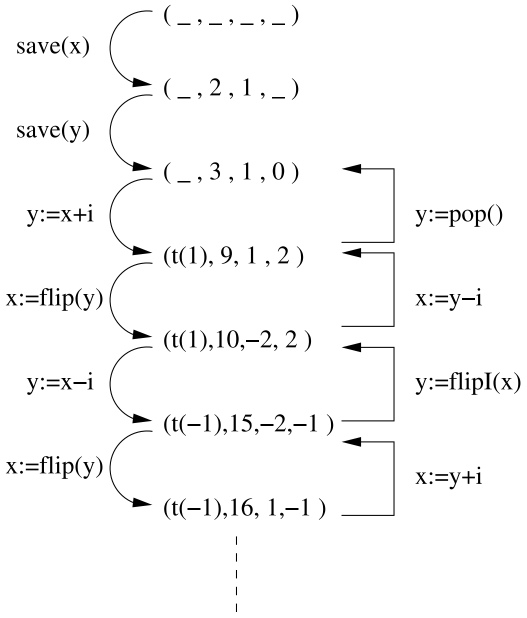
- A transformation consists of statements (action in BIR notation) that should be run simultaneously. The other threads cannot interfere until a transformation is completed.
- Control-flow of BIR follows one of guarded commands. Only transformations whose guards are valid can be executed. If more than one guard is valid, one of corresponding transformations are chosen non-deterministically.
- BIR allows a functional language style function definition whose body is a pure expression.

3 Motivating Example

```
system RevTest {
   int x := 1;
   int y := 0;

   thread t(int i) {
      loc loc0:
      when (x>0)
         do {
            y:=x+i;
            x:=flip(y);
         }
         goto loc0;
      when (x<=0)
         do {
            y:=x-i;
            x:=flip(y);
         }
         goto loc0;
   }
}
```

(a) A simple BIR program. flip(x) inverts a sign of x and is defined as -x.



(b) An execution trace instance, when two thread `t(1)` and `t(-1)` are active, and its forward (left) and backward (right) statements. flipI(x) is an inverse function of flip(x) and is defined as -x.

Fig. 2. In (b), a state is expressed as a tuple of thread id, transition id, variable x's value, and variable y's value.

Figure 2(a) is a simple BIR program that allows multiple non-terminating threads to run simultaneously, a typical case in multi-thread programs. Suppose that we provide backtracking by state-saving. Then we need to save every variable change as well as thread and location change. If we employ checkpointing, variable changes are saved less often, but inherently it shares the same memory blow-up problem with state-saving method. It also does not seem to be possible to generate a reverse program of Figure 2(a) by static dependency analysis, since nondeterminism between multiple threads imposes the absence of overall control flow. Note that in any methods above, past threads and locations should be saved for backtracking. The above observations motivated us to make a backtracking algorithm based on thread/location history.

We want to generate reverse code per each assignment while we are running a program on debugger. Figure 2(b) demonstrates what dynamic reverse code looks like. In the middle of Figure 2(b) are state flow when two threads `t(1)` and `t(-1)` are running simultaneously, where each state is denoted as a 4-tuple of a thread, a line number of the current location[4] and the values of x and y. Statements on the arc arrows show what statements are executed along with the state change. A special debugging command save is used to hold initial values of x and y in a LIFO manner. On the other hand, right-had side of the figure are dynamic reverse statements. Reverse statements is generated from the previous assignments and save commands executed ahead of the current location. For example, in order to come up with a reverse statement x:=y-i, we are using the previous assignment y:=x+i, exploiting the fact that y and i have not yet changed. Similiary a reverse statement y:=flipI(x), where flipI(x) is an inverse function of flip(x), is based on the previous assignment x:=flip(y). Another special debugging command pop is the counterpart of save to restore a value. Note that it is enough to save only initial values of two global variables and thread/location history. Previous assignments can be projected from the thread/location history. From the next sections, we will show in detail how we generate reverse code in runtime.

4 Reverse Code Generation

In this section, we suggest a framework where reverse code fragments are automatically computed at runtime. The basic idea is to log pointers to program locations and figure out the previous values of variables based on the current values of the variables and previously-run statements, which are available from program location history.

We demonstrate dynamic reverse code generation beginning with the definition of reverse code fragment and other necessary concepts.

Definition 1 (Transition)
A transition is a pair $\langle threadNum, transformation\rangle$*, where*

$$\begin{aligned} threadNum &\in \mathbb{Z} \\ transformation &\in Transformation[system] \end{aligned}$$

A distinct $threadNum$ *is assigned to each instance of thread.*

Definition 2 (Reverse code fragment)
A reverse code is a triple $\langle transition, rpoint, rstmt\rangle$*, where*

$$\begin{aligned} rpoint &\in Loc[system]; \textit{ denotes a reverse point} \\ rstmt &: \textit{a sequence of assignments; denotes a reverse statement} \end{aligned}$$

By $Transformation[system]$ we denote a set of the semantic counterparts of ⟨transformation⟩'s that appear in a given system. Similarly, $Loc[system]$ is a set of semantic counterparts of ⟨location⟩'s of a given system. $Assignment$ is a set of assignments whose syntax follow ⟨assign-action⟩ of BIR.

[4] The current location is the line number of the last statement.

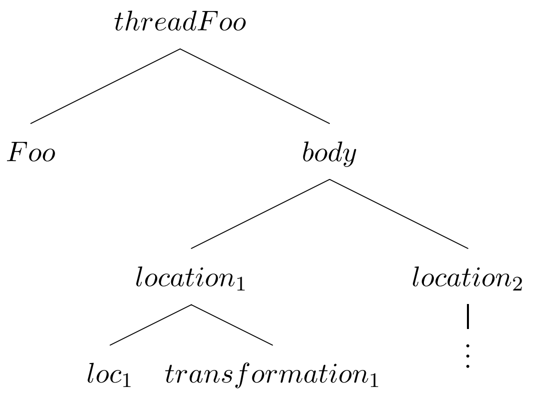


Fig. 3. A part of an abstract syntax tree of a simplified BIR program that contains a thread *Foo*

A reverse code $\langle t, rp, rs \rangle$ denotes that if t is the last transition executed, the previous program location is rp, and rs should be executed to restore the previous values. Our framework infers rp and rs based on a stack of transitions. To make the argument clear, we define a backtracking stack as follows:

Definition 3 (Backtracking stack)
A backtracking stack *is a sequence of transitions* $t_1 t_2 \ldots t_n$*, where* $n \geq 1$*, placed in order of execution.*

In the next two subsections, we show how the backtracking stack is used to infer reverse point and reverse statement.

4.1 Inferring a Reverse Point

A program written in a structured language like BIR is usually maintained as an abstract syntax tree inside an interpreter. Figure 3 shows a snippet of an abstract syntax tree of a simplified BIR program.

Once the last transition t is fetched from the backtracking stack, by $threadNum$ of t a thread to be backtracked is decided, and another element of t, $transformation$ can be used to infer the previous program location along with an abstract syntax tree of a given program. For example, in Figure 3, it is straightforward to see that the previous location of $transformation_1$ is $location_1$ and its identification is loc_1.

4.2 Inferring a Reverse Statement

In our simplified BIR, only assignments[5] change the state except for program locations. Hence we focus on an assignment trace defined as below:

Definition 4 (Assignment trace)
An assignment trace *is an ordered sequence of assignments projected from a backtracking stack. We assume that no local variables in different thread instances use the same name. For example, they can be indexed by the thread id.*

[5] corresponding to ⟨assign-action⟩ in Figure 1

The right hand side of an assignment that has no more than one variable can be expressed as a function application in the following way:

Definition 5 (Assignment)
An assignment with no more than one variable on its right-hand side is described in the following syntactic form.

$$x := (\lambda p.s)y \quad s.t.\ FV(\lambda p.s) = \emptyset$$

$$\begin{aligned} x, p, y \in& Var \\ s \in& FoExp \end{aligned}$$

where,

$$\begin{aligned} Var ::=&\ \textit{a set of variables} \\ FoExp ::=& Var \mid \mathbb{Z} \mid FoExp\ op\ FoExp \mid \\ & (\lambda Var.FoExp)FoExp \\ op ::=& + \mid - \mid * \mid / \end{aligned}$$

$FV(s)$ returns a set of free variables of a term s. Often we will give a name to a lambda abstraction, hence $(\lambda p.s)y$ will be denoted in form $e(y)$, where e is a function that takes one argument.

For example, $x := y + 1$ can be viewed as $x := (\lambda p.p + 1)y$, or $x := f(y)$ when $f \overset{def}{=} \lambda p.s \wedge FV(f) = \emptyset$. However Definition 5 is not immediately applicable to an assignment whose right hand side contains more than one variable. For instance, $x := p + q$ cannot be defined in the form of Definition 5. For the moment, let us restrict to the case where the right hand side of an assignment depends on at most one variable. We will expand our consideration to multiple-variable case in Section 4.2.1.

Now consider the following assignment trace fragment under our consideration. Assignments are labeled for the sake of explanation.

$$\ldots \to [x := e_1(y)]^1 \xrightarrow{*} [y := e_2(z)]^2 \to \ldots$$

Let us assume that we are about to backtrack over the assignment labeled with "2". Provided that variable y is not defined between assignment "1" and "2", we can restore the old value of y by executing $y := e_1^{-1}(x)$ where $\forall y \in \mathbb{Z} : e_1^{-1} \circ e_1(y) = y$ under the assumption that x can be recovered to the value as it was just after executing assignment one. In other words, $y := e_1^{-1}(x)$ constitutes a reverse statement corresponding to a transformation that contains assignment two. We will call that kind of assignment a inverse assignment (IA). We capture the above description on the following diagram, where a function *restore*[6] recovers a value of x to the value as it was immediately behind assignment one.

$$\begin{array}{lll} \ldots \to [x := e_1(y)]^1 \xrightarrow{*} [y := e_2(z)]^2 & & \to \ldots \\ \qquad\qquad \ldots \leftarrow [y := e_1^{-1}(restore(x))]^2 & & \leftarrow \ldots \end{array}$$

From now on, we will generalize the above observation. What we want is to obtain an inverse assignment for each assignment in a transformation. The first step is to find a reminiscent assignment defined as below:

Definition 6 (Reminiscent assignment (RA))
A reminiscent assignment ra *of an assignment* $y := rhs$ *must satisfy the following three conditions in a given assignment trace* at*.*

(i) ra *occurs no later than* $y := rhs$ *in* at*,*

(ii) y *appears in the right-hand side of* ra*, and*

(iii) y *is not defined between* ra *and the assignment* $y := rhs$ *in* at*, including* ra *and excluding the assignment* $y := rhs$*.*

In the previous example, the RA of assignment two is $x := e_1(y)$. Note that if an assignment is self-defined as in the form of $x := e(x)$, then the RA of it is trivially itself. Also note that, due to the third condition, unless an assignment α is self-defined, the RA of α cannot have the same variable on both sides of the RA.

As illustrated before, an RA can be used to restore the previous value of a variable. However, not every RA has such an ability. Even after we have found an RA $x := e(y)$, only when e^{-1} is known or derivable is the RA of use. We name such an RA a reversible RA, which is defined as below:

Definition 7 (Reversible RA (RRA))
A reversible reminiscent assignment $ra \equiv x := e(y)$ *of an assignment* $y := rhs$ *must satisfy the following conditions.*

- *all three conditions described in Definition 6, and*
- *an inverse function* e^{-1} *is given or derivable.*

An inverse function may be given by a user, or more preferably often it can be derived in an algorithmic way. In Section 6, we will depict how we can derive an inverse function from an assignment with an arithmetic expression on its right hand side.

There may exist multiple RRA's, and in which case we simply choose the last one in a given assignment trace. On the other hand, there may be no RRA. A variable may be redefined before being used in later assignments, or even if there exists an RA, it may not be reversible. At the moment, we simply store the previous

[6] See Section 4.2.2.

value of a variable if no RRA exists[7]. The previous values of variables are stored in a value stack defined as follows:

Definition 8 (Value stack)
A value stack *is a sequence of $\langle var, val\rangle$'s placed in order of generation, where*

$$\begin{aligned} &var \in \text{a set of variables} \\ &val \in \mathbb{Z} \end{aligned}$$

If an RRA is found, the next step to construct an inverse assignment is to apply an inverse function that matches the function on the right hand side of the RRA. An inverse assignment with regard to an RRA is defined as follows:

Definition 9 (Inverse Assignment (IA) w.r.t. RRA)
If $x := e(y)$ is an RRA of $y := rhs$, then the inverse assignment *of $y := rhs$ is:*

$$y := e^{-1}(restore(x))$$

where a function $restore(x)$, which will be defined in Section 4.2.2, returns the value of x as it was immediately after the RRA was executed.

If an RRA does not exist, the previous value to be restored should be kept in a value stack, and an IA for that variable is to get the saved value. In the following, we define an IA when no RRA is available using a helper function *pop*.

Definition 10 (A function $pop(var, n, vs)$)
A function $pop(var, n, vs)$, where $n \in \mathbb{N}$, returns the value associated to the nth occurring of a variable var in a value stack vs. For example, $pop(x, 2, [\langle x, 1\rangle, \langle y, 2\rangle, \langle x, 0\rangle])$ yields 1.

Definition 11 (Inverse Assignment (IA) w.r.t. value stack)
If an assignment $y := rhs$ is related to no RRA, an inverse assignment of $y := rhs$ is:

$$y := pop(y, 1, vs)$$

Now we are ready to make a reverse statement. Suppose that a transformation contains a sequence of assignments $a_1, a_2, \ldots, a_n$ (we ignore other types of statements), then we can calculate an inverse assignment ia_i for each assignment a_i. Now a reverse statement is $ia_1, ia_2, \ldots, ia_n$.

4.2.1 Multiple Argument Expression

Until now we assumed that the right hand side of an assignment contains at most one variable. Extending it to multiple occurrences requires only a modest change which is described in this subsection.

[7] We believe there would be other ways to avoid state-saving like the use of reaching definition suggested in [4].

Suppose that the right hand side of an assignment is $y + z$. In a lambda expression, $((\lambda p.\lambda q.p + q)y)z$ or $((\lambda q.\lambda p.p + q)z)y$, which becomes, respectively, $(\lambda p.y + p)z$ and $(\lambda p.p + z)y$. Now they are in accordance with Definition 5 except that the set of free variables of each lambda abstraction is not empty. For example, $FV(\lambda q.y + q) = \{y\}$.

In order to allow multiple variables on the right hand side of an assignment, we extend Definition 5 as below:

Definition 12 (Assignment with restriction)

$$x := (\lambda p.s)y/\hat{r} \quad s.t.\ FV(\lambda p.s) = \{x \mid x \in \hat{r}\}$$

$$\begin{aligned} \hat{r} &\in 2^{Var-\{y\}} \\ x, p, y &\in Var \\ s &\in FoExp \end{aligned}$$

$(\lambda p.s)y/\hat{r}$ denotes $(\lambda p.s)y$ where each $v \in \hat{r}$ is to be substituted with the value it denotes when the assignment is about to be executed. Var and $FoExp$ are defined in the same way as in Definition 5.

Note that several syntactic forms of assignment are possible when more than one variable is used in the right hand side of an assignment. In the above example, we can view an assignment $x := y + z$ as either $x := (\lambda p.c_y + p)z$ or $x := (\lambda p.p + c_z)y$, where c_y and c_z, respectively represent the value of y and z at the assignment. What form is appropriate depends on what variable we want to restore. In a general context, the following mutation property holds:

Property 1 (Mutation)
Given an assignment $x := (\lambda p.s)y/\hat{r}$, the right hand side can be transformed according to the following rule:

$$(\lambda p.(s[y/p])[p/r])r/(\hat{r} - \{r\}) \cup \{y\}$$

where $s[a/b]$ means that a is substituted for b in s.

Now we revise Definition 9 as follows:

Definition 13 (IA w.r.t. RRA (revised))
If $x := e(y)/\hat{r}$ is an RRA of $y := rhs$, then the IA of $y := rhs$ is:

$$y := e^{-1}(restore(x))[\forall r \in \hat{r} : restore(r)/r]$$

4.2.2 Restore Function

In this section, we illustrate the *restore* function used in Definition 9 and 13. Function application $restore(x)$ actually also carries on previous-value-inference procedure for x based on an assignment trace. In the following, we show an algorithm for the *restore* function, along with one more definition used in the algorithm.

Definition 14 (Assignment freezing)
If we say to freeze an assignment, mutation (Property 1) on the right-hand side of the assignment is prohibited.

Algorithm 1 (Restore)

(1) *Let a_1 and a_2, respectively, be the assignment we want to backtrack and the RRA of a_1.*

(2) *Suppose that the IA corresponding to a_2 is:*

$$y := e_1^{-1}(restore(x))[\forall r \in \hat{r} : restore(r)/r]$$

(3) *Freeze a_2.*

(4) *Let us define a set $S = \{x\} \cup \{r \mid \in \hat{r}\}$.*

(5) *For all $x \in S$, we perform $restore(x)$, beginning with checking if x is redefined (x's value is changed) between a_2 and a_1.*

  (a) *If redefined, let the redefining assignment be a_r, and search for the RRA of a_r.*

   (i) If an RRA $z := e_2(x)/\hat{r'}$ is found, $restore(x)$ returns the following:

$$e_2^{-1}(restore(z))[\forall r \in \hat{r'} : restore(r)/r]$$

   Then repeat the procedure from (3), setting a_2 to the newly-found RRA while leaving a_1 the same as before.

   (ii) If no RRA is found, return $pop(x, n, vs)$, where n represents the number of redefining assignments between a_2 and a_1, and vs denotes a value stack.

  (b) *If not redefined, return the current value of x.*

Property 2 (Termination)
Algorithm 1 always terminate.

Proof. In (a), we only need to look into assignments between a_r and a_2 (including a_r) because no RRA for variable y does not allow redefinition of y (see Definition 6). Therefore a_2 approaches a_1 and, in turn, the above procedure always terminates.□

Property 3 (Correctness)
Algorithm 1 always returns a correct value.

Proof. If we assume that we save all variable values that cannot be restored by RRA's[8], it is enough to show, whenever an RRA of a variable y exists, the following

[8] See Section 5

relation holds:

$$restore(x) = e(y_p)[\forall r \in \hat{r} : restore(r)/r]$$

where y_p denotes the previous value of y. And that is true by induction. □

4.3 Analysis

It is straightforward to see that reverse point is inferred in constant time. However reverse statement calculation requires more than that. Let n be the number of assignments between an assignment a_1 to be reverted and its RRA a_2. Then it requires $O(n)$ to attain the first IA. If we succeed to find an IA and the IA induces restoration of other sub-variables, each restoration process takes $O(n)$ (note that we look into assignments only between the redefinition of a sub-variable and a_2). This iterative process can take place $O(n)$ times. Therefore if we denote the maximum number of sub-variables during iterative process by m, the overall cost is $O(m^n \cdot n)$. In worst case, n can increase up to the entire execution length, but practically n tends to be small enough owing to locality of variable accesses.

5 Selective Store

In the previous sections, we assumed that we store a variable value only if it cannot be restored through a corresponding RRA. This section is about how we implement such a selective store.

Figure 4 exhibits how a selective store can be built while executing each action. If two successive tests, one for the check of self-defined RRA (line 8) and the other one for the search for an RRA in an assignment trace (line 9), fail, the current value of an assignment variable (the value before the assignment is performed) is stored in a value stack (line 10).

```
1  global $atrace$ #assignment trace#
2  proc forward($t$:Transition)
3    let $\langle thNum, trf\rangle \equiv t$
4    $actions$ = getActions($trf$)
5    for each action $a$ in $actions$:
6      if $a$ is an assignment action:
7        let $(x := e(y)) \equiv a$
8        if !isRRA($a$):
9          if !existRRAof($a$,$atrace$):
10           store($\langle x, val(x)\rangle$)
11     execute($a$)
```

Fig. 4. forward execution procedure performing selective store on previous variable values.

```
fun isRRA($a$)
  let $(x := e(y)) \equiv a$
  if $x = y$:
    return invertible($a$):
  else:
    return false
```

Fig. 5. isRRA function used in Figure 4, returning true if a given assignment is a self-defined RRA, and false otherwise.

```
 1  fun existRRAof($a$,$trace$)
 2    if empty($trace$):
 3      return false
 4    else:
 5      let $(x := e_y(y)) \equiv a$
 6      let $(p := e_q(q)) \equiv$ last($trace$)
 7      if $p = x$:
 8        return false
 9      else if $q = x$:
10        if invertible($p := e_q(q)$):
11          return true
12        else: existRRAof($a$,$trace$−last($trace$))
13      else: existRRAof($a$,$trace$−last($trace$))
```

Fig. 6. existRRAof function used in Figure 4, returning true if there exists at least one RRA in a given assignment trace, otherwise false.

Two functions, isRRA and existRRAof, used in Figure 4 are separately depicted in Figure 5 and Figure 6. Recall that isRRA(α) checks if a given α is a self-defined RRA, and existRRAof(α,$trace$) looks through assignment trace $trace$ to see the existence of the RRA of α. In order for an assignment itself to be an RRA, both sides of the assignment should be expressed with the same variable (see line 3 of Figure 5), and the assignment should be invertible (there should exist an inverse function corresponding to the right-hand side of an assignment). Meanwhile, if an RRA is to be one of the previously executed assignments, the assignment variable must be used in an RRA (see line 9 of Figure 6), and the RRA should be invertible. If no RRA is found until all elements of a given assignment trace are searched through, or while looking through an assignment trace, we hit an assignment whose assignment variable is the same as the variable to be restored (line 7 of Figure 6), we judge no RRA exists.

6 Derivation of Inverse Functions

Throughout the argument, we relied on inverse functions to achieve value restoration. Inverse functions may be either given by a user or more preferably derived automatically. As mentioned in a later section, derivation of inverse functions is another active research topic. However since we express an arithmetic expression as a function format, we suggest a way to derive an inverse function from it.

Figure 7 lists base rules applicable when a function argument is not under another subexpression. Each rule is associated with a constraint to exclude expressions which are impossible or infeasible to derive inverse functions, and some of constraints require more than static lexical checking. Constraints of (3) and (4) look into the assignment variable of RRA in runtime. Meanwhile modular conditions in (7) and (8) need to be flagged when assignments containing division is executed, so that cached modular conditions are to be available when those assignments are used as RRA's.

In the case a function argument variable is inside a subexpression, we apply the expansion rules displayed in Figure 8 until the last rule is applied. These expansion rules basically inverse subexpressions and hold them between angle brackets. Then a substitution rule shown in Figure 9 put together the inverse subexpressions by

$$\begin{aligned}
(\lambda x.x+q)^{-1} &= \lambda x.x-q && \text{if } x \notin FV(q) && (1)\\
(\lambda x.p+x)^{-1} &= \lambda x.x-p && \text{if } x \notin FV(p) && (2)\\
(\lambda x.x\times q)^{-1} &= \lambda x.x/q && \text{if } x \notin FV(q) \wedge y \neq 0 \text{ when an RRA is } y = x\times q && (3)\\
(\lambda x.p\times x)^{-1} &= \lambda x.x/p && \text{if } x \notin FV(p) \wedge y \neq 0 \text{ when an RRA is } y = p\times x && (4)\\
(\lambda x.x-q)^{-1} &= \lambda x.x+q && \text{if } x \notin FV(q) && (5)\\
(\lambda x.p-x)^{-1} &= \lambda x.p-x && \text{if } x \notin FV(p) && (6)\\
(\lambda x.x/q)^{-1} &= \lambda x.x\times q && \text{if } x \notin FV(q) \wedge x\%q = 0 && (7)\\
(\lambda x.p/x)^{-1} &= \lambda x.p/x && \text{if } x \notin FV(p) \wedge p\%x = 0 && (8)
\end{aligned}$$

Fig. 7. Base rules for inverse function derivation

applying the rule until no element is left between the angle brackets.

$$\frac{(\lambda x.op_1 \circ op_2)^{-1}\langle L\rangle \quad x \in FV(op_1) \quad op_1 \neq x \quad x \notin FV(op_2)}{(\lambda x.op_1)^{-1}\langle L, (\lambda x.x \circ op_2)^{-1}\rangle}$$

$$\frac{(\lambda x.op_1 \circ op_2)^{-1}\langle L\rangle \quad x \notin FV(op_1) \quad op_2 \neq x \quad x \in FV(op_2)}{(\lambda x.op_2)^{-1}\langle L, (\lambda x.op_1 \circ x)^{-1}\rangle}$$

$$\frac{(\lambda x.op_1 \circ op_2)^{-1}\langle \ldots, (\lambda x.x \circ op)^{-1}, \ldots\rangle}{(\lambda x.op_1 \circ op_2)^{-1}\langle \ldots, \lambda x.s, \ldots\rangle}$$

$$\frac{(\lambda x.op_1 \circ op_2)^{-1}\langle \ldots, (\lambda x.op \circ x)^{-1}, \ldots\rangle}{(\lambda x.op_1 \circ op_2)^{-1}\langle \ldots, \lambda x.s, \ldots\rangle}$$

$$\frac{(\lambda x.op_1 \circ op_2)^{-1}\langle L\rangle \quad op_i = x \quad x \notin FV(op_{i\%2+1})}{\lambda x.s\langle L\rangle}$$

Fig. 8. Expansion rules for deriving inverse functions. A symbol $\circ$ represents $+, -, *$ or $/$, and the function body s is constructed according to the base rules in Figure 7.

$$\frac{\lambda x.s\langle l_1, l_2, \ldots, l_{n-1}, l_n\rangle}{\lambda x.s[l_n x/x]\langle l_1, l_2, \ldots, l_{n-1}\rangle}$$

Fig. 9. Substitution rule for deriving inverse functions

Example 1

Trace of inverse function derivation for $\lambda x.((p * x) + (r * s))/t$

$$
\begin{aligned}
&(\lambda x.((p * x) + (r * s))/t)^{-1}\langle\rangle \\
&\longmapsto (\lambda x.(p * x) + (r * s))^{-1}\langle(\lambda x.x/t)^{-1}\rangle \\
&\longmapsto (\lambda x.(p * x) + (r * s))^{-1}\langle\lambda x.x * t\rangle \\
&\longmapsto (\lambda x.p * x)^{-1}\langle\lambda x.x * t, (\lambda x.x + (r * s))^{-1}\rangle \\
&\longmapsto (\lambda x.p * x)^{-1}\langle\lambda x.x * t, \lambda x.x - (r * s)\rangle \\
&\longmapsto \lambda x.x/p\langle\lambda x.x * t, \lambda x.x - (r * s)\rangle \\
&\longmapsto \lambda x.(x - (r * s))/p\langle\lambda x.x * t\rangle \\
&\longmapsto \lambda x.((x * t) - (r * s))/p
\end{aligned}
$$

The derivation rules shown here do not cover every case. If we fail to get an inverse function, we rely on state-saving.

7 Related Work

There have been a number of attempts to provide reverse execution for debuggers. The easiest way to achieve reverse execution is to save program locations and old values that may be necessary in running a program backwards [2,11,26]. A clear drawback of this *state-saving* approach is that the amount of data to be saved grows very high easily as a program runs.

More often, reverse execution is simulated by reexecuting a program until the earlier point. It appears to work well with small size program, but it cannot avoid suffering from increasing time overhead as a program size gets bigger. Hence the reexecution idea often comes in tandem with *checkpointing*. Old values are saved per periodic checkpoints, not per every statement or instruction, and when we want to backtrack, first we go back to the closest earlier checkpoint from the position we want to jump back to, and then reexecute the remaining part down to the destination point [1,8]. For example, [1] sets checkpoints on borders of control structures of a program, such as the beginning and end of `if` or `while` statements. However, essentially checkpointing and reexecution also cannot avoid memory accumulation because memory is consumed every checkpoint.

Lastly, there is a way to run a program backwards with *reverse code* as presented in [4,6,9] and this paper. Reverse code rids us of heavy dependencies on state saving. A big challenge in this approach is how to generate reverse code. No existing method can fully generate reverse code, and if reverse code fails to be generated, state-saving takes care of reverse execution. [6,9] generate reverse code only for self-defined assignments (e.g. reverse code of `x:=x+1` is `x:=x-1`), and [4] produces reverse code based on static analysis result for data dependency (for value retrieval) and control dependency (for location retrieval). Currently [4] confines its use to a single threaded program, and its extension to multi-threaded programs would require a dramatic cost increase of static analysis.

8 Discussion

Through this paper, we have used inverse functions to generate reverse code, and we have presented how to generate an inverse function whose function body consists of an arithmetic expression. For more general functions, we hope to exploit research result about program inversion. The origin of program inversion is Dijkstra's short note [13]. Program inversion computes an inverse program p^{-1} of a given program p with relation of $y = p(x)$ if and only if $p^{-1}(y) = x$. Note that if we apply program inversion to a function, we can obtain a corresponding inverse function.

There have been several works that can infer an inverse program based on annotations of a pair of pre/post conditions around program constructs [10,13,18]. Most research about automatic program inversion was pursued for functional programming programming language. [14,17,22] calculates inverse functions for restricted form of first-order functions. Recently Glück and Kawabe are working on this area [15,16,21].

Meanwhile there is a limitation on the extent of statements that can benefit from the presented method. Most heap update statements are not invertible. For example, although `o:=o.f` is self-defined assignment [9], it does not give a clue about the previous value of `o`. Note that the same limitation is shared by [4] that also generates reverse code, based on control/data dependency analysis result in static time, as explained in Section 7.

We conjecture that our reverse code generation method may fit better with functional programming languages, although we demonstrated it in imperative programming language. In functional programming language, program text provides more clues about heap update through, e.g., pattern matching.

Backtracking plays an important role in many areas in computer science besides debugging. Simulation, model checking, theorem proving and logic programming are a few of them. And more often than not, efficient memory usage is one of the critical issues in those areas. We hope the backtracking method presented here can help alleviate memory blow-ups in other areas too.

9 Conclusions

We have presented a reverse code generation method that can be used for reverse execution while debugging. The novelty of our work is that we generate reverse code on-the-fly based on a logged history of transitions, which are basically pointers to program locations. This dynamic generation makes it possible to be get reverse code even for multi-threaded programs.

One possible future work is to see how one can benefit from dynamic slicing as in [2,5]. We suppose it will help improve reverse code generation time by removing unnecessary assignments from an assignment trace. There are also a number of engineering issues. For example, it may be possible to calculate reverse code in the background while tracing a program forward step by step in a debugger, and

[9] `f` is a field of `o` whose type is the same as `o`.

have reverse code ready when one needs to backtrack. Or checkpointing could be employed and reverse code should only be generated between checkpoints. We can also cache reverse code and reuse it later if variable update is only locally affected. Lastly, as we mentioned in the previous section, we are also interested in extending this work to functional programming languages.

Acknowledgement

The author would like to thank especially Mogens Nielsen and Olivier Danvy for their valuable comments and advice.

References

[1] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. An execution backtracking approach to program debugging. *IEEE Software*, pages 21–26, 1991.

[2] Hiralal Agrawal, Richard A. Demillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software-Practice and Experience*, 23(6):589–616, 1993.

[3] Tankut Akgul and Vincent J. Mooney III. Instruction-level reverse execution for debugging. *Workshop on Program Analysis For Software Tools and Engineering 2002*, 2002.

[4] Tankut Akgul and Vincent J. Mooney III. Instruction-level reverse execution for debugging. Technical Report GIT-CC-02-49, Georgia Institute of Technology, September 2002.

[5] Tankut Akgul, Vincent J. Mooney III, and Santosh Pande. A fast assembly level reverse execution method via dynamic slicing. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, 2004.

[6] Bitan Biswas and R. Mall. Reverse execution of programs. *SIGPLAN Notices*, 34(4):61–69, 1999.

[7] Simon P Booth and Simon B Jones. Walk backwards to happiness - debugging by time travel. *Automated and Algorithmic Debugging*, pages 171–183, 1997.

[8] Bob Boothe. Efficient algorithms for bidirectional debugging. *ACM SIGPLAN Notices*, 35(5):299–310, May 2000.

[9] Christopher D. Carothers, Kalyan S. Perumalla, and Richard M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, July 1999.

[10] Wei Chen and Jan Tijmen Udding. Program inversion: more than fun! *Science of Computer Programming*, 15:1–13, November 1990.

[11] Jonathan J. Cook. Reverse execution of java bytecode. *The Computer Journal*, 45:608–619, 2002.

[12] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from java source code. *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[13] Edsger W. Dijkstra. Program inversion. In *Program Construction, International Summer School*, volume 69 of *LNCS*, pages 54–57. Springer-Verlag, 1978.

[14] David Eppstein. A heuristic approach to program inversion. In *International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 219–221, 1985.

[15] Robert Glück and Masahiko Kawabe. A program inverter for a functional language with equality and constructors. *Asian Symposium on Programming Languages and Systems*, pages 246–264, 2003.

[16] Robert Glück and Masahiko Kawabe. Derivation of deterministic inverse programs based on LR parsing. *Proceedings of Functional and Logic Programming: 7th International Symposium*, pages 291–306, March 2004.

[17] Robert Glück and Masahiko Kawabe. Revisiting an automatic program inverter for Lisp. *The Third Workshop on Programmable Structured Documents*, January 2005.

[18] David Gries. Inverting programs. In David Gries, editor, *The Science of Programming*, Monographs in Computer Science, chapter 21, pages 265–274. Springer-Verlag, 1981.

[19] Gerald.J. Holzmann. The model checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[20] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.

[21] Masahiko Kawabe and Robert Glück. The program inverter LRinv and its structure. *Practical Aspects of Declarative Languages*, pages 219–234, 2005.

[22] Richard E. Korf. Inversion of applicative programs. In *International Joint Conference on Artificial Intelligence (IJCAI-81)*, pages 1007–1009, 1981.

[23] K. L. McMillan. *The SMV language.* http://www.cis.ksu.edu/santos/smv-doc/.

[24] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, pages 267–276, 2003.

[25] SAnToS Laboratory. *Bogor Software Model Checking Framework: User Manual*, March 2005. http://bogor.projects.cis.ksu.edu/.

[26] Marvin V. Zelkowitz. *Reversible Execution as a Diagnositc Tool.* PhD thesis, Department of Computer Science, Cornell University, 1971.