平成 11 年度冬学期 第 6 回情報工学輪講資料

デバッグのためのプログラム逆実行

寺田研究室　修士 1 年　丸山 一貴*

11-22-1999

[出典]

Henry Lieberman and Christopher Fry,
"ZStep 95: A Reversible, Animated Source Code Stepper",
in *Software Visualization: Programming as a Multimedia Experience*,
John Stasko, John Domingue, Blaine Price, Marc Brown, eds.,
MIT Press, 1997.

Bitan Biswas and R. Mall,
"Reverse Execution of Programs",
ACM SIGPLAN Notices Vol.34, No.4, 1999.

要旨

プログラマのデバッグ作業を支援するソフトウェアとして、デバッガがある。デバッガは主に、(1) プログラムを指定したところで停止させる、(2) プログラムを一行ずつ実行する、(3) 停止した場所での変数の値を表示・変更する、といった機能を持つ。「どのコードが間違いを引き起こしているか」を決定するためには、(1) や (2) のような実行制御は欠かすことができない。

しかしながら、現在のデバッガはプログラムを通常実行と同じように順方向にしか実行させることしかできないため、正しいと思い込んで、あるいは操作ミスにより、バグの現場を通り過ぎてしまうことがある。こうした場合に、デバッガによってプログラムを逆方向に実行して見せる機能が提供されることは、デバッグ支援として非常に有効である。

本輪講では、逐次処理のプログラムを対象として、このような「プログラム逆実行」や「プログラム実行履歴の再現」を実現するいくつかの研究について紹介する。

1　はじめに

プログラミング作業においてバグはつきものであり、多くのバグは実行時に発見される。こうした「コンパイル時の静的解析では分からないバグ」の解析を補助するツールとしてデバッガがある。

デバッガはデバッグ対象プログラム (デバッギ) のメモリ空間にアクセスすることが可能である。例えば、命令列の一部に特殊な命令を挿入することにより「ブレークポイント」が実現できるし、ある時点での変数の値を調べたり変更したりすることができる [1]。

デバッガを用いたデバッグ作業は、およそ以下の手順になると言える：

1. 誤りの発覚する場所にブレークポイントを設定し、プログラムをそこまで実行する
2. その場所での変数の値を調べる
3. 原因となっていそうなところの直前にブレークポイントを設定し、プログラムを最初からそこまで実行し直す
4. プログラムを一行ずつ動かしながら、変数の値などを調べる

*kazutaka@sanpo.t.u-tokyo.ac.jp

このように、プログラムを任意の場所で停止させ、その時点での実行状態を調べるのが基本となる。従って、デバッガにおいてデバッギの実行を制御できることは極めて重要である。

しかしながら、通常のデバッガはプログラムをその実行順通りにしか動かすことができず、現在位置の直前の情報を得るためにさえ、上記 3. にあるように、手動で最初から実行し直さなくてはならない。

そこで、「プログラムの逆実行」を可能にしようという試みがある。本輪講では、逆実行を実現しているいくつかの研究について、その手法を紹介する。

2 プログラムの逆実行

逆実行が実現できると、例えば以下のような場合に有効である：

- 以前のプログラムの実行状態 (変数の値など) を得る
- 本来見るべき点を通り過ぎてしまったときに、容易にそこへ戻る
- 通り過ぎた分岐点に戻り、条件を変えて、そこから再実行する

これらはいずれも、デバッグ中に頻繁に行いたくなる操作であるが、従来のデバッガではこうした操作を行うために、まず「手動で」先頭から実行し直すことが必要であった。その際には、それまでのデバッグ過程で設定していたブレークポイントを一旦無効にした上で、実行点が目的の位置に正しく到達するように (そして、誤って通り過ぎてしまわないように) ステップ実行を繰り返さなければならない。

ここで重要なことは、目標点のソースコードに単純にブレークポイントを設定した場合には、「現在の呼び出し」ではなく「過去の呼び出し」で停止してしまうかも知れないため、手動で「正しく」現在位置に復帰するのはそれほど容易ではない、ということである。

従って、これをプログラマ自身が行っている現状では、明らかにデバッグ作業が阻害されていると言える。

「プログラム逆実行」を利用することで、このような余分な作業を自動化できるようになり、プログラマはデバッグに集中できるようになる。

本輪講では、逆実行を実現するには手法として、以下の 3 つの方式を紹介する：

- 状態完全保存による、任意の履歴の表示
- Inverse Statement の利用
- 関数単位の疑似逆実行

第 3節ではまず実用的なものの例として、プログラムが停止したときのみ一部の状態を保存する DDD について紹介し、さらに上記のそれぞれについて第 4節以降で説明する。

3 状態の一部保存：DDD

本節では、逆実行は行わないが、実用的なものの例として、 DDD (The Data Display Debugger) [3] を紹介する (図 1)。

DDD は、 UNIX 上の各種デバッガの GUI (Graphical User Interface) として実装される、フロントエンドである (図 2)。下位のデバッガとして、 GDB 以外にも JDB や Perl debugger なども使えるようになっている。

3.1 DDD の特徴

DDD の特徴はその名の示す通り、プログラム中のデータ構造を表示することにある。線形リストを表示した例を図 3に示す。この例では “Detect aliasing” オプションを設定して、同じセルを指すポインタが 2 つある場合には、セルを共有するようになっている。

“display” ウィンドウに表示されたデータ構造の配置は、ユーザがマウスで自由に変更することが可能になっている。ポインタを示す矢印は自動的に追尾し、その表示位置もカーブの曲率を変える程度には変更できる。

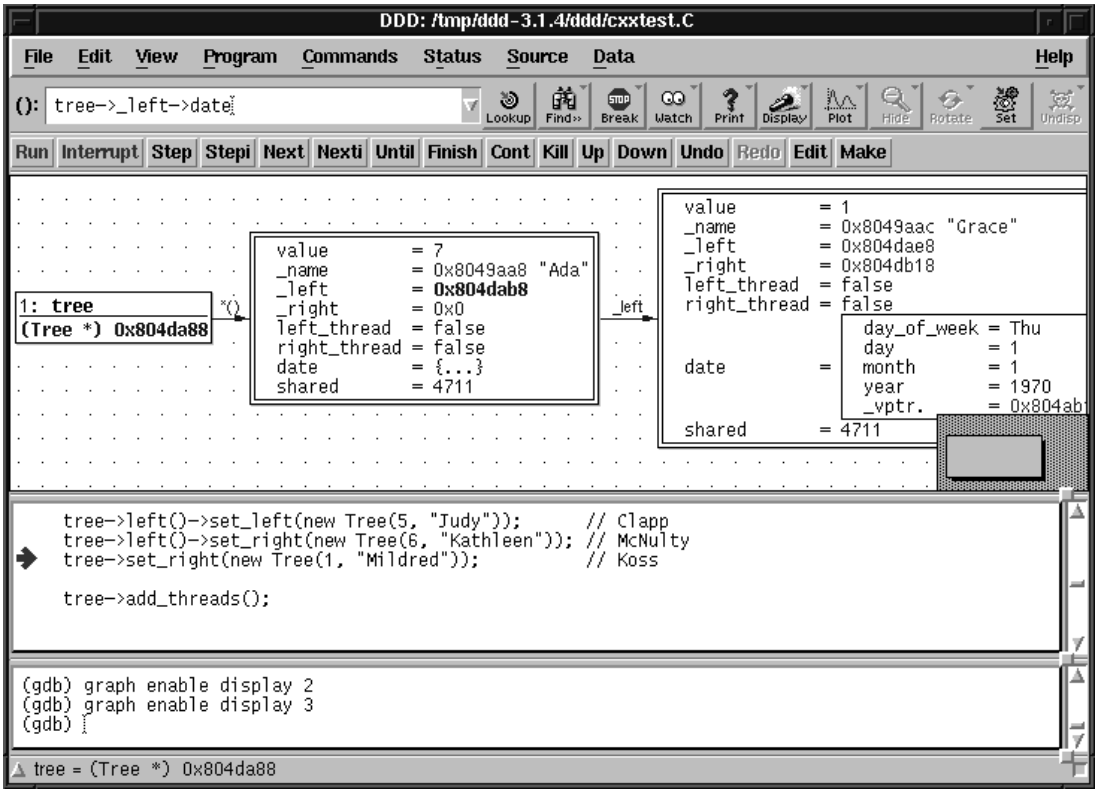

図 1: DDD – The Data Display Debugger –

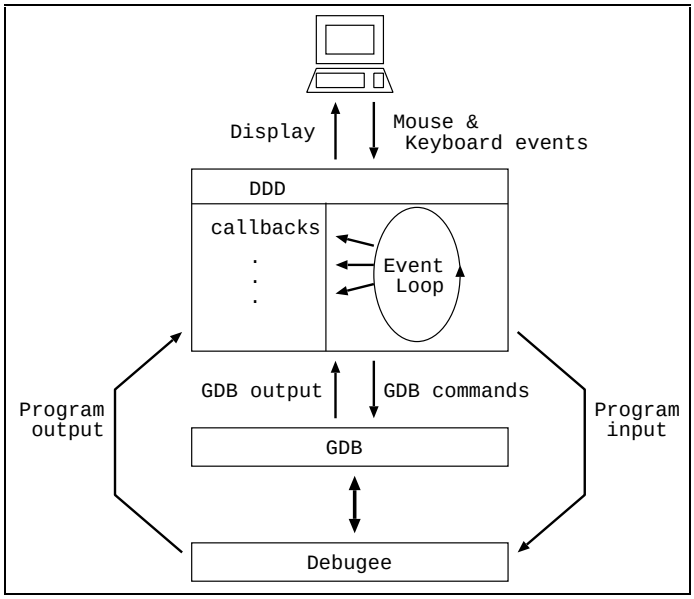

図 2: DDD の概要

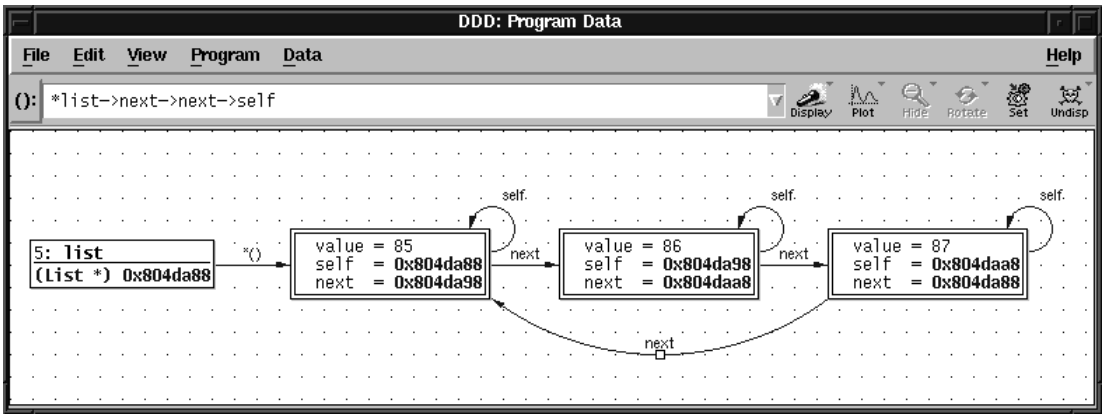

図 3: DDD の “display” ウィンドウ

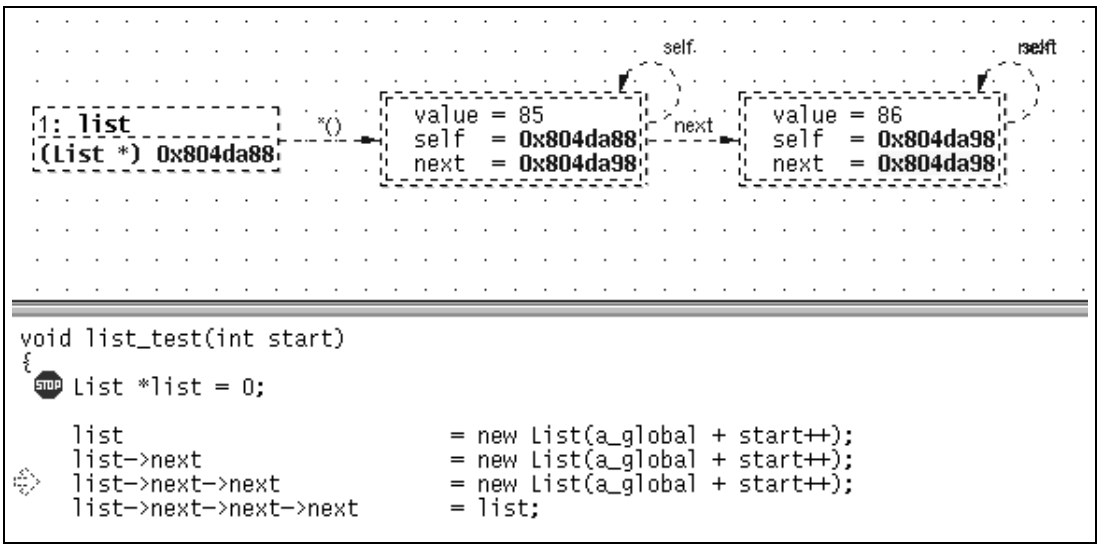

図 4: DDD における undo

一般にこのようなデータの表示方法は、表示するデータ量が少し大きくなるとすぐに破綻することが知られているが、ここではユーザが自分で配置と表示・非表示を選択することによって解決できる (解決しなければならない) ようになっている。

3.2 DDD における実行履歴表示

DDD では実際に実行を逆に戻すことは一切できないが、“undo” という形で DDD に与えたコマンドを undo して、以前の状態を見ることができる。

DDD は制御がデバッギから自分に戻るたびに、ユーザから与えられたコマンドと、その時点での自分の状態 (実行点とそのときの “display” の内容) とを保存する[1]。ユーザから “undo” の指示があると、DDD はバッファに保持していた内容を復元し、“display” ウィンドウの内容を 1 コマンド分、以前の状態に戻す (図 4)。“undo” 中であることを示すため、DDD は実行点を示す矢印と “display” ウィンドウ内の枠を点線で表す。

この例では、“undo” により図 3における $value = 87$ のセル (右端) が表示される前の状態にまで遡っているため、図 4ではこのセルはなく、$value = 86$ のセルの`next` ポインタは自分自身を指して、`self` ポインタと重なって見えている。

[1] バッファのサイズは可変で、負の値を指定すると無制限ということになる。

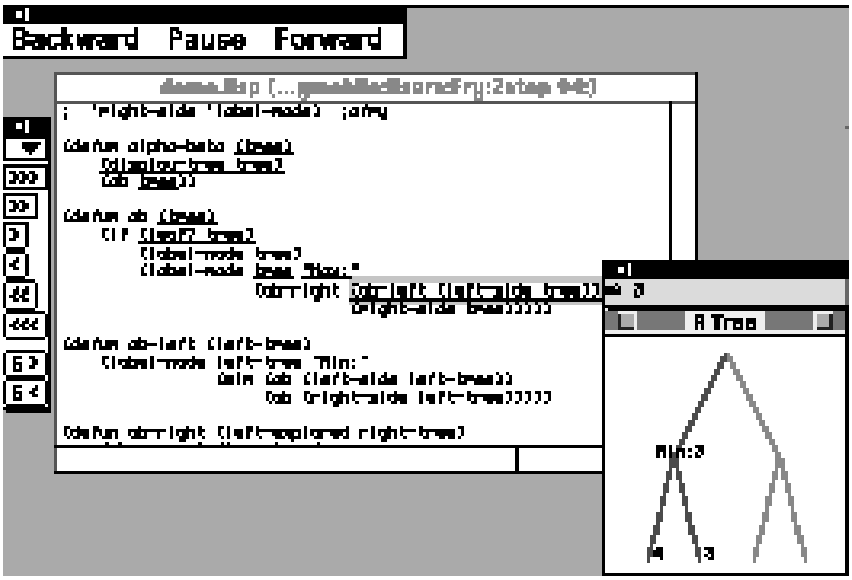

図 5: ZStep 95

この状態は単なる記録の再生に過ぎず、ここで “undo” や “redo” 以外の指示を送ると、DDD は実行を中断していたところまで表示を戻し、そこから通常の実行を再開する。

3.3 DDD の問題点

DDD の “undo” を用いることで、「さっき見た変数の値」を見ることはできる。しかしながら、これには以下のような制限がある：

- その「状況を再現したい時点」を実行する時に、そこでプログラムの実行を一時停止していなければならない (DDD に制御が戻っていなければならない)
- 事前に “display” を指示していなければならない[2]

即ち、“continue” や “next” などを用いて「通り過ぎて」しまった点での情報は取得できないので、DDD の “undo” では知ることが出来ない[3]。

次節で紹介する ZStep 95 は、この点に関しては解決している。

4 状態の完全保存：ZStep 95

プログラマは「静的な」プログラムコードを書きながら、それが実行されるときの「動的な」状況を考えなければならない。ZStep 95 [4] は、その溝を埋めるためのデバッグ環境として、Macintosh Common Lisp 2.0 上に構築されたシステムである (図 5)。

ZStep 95 は以下のような特徴を持つ：

- 専用のインタプリタを用いた、差分生成による完全なプログラム実行履歴と、出力の履歴の保持
- “Video recorder” 形式ボタンによる、実行方向と表示の深さの制御
- プログラム実行を、ソーステキストの形でアニメーション
- 現在の実行点に表示される、変数表示用のウィンドウ
- グラフィカルオブジェクトとそれを表示したプログラム部分を、1 クリックでアクセス

ここでは実行制御に注目するので、最初の 2 点についてのみ説明する。

[2] “undo” 中は Popup Tips による変数値表示は出ない。
[3] そうは言っても、こまめなバージョンアップを繰り返し、公開されて実用に供されている点は評価できる。

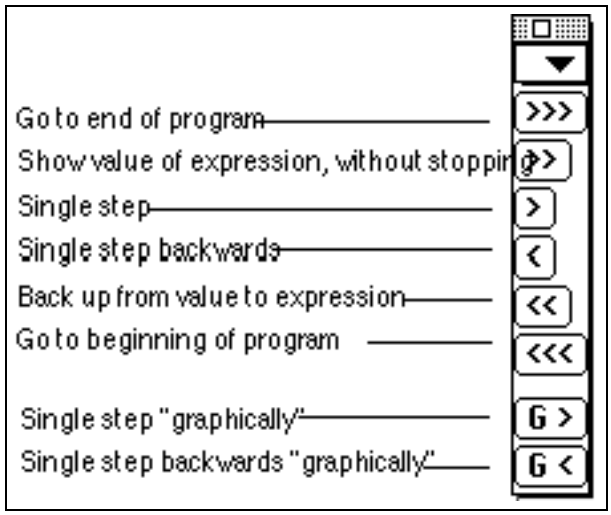

図 6: ZStep 95 の control menu

4.1 履歴の保持

ZStep は特別なインタプリタを用意して、Lisp の式を評価する際に、その順序を list 状に記憶しておくようになっている。

即ち、次に評価すべき式と直前に評価した式を、それぞれ関数の内部に立ち入る場合と立ち入らない場合に分けて保存しておき、制御を移動するコマンドが入力されると、その link をたどり、次に評価すべき式を決定する [6]。

4.2 実行方向の制御

ZStep 95 では、“video recorder” メタファによるメニュー (図 6) によって動作の指示を与える：

> GDB における “step”。関数呼び出しであれば、その内部に入って実行をトレースする
>> GDB における “next”。関数呼び出しであってもその内部はスキップする
< “>” の逆。関数内部に入って逆向きに再生する
<< “<<” の逆。関数内部には入らず、逆向きに再生する

4.3 ZStep 95 の主張

全状態を保存し、専用のインタプリタを用いることは、明らかに著しい速度低下をもたらし、また実装の記述量も大きい。出典の論文 [4] ではそのことに触れており、これに対して以下のように回答している：

- そもそもプログラムはバグで動かないのだから、それ以上遅いということはない。余りにも巨大なプログラムが ZStep に向かないことは確かだが、大抵のバグは根の浅いものであり、そういうものを素早く見つけて生産性を上げられるので、テスト中に実行が遅いくらいは問題ない。
- 幅広いプログラムを実行可能にするには、履歴保存機構を選択的にオン・オフするツールが必要である。簡単な方法は、まずエラーが起こるまで通常実行し、エラーが起こったところでコールスタックを調べ、そこにエントリを持つ関数の履歴だけを取る、という方法である。先頭から再実行しなくてはいけないところが難点ではある。

ZStep を構築した目標は、動的なプログラム視覚化に対する新しいインタフェースを提唱することであり、何もプログラマを鍛えたいということではないのである、と著者らは主張している。

表 1: Inverse Statement

Original Statement	Inverse Statement
`a += b;`	`a -= b;`
`i++;`	`--i;`
`b += c++ + 1;`	`b -= --c - 1;`

4.4 ZStep 95 の問題点

DDD や ZStep では実行履歴を保持するため、それ相応の記憶領域が必要になるため、ディスクへの書き出しなどを行い始めると、かなりの速度低下を生むと思われる[4]。

また、この専用インタプリタは Common Lisp のサブセットしか受け付けないため、現実的なアプリケーションを対象にすることができない。

5 Inverse Statement の利用 : BDb

BDb (Bi-directional Debugger) では "Inverse Statement" という手法が導入されている [8]。これはプログラムのある statement の逆変換となる statement を用いて、保存すべき中間状態の量を削減している点が他と異なっている。

Inverse Statement の例を表 1に示す。

ただし、$a/=b$; のような statement では、 Inverse statement を用いても元の値を完全に正しくは復元できない。そうした種類の statement は、 Inverse statement を用いずに、次に述べる Trace file にその値を格納しておくことになっている。

5.1 BDb の動作

Inverse Statement が利用できるのは、表 1のような、逆操作が定義できる statement だけであり、いつも利用できるわけではない。例えば `a = i;` のような statement は、それまで `a` が保持していた値を完全に失ってしまう。また、条件分岐があった場合には、逆実行時に "then" part と "else" part のいずれを通ってきたかを決定する方法はない。繰返しにおいても同様である。このため、 Inverse Statement だけでは状態を復元することはできない。

そこで、 BDb では Inverse Program からは生成できない、上述の`a` の古い値や制御の合流のような、「履歴」を保持するための Trace File をプログラムの順方向実行時に生成する。また、元のプログラムからは Inverse Statement を利用した Inverse Program を生成しておき、必要に応じて履歴を読み込むための命令を、そこに挿入する。

従って、 BDb ではプログラムの実行に

- Original Program
- Trace File
- Inverse Program

の 3 つが必要になる。

関数呼び出しとそこからのリターンを逆向きにたどるために、関数定義内で用いられた局所変数は、関数からリターンする前に Trace file へ書き出す。

また、 I/O には関知しないことになっている。

5.2 Inverse Program の例 – 条件分岐

図 7に、 Inverse Program の例を示す。

[4] 出典では具体的な数値には触れられていない。

	Original program	Trace file		Inverse program
8	`a = 20;`	`a = 0;`	8	`/* Read history */`
9	`if(a < 10){`		9	
10	`a = 1;`		10	`/* Read history */`
11	`b *= a;`		11	`b /= a;`
12	`}`	`Goto_line 9;`	12	`/* Read history */`
13	`else if(a > 10){`		13	
14	`a = 100;`	`a = 20`	14	`/* Read history */`
15	`c += a;`		15	`c -= a;`
16	`}`	`Goto_line 15;`	16	`/* Read history */`
17	`else{`		17	
18	`a = 10;`		18	`/* Read history */`
19	`d -= a;`		19	`d += a;`
20	`}`	`Goto_line 16;`	20	`/* Read history */`

図 7: Inverse Program の例 – 条件分岐

この例での逆実行過程は以下のようになる：

1. Inverse prog. の 20 行目を読み、 Trace file を読む
2. Inverse prog. の 16 行目に飛ぶ
3. Trace file を読む
4. Inv. prog. の 15 行目に飛ぶ
5. `c -= a;` を実行する
6. Inv. prog. の 14 行目を見て Trace file を読む
7. $a = 20$ を実行する
8. Inv. prog. の 12 行目を見て Trace file を読む
9. Inv. prog. の 9 行目に飛び、 8 行目を見て Trace file を読む
10. $a = 0$ を実行する

5.3 BDb の構成

BDb の構成は図 8のようである。詳細は略すが、インタプリタによって動作しているという点が重要であり、実行速度に関しては甚だ疑問の残る構成となっている。

5.4 BDb の問題点

ZStep と比べると、 Inverse statement を利用しているために、実行履歴は小さくなっていると思われるが、実際にどの程度の大きさになるかや、実行時間については全く触れられていない。

また、 ZStep と同じく、実行方向の指示や変数の値の表示などのコマンドしか許されていない節がある。 Trace file を変更できるように実装を変更すれば、逆実行時に変数の値の変更ができると思われるが、そうなっていないのは残念である。

デバッグを標榜するからには、遡ったところでも変数の値を変更するなど、通常のデバッガが可能で、かつ重要な機能は無視できない。次節では、この問題に対する 1 つの解を提示する。

6 関数単位の疑似逆実行

これまで紹介した手法では、逆実行の単位を「1 つの S 式」や「1 つの statement」としていた。しかしそのためには、膨大となるであろう実行履歴への対策が主となってしまう。また専用インタプリタを用いることになるので、

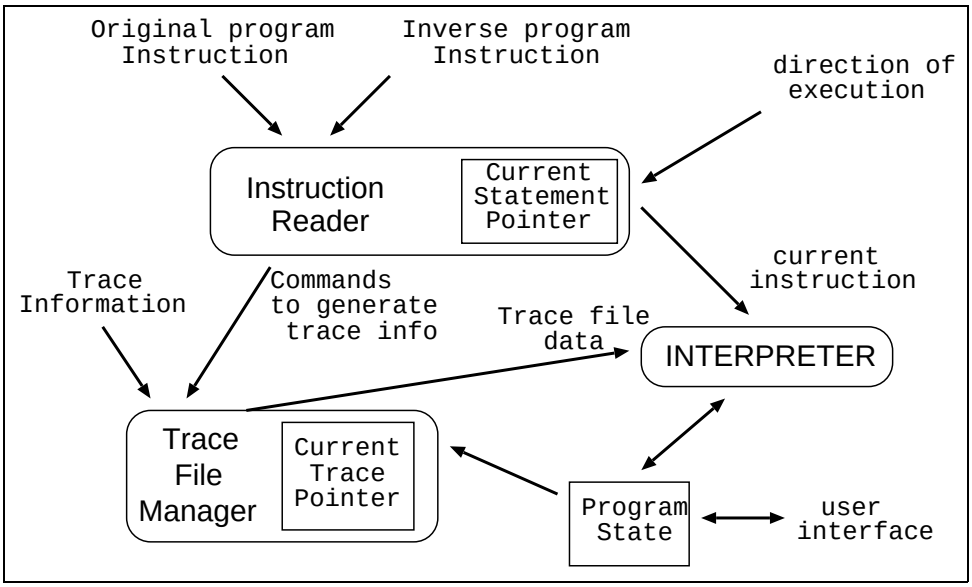

図 8: BDb の構成

- 実行速度が犠牲になる
- プログラミング言語の機能の一部しか使えないために、現実的なアプリケーションを対象にできない
- デバッガとしては、従来型のものに及ばない

など、種々の制約が加わる。

本節では、逆実行の単位をより大きく「関数呼び出し」とすることによって実行履歴の保存を回避し、既存のデバッガを利用することで疑似的な逆実行を可能とする手法を紹介する。

6.1 疑似逆実行

疑似的な逆実行とは、従来のデバッグ環境で人間が手動で行っている手法と同じもので、「先頭から現在点の直前まで再実行」することによって逆実行をしたのと同じ効果をもたらす。これをデバッガが自動的に行うことができれば、プログラマのデバッグ時の負担は小さくなると考えられる。

人間が既存のデバッガを用いて手動で逆実行する時の手順は、

1. 戻りたい場所にブレークポイントを設定する
2. プログラムを先頭から再実行する

である。ブレークポイントを利用することによって、先頭からその場所まではプログラムは通常実行と同じ速度で実行することができるので、再実行によるオーバヘッドがあるとは言え、専用インタプリタよりも現実的な実行時間が得られると考えられる。

図 9に、「現在実行中の関数の入口に戻る」場合を示す。プログラマはその入口に戻ったところで、そこから通常の “next” などを用いて、本来の目標点まで実行を進めれば良いことになる。

6.2 プロファイル用関数 : mcount

プログラムの実行過程を、「関数呼び出し」の単位で扱うために、本来はプロファイル用関数である mcount を利用する。

GCC で -pg オプションを指定すると、対象プログラムにおける実行時の関数実行回数や関数間呼び出し回数といった、プロファイル情報を取得できる。

これは、全ての関数の入口で、最初に mcount というプロファイル情報取得用関数が実行されるためである (図 10)。-pg オプションが指定されると、 GCC は全ての関数定義の先頭に、自動的に mcount の呼び出しを挿入するので、ユーザはオプション指定以外の操作をする必要はない。

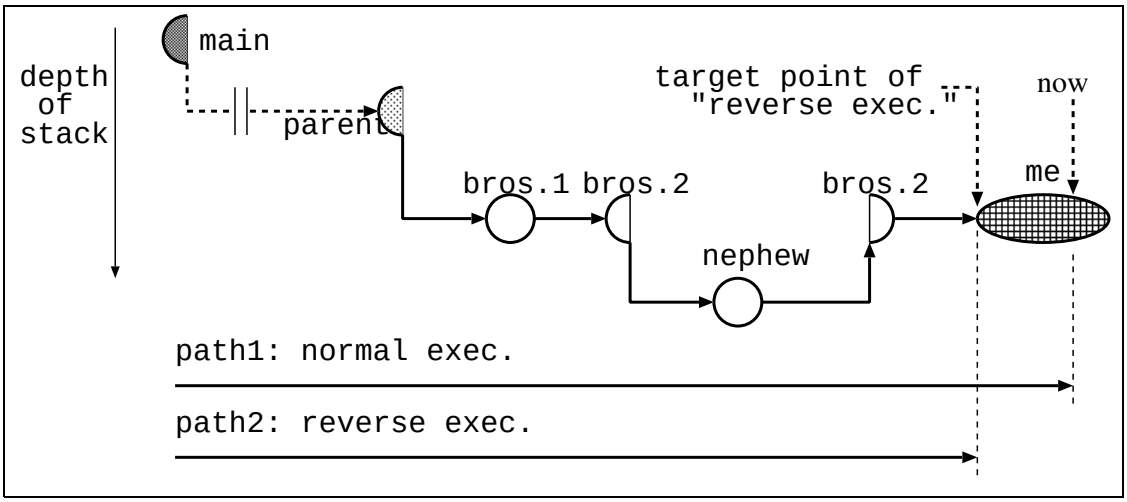

図 9: 疑似逆実行

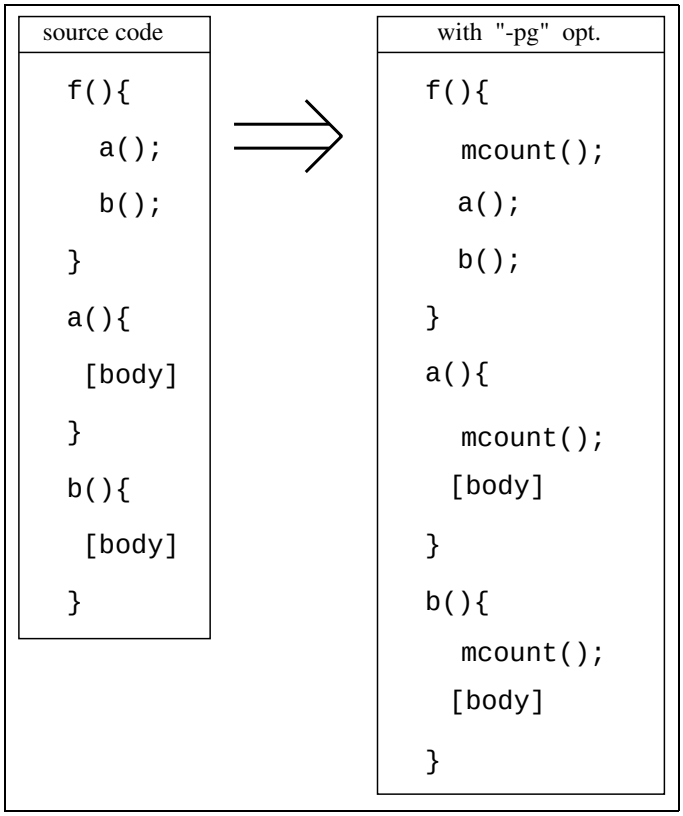

図 10: mcount 呼び出しの挿入

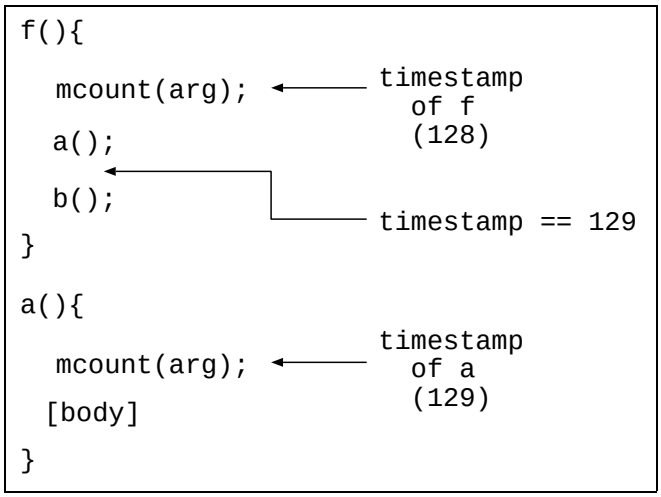

図 11: タイムスタンプの更新

mcount の定義自身は libgmon.a といった名前のライブラリにあり、GCC はコンパイルの最終段階でこのライブラリをリンクして、executable code を生成する。

この mcount を、システム標準の (libgmon.a で定義されている) 実装ではなく、自分で同名の関数を用意することにより、「全ての関数呼び出しの入口」を捕捉できるようになる。

即ち、mcount 内で自身が呼び出されるたびにある変数 (以下、タイムスタンプ) を 1 増やす処理を行えば、全ての関数呼び出しはそのタイムスタンプによって unique に決定できる。そこで、タイムスタンプを利用して条件付ブレークポイントを設定すれば、プログラムの先頭から任意の関数呼び出しの入口まで、制御を移すことができる。

そこで、問題は「戻り場所のタイムスタンプ」をどのように取得するかということに帰結する。

6.3 タイムスタンプの取得

「現在実行中の関数の入口へ戻る」と指示されたときに、単純に現在のタイムスタンプを利用できないことは、図 11より明らかである。

これを解決するためには、タイムスタンプを現在実行中の関数呼び出しに固有の情報として保持できれば良いので、スタックフレームにこれを持つことができれば良いことになる。しかし、スタックフレームを改造することはポータビリティの観点からは問題があるので、ここでは採用しない。

代わりに、スタックフレームを自作の mcount 内でシミュレートし、スタックフレームの各レベルでの最新のタイムスタンプを保持すれば良いことになる。このためには関数の呼び出しとともに、関数から返るところも捕捉できる必要がある。

しかしながら、mcount だけでは「出口」は捕捉できないため、通常実行時にスタックフレームを近似的にしかシミュレートすることができない。そこで、「逆実行」が指示されたときに

scan pass 先頭から再実行して正確なスタックフレームを獲得し、逆実行の目標点 (タイムスタンプ) を取得する

execute pass scan pass で手に入れたタイムスタンプまで、先頭から再度実行し直す

という、2 パス実行を行うことで、疑似逆実行を実現することができる。

6.4 特徴

よって、この手法は以下のような特徴を持つ：

- mcount を変更するだけで済むので、専用のインタプリタを作成するのに比べて、記述量が非常に少ない。
- インタプリタに比べてオーバヘッドがずっと小さい。
- mcount を 1 種類だけ用意すれば、アーキテクチャの違いは GCC が吸収してくれるので、ポータブルな実装が可能である。
- 逆実行に 2 パスを要してしまう。

6.5 自分の研究

上述の 2 パスを要してしまう問題は、「関数の出口で処理をする関数」を用意して、通常実行中にスタックフレームを正確にシミュレートできれば解決する。

しかし、関数の出口にはその返り値が存在するため、

- ソースコードを自動的に変更して、出口に関数呼び出しを挿入するという方法は、返り値を一時的に変数に保持しなければならず、その型を決定するにはソースコードを解析しなくてはならない。
- GCC における mcount 呼び出しの挿入と同様にアセンブリ言語レベルで行うと、ポータビリティが失われる。

ということが分かっており、これをポータブルな形で GCC に実装すべく、研究を進めている。

7 まとめと展望

本輪講では、プログラムの実行履歴再生や逆実行に関するいくつかの手法について紹介した。この分野の研究は 10 年ほど前までは活発に行われていたようだが、ZStep のような、正攻法とも言うべき手法の限界のためか、ここ数年はあまり研究が行われていなかったように見える。

一方で、コンパイラが元よりもっている機構 (プロファイル用関数) を利用して、完全ではないが実用的な補助的手法が登場し、工夫次第でまだ研究の余地が残っていると考えている。

参考文献

[1] Jonathan B. Rosenberg, "How Debuggers Work", John Wiley & Sons, Inc., 1997. (吉川邦夫訳, "デバッガの理論と実装", アスキー出版局, 1998.)

[2] Andreas Zeller and Dorothea Lütkehaus, "DDD – A Free Graphical Front-End for UNIX Debuggers", Informatik-Bericht No.95-07, 1995.

[3] DDD – The Data Display Debugger –, http://www.cs.tu-bs.de/softech/ddd/.

[4] Henry Lieberman and Christopher Fry, "ZStep 95: A Reversible, Animated Source Code Stepper", in *Software Visualization: Programming as a Multimedia Experience*, John Stasko, John Domingue, Blaine Price, Marc Brown, eds., MIT Press, 1997.

[5] David Ungar, Henry Lieberman, and Christopher Fry, "Debugging and the Experience of Immediacy", Communications of the ACM, Vol.40, No.4, pp.38-43, 1997.

[6] Henry Lieberman, "Reversible Object-Oriented Interpreters", First European Conference on Object-Oriented Programming, Springer-Verlog, 1987.

[7] Henry Lieberman, "ZStep 94 Home Page", http://lieber.www.media.mit.edu/people/lieber/Lieberary/ZStep/ZStep.html.

[8] Bitan Biswas and R. Mall, "Reverse Execution of Programs", ACM SIGPLAN Notices Vol.34, No.4, 1999.