命令型プログラミング言語におけるプログラム可逆変換の形式化

M2012MM016 前林達也
指導教員：横山哲郎

1 はじめに

プログラムの可逆実行には，効果的な応用が知られている．例えば，投機的実行では，ロールバックの代わりに可逆実行を用いることでメモリ効率が高まる．特に 1 回の計算が少量である場合，逆方向の実行に要する時間は短く，効果的である．また，デバッグでは，バグの検出箇所から原因を特定する作業を効率化できる場合がある．しかし，現実に用いられているプログラムの多くは可逆性を前提としておらず，可逆実行が不可能である．よって，これらの応用をそのまま導入することはできない．

非可逆なプログラミング言語で書かれたプログラムであっても，プログラム可逆変換によって得られたプログラムを用いれば，可逆実行の導入によって，投機的実行やデバッグを効率的に行える．そのため，既存研究では，非可逆な命令型プログラミング言語に対して可逆性を導入するために，プログラム可逆変換器により可逆実行可能なプログラムを生成する手法が提案されている．一方，投機的実行の効率化などを主目的としたプログラム可逆変換手法では，プログラム可逆変換の正しさが形式的に証明されておらず，常に正しいことを期待できない．その結果，プログラム可逆変換によって得られたプログラムは可逆実行できない場合があり，投機的実行やデバッグにおける振る舞いが正しくない可能性がある．

本研究では，命令型プログラミング言語における，プログラム可逆変換の形式化を行う．われわれの知る限りでは，命令型プログラミング言語における可逆変換に関する研究の中に，プログラム可逆変換を形式化し，その正しさを示した手法は知られていない．形式的な正しさを保証することで，制限された構文のもとでは，現実のプログラミング言語において同様に正しくプログラムを可逆実行できることが期待される．その場合，投機的実行やデバッグといった可逆実行の応用例においても，常に正しい振る舞いをすることが期待される．

本研究の対象とする命令型言語には，一般に利用される C 言語を用いる．しかし，C 言語には公式には意味論が存在しないので，プログラムの性質を形式的に示すことができない．そこでまず，本研究では，Clight[1] とよばれる，形式的な意味論を与えられた C 言語のサブセットを用いる．構文は，Clight から抜粋した基本的なものを対象とし，意味論には構造的操作的意味論を用いる．次に，非可逆なプログラムから，可逆プログラムおよびこれに対応する左逆プログラムの生成器を定義する．最後に，これらの生成器によって得られたプログラムが可逆実行可能であることを形式的に示す．可逆実行可能なプログラムとは，逆方向に実行可能なプログラムが存在し，順方向の実行の後に逆方向の実行をすることで元の状態に戻るような性質をもつプログラムである．このような性質を示す定理を立て，帰納法を用いて証明する．

2 関連研究

論理的な可逆性を得る方法には，Landauer による埋め込み法が知られている [5]．埋め込み法とは，演算のトレースを作成するために，追加の情報を記録する方法である．本研究の目的であるプログラム可逆変換も，データと制御のトレースを記録することで実現可能である．

既存のプログラム可逆変換手法には，Perumalla らによる C 言語プログラムの可逆変換手法 [7] がある．Perumalla らが作成したコンパイラ RCC は，C 言語プログラムを入力すると，可逆変換されたプログラムを生成し，同時に，逆変換されたプログラムを生成する．しかし，Perumalla らの手法では，必ずしも正しく可逆変換されることは期待されない．なぜならば，RCC は投機的実行の効率化への応用を目的として開発されており，可逆変換の正しさを形式的に証明していないからである．本研究では，可逆変換の正しさに着目し，可逆変換によって生成されるプログラムの可逆性を保証する手法を提案する．

プログラムの可逆変換は，投機的実行への応用に有効である．例えば，RCC を用いた投機的実行の効率化が行われている [2]．可逆実行が可能であれば，ロールバックが不要となり，メモリ効率が良い．特に，1 回の計算が少量であれば逆方向の実行に要する時間は短く，より効果的である．しかし，RCC により得られたプログラムを用いた可逆実行が必ずしも正しく振る舞うことは保証されず，投機的実行は状態を復元することができず計算内容を損失する可能性がある．

可逆実行は他にも，デバッグへ応用されている [4]．バグの存在箇所と検出箇所は異なる場合があり，このとき，検出されてから実際のバグを特定するまでに，順方向の実行を繰り返す必要がある．しかし，プログラムが可逆実行可能ならば，逆方向の実行を用いることで実行回数を削減することができ，作業を効率化できると考えられる．

プログラムを可逆実行するには，可逆変換による方法だけでなく，可逆実行可能であることを前提とした可逆プログラミング言語によってプログラムを作成することでも実現できる．可逆プログラミング言語には，例えば， Janus[6] や, R[3] などが存在する．命令型言語である Janus は，言語の可逆性が形式的に証明されている．Janus の構文規則および制約を満たすとき，プログラムは必ず可逆となるよう設計されている．よって，入力に対し出力が定義される Janus プログラムは可逆である．一方，非可逆なプログラムを認めないため，可逆性を前提としないアルゴリズムなどをそのまま導入することはできない場合がある．R も同様に命令型の可逆プログラミング言語であり，Janus と類似した機能を持つ．

一般的なプログラム開発やプログラミング学習の現場では，蓄積された知見を組み合わせて課題を解決するが，それらの知見は可逆性を前提としていない．ゆえに，可

逆プログラミング言語ではそれらの知見を直接的に導入することはできない．そこで，非可逆なプログラムを可逆化するという考えに基づくプログラム可逆変換を用いれば，蓄積された知見をそのまま用いることができる．したがって，可逆実行を幅広く応用するには，プログラム可逆変換のアプローチが有効であると言える．

3 プログラム可逆変換の形式化

本章では，プログラム可逆変換の正しさを示すために，形式化を行う．形式化した命令型プログラミング言語の構文に基づき，その上でプログラム変換を定義して，形式的に正しいことを保証する．

対象とする言語は，広く利用されている C 言語とする．Perumalla らによる先行研究で提案された RCC においても，C 言語で作成されたプログラムに対する可逆変換を行っている．しかし，形式的意味論が公式に存在しない C 言語の性質を示すには，形式的意味論を与える必要がある．そこで，形式的意味論を与えられた C 言語のサブセットとして知られる Clight[1] を対象とし，これに基づいてプログラム可逆変換の形式化を行う．

プログラム可逆変換の形式化のために，関連する概念を形式的に定義することが必要である．プログラム可逆変換が形式的に正しいことを，関連する定理に証明を与えることで保証する．

本研究では，Landauer が示した埋め込み法に基づく，すべてのトレースを記録するアプローチでプログラム可逆変換を行う．よって，計算時間およびメモリ空間の効率を考慮していない．

3.1 プログラム可逆変換とは

可逆性を前提としないプログラミング言語で作成されたプログラムは，通常，可逆ではない．C 言語など多くのプログラミング言語は，可逆性を前提としていない．可逆ではないプログラムを用いて可逆実行をするために，プログラム可逆変換が必要である．プログラムの可逆実行を説明するために，左逆プログラム，可逆化されたプログラムおよび可逆実行可能なプログラムの定義を導入する．

定義 1 (左逆プログラム)．プログラム p が与えられたとき，任意の x, y について，

$$[\![p]\!]x = y \Rightarrow [\![p^{-1}]\!]y = x \tag{1}$$

を満たす p^{-1} を p の左逆プログラムとよぶ．

定義 2 (可逆化されたプログラム)．プログラム p が与えられたとき，あるプログラム p^i が存在して，任意の x, y について，

$$[\![p]\!]x = y \Rightarrow [\![p^r]\!]x = (y, h) \wedge [\![p^i]\!](y, h) = x \tag{2}$$

を満たす p^r を p の可逆化されたプログラムとよぶ．

定義 3 (可逆実行可能なプログラム)．プログラム p が与えられたとき，p の左逆プログラム p^{-1} が存在して，任意の x について，$[\![p]\!]x$ が定義されるならば，

$$[\![p^{-1}]\!]([\![p]\!]x) = x \tag{3}$$

を満たす p を可逆実行可能なプログラムとよぶ．

p	p^r	p^i
x `=` e`;`	`SAVE(`x`);` x `=` e`;`	`RESTORE(`x`);`

図 1 代入文の可逆変換

ここで，順方向の実行が常に先行することを前提とし，左逆のみを扱うことから，

$$[\![p]\!]([\![p^{-1}]\!]x) = x \tag{4}$$

を満たすことを保証しない．ただし，本研究の手法の応用例に挙げられている投機的実行やデバッグには，式 4 を必ずしも満たす必要はない．

C 言語を用いた，プログラム可逆変換の例を図 1 に示す．プログラムは，変換する元のプログラム p と，p の可逆化されたプログラム p^r，p^r の左逆プログラム p^i である．ここで，2 種類のマクロ `SAVE` と `RESTORE` は，トレースを記録するスタックに対する操作を行う命令である．メモリが有効である限り，p^r は可逆実行可能なプログラムである．

3.2 構文

本研究では，トレースの記録によってプログラム可逆変換を行い，正しさを示すことを目的としている．ゆえに，構文は簡略化したものを用い，単純な方法での可逆変換を実現する．本研究で対象とする，Clight から抜粋した抽象構文を以下に示す：

文	s	::=	id `=` a	代入文
		|	s`;`s	文の並び
		|	`if (` a `)` s `else` s	条件文
		|	`while (` a `)` s	ループ
		|	`{` dcl s `}`	複合文
宣言	dcl	::=	(`uint` id)?	
式	a	::=	id	変数識別子
		|	n	整数定数
		|	$a \odot a$	二項演算
二項演算子	$\odot$	::=	`+` | `-` | `^` | `>` | `==`	

整数定数には，標準符号無し整数型の整数を用いる．表記を簡単にするため，`unsigned int` を `uint` と記す．

3.3 意味論

対象とする構文について，構造的操作的意味論を用いて形式的に意味論を定める．構造的操作的意味論では，プログラムにおける式の評価，文の実行のような各ステップでの遷移を自然に記述できる．本研究では，プログラムの順方向の実行および逆方向の実行による遷移を形式的に記述する必要があるため，構造的操作的意味論を用いることで見通しよく表現できる．まず，意味領域を定める：

ブロックの参照	b	$\in$	$\mathbb{Z}$
値	v	$\in$	$\mathbb{Z}_{\mathtt{uint}}$
局所環境	E	::=	$id \mapsto b$
記憶域	M	::=	$b \mapsto v$

式の意味規則：

$$\frac{\langle (E,M), a_1\rangle \to v_1 \quad \langle (E,M), a_2\rangle \to v_2}{\langle (E,M), a_1 \odot a_2\rangle \to [\![\odot]\!](v_1, v_2)}\textsc{Bin}$$

文の意味規則：

$$\frac{\langle (E,M), a\rangle \to v}{\langle (E,M), id\ \texttt{=}\ a\rangle \to M[E(id) \mapsto v]}\textsc{Assign}$$

$$\frac{\langle (E,M), a\rangle \not\to 0 \quad \langle (E,M), s_1\rangle \to M'}{\langle (E,M), \texttt{if (}\ a\ \texttt{)}\ s_1\ \texttt{else}\ s_2\rangle \to M'}\textsc{IfTrue}$$

$$\frac{\langle (E,M), a\rangle \to 0 \quad \langle (E,M), s_2\rangle \to M'}{\langle (E,M), \texttt{if (}\ a\ \texttt{)}\ s_1\ \texttt{else}\ s_2\rangle \to M'}\textsc{IfFalse}$$

図 2 意味規則の例

次に，意味規則を定める：

$$\begin{array}{ll}\langle (E,M), a\rangle \to v & \text{(式の右辺値の評価)}\\ \langle (E,M), s\rangle \to M' & \text{(文の実行)}\end{array}$$

例えば，$\langle (E,M), a\rangle \to v$ という記述は，局所環境 E および記憶域 M のもとで，式 a を評価し，その結果として値 v を得られることを表す．

構文に対して意味を定める意味規則の例を図 2 に示す．ここで，与えられた a, E, M に対して，$\langle (E,M), a\rangle \to v'$ となる v' が存在して $v \neq v'$ であることを $\langle (E,M), a\rangle \not\to v$ と表す．また，算術演算子+, -は，$[\![\odot]\!](v_1, v_2) = v_1 \odot v_2 \bmod 2^{32}$ と定義する．

3.4 左逆文，可逆文，可逆化された文

プログラムの可逆変換を行うためには，プログラムの実行において記憶域を変更し得る文の実行を可逆化する必要がある．本節では，左逆文，可逆文および可逆化された文の定義を導入する．

定義 4 (左逆文)．文 s が与えられたとする．任意の E, M について

$$\langle (E,M), s\rangle \to M' \implies \langle (E,M'), s'\rangle \to M'' \quad (5)$$

ならば $M = M''$ であるとき，s' を s の左逆文とよぶ．

例えば，`x = x + 1` は `x = x - 1` の左逆文である．一方，`x = x * 2` には左逆文が存在しない．例えば，`x` の型が 32 ビットの標準符号無し整数型であるとすると，`x` = 2147483648 の場合，式 `x * 2` の評価結果はオーバーフローする．このとき，定義 4 をみたすような左逆文が存在しない．

可逆文を，次のように定義する．任意の可逆文は，実行する直前および直後の状態が一意に定まる．

定義 5 (可逆文)．任意の E, M, M', M'' について

$$\langle (E,M), s\rangle \to M' \ \wedge\ \langle (E,M), s\rangle \to M'' \implies M' = M'' \quad (6)$$
$$\langle (E,M'), s\rangle \to M \ \wedge\ \langle (E,M''), s\rangle \to M \implies M' = M'' \quad (7)$$

である文 s を可逆文とよぶ．また，式 6 を満たすとき前方決定的であるといい，式 7 を満たすとき後方決定的であるという．

$$\begin{aligned}\mathcal{R}[\![x\ \texttt{=}\ e]\!] &= \texttt{SAVE(}\ x\ \texttt{);}\ x\ \texttt{=}\ e\\ \mathcal{R}[\![\texttt{if (}\ e\ \texttt{)}\ s_1\ \texttt{else}\ s_2]\!] &= \texttt{if (}\ e\ \texttt{) \{}\\ &\qquad \mathcal{R}[\![s_1]\!]\texttt{;}\\ &\qquad \texttt{SAVE(1)}\\ &\quad \texttt{\} else \{}\\ &\qquad \mathcal{R}[\![s_2]\!]\texttt{;}\\ &\qquad \texttt{SAVE(0)}\\ &\quad \texttt{\}}\end{aligned}$$

$$\begin{aligned}\mathcal{I}[\![x\ \texttt{=}\ e]\!] &= \texttt{RESTORE(}\ x\ \texttt{)}\\ \mathcal{I}[\![\texttt{if (}\ e\ \texttt{)}\ s_1\ \texttt{else}\ s_2]\!] &= \texttt{\{}\\ &\qquad \texttt{uint \_b;}\\ &\qquad \texttt{RESTORE(\_b);}\\ &\qquad \texttt{if (\_b) \{}\\ &\qquad\quad \mathcal{I}[\![s_1]\!]\\ &\qquad \texttt{\} else \{}\\ &\qquad\quad \mathcal{I}[\![s_2]\!]\\ &\qquad \texttt{\}}\\ &\quad \texttt{\}}\\ &\quad \textbf{where } \texttt{\_b} \notin \mathrm{Var}(s_1, s_2)\end{aligned}$$

図 3 生成器の例

例えば，`x = x + 1` は可逆文である．一方，Clight の文は必ずしも可逆ではない．例えば，文 `x = 1` の実行直前の $M(\texttt{x})$ の値は一意に定まらない．したがって，この場合は，式 7 は満たされない．

可逆化された文を，次のように定義する．定義に示すとおり，可逆化された文は，元の文の意味を保持している．

定義 6 (可逆化された文)．ある文 s とある可逆文 s' が与えられたとする．任意の E, M, M' および任意の b に対して

$$M'(b) \neq \mathit{undef} \implies M'(b) = M''(b) \quad (8)$$

であるようなある M'' について

$$\langle (E,M), s\rangle \to M' \iff \langle (E,M), s'\rangle \to M'' \quad (9)$$

が成り立つならば，s' を s の可逆化された文とよぶ．

3.5 可逆化文生成器と左逆可逆化文生成器

本節では，可逆実行に必要な文を生成する 2 つの変換器 $\mathcal{R}$ と $\mathcal{I}$ を定義する．$\mathcal{R}[\![s]\!]$ では s の可逆化された文 s^r を，$\mathcal{I}[\![s]\!]$ では s^r の左逆文をそれぞれ生成する，すなわち，$\mathcal{R}$ を可逆化文生成器，$\mathcal{I}$ を左逆可逆化文生成器になるようにする．生成器の例を，図 3 に示す．

ここで，マクロ SAVE，RESTORE を，意味規則

$$\frac{\langle (E,M), a\rangle \to v}{\langle (E,M), \texttt{SAVE(}\ a\ \texttt{)}\rangle \to M[\mathrm{stk} \mapsto v :: M(\mathrm{stk})]}\textsc{Save}$$

$$\frac{v :: \mathrm{stk}' = M(\mathrm{stk})}{\langle (E,M), \texttt{RESTORE(}\ x\ \texttt{)}\rangle \to M[E(x) \mapsto v][\mathrm{stk} \mapsto \mathrm{stk}']}\textsc{Restore}$$

を満たすものとして定義する．stk とは，記憶域 M における特別なブロックの参照であり，マクロ SAVE，RESTORE によってのみ変更される．stk により参照される値を基準に，記録する値を連結することで，スタックの役割をする．

生成器を用いて得られた文が可逆実行可能であることを示すために，満たすべき性質の証明を行う．まず，任意の文について，可逆化文生成器によって得られた文が可逆化された文であることを示す．

定理 7. 任意の文 s について，$\mathcal{R}[\![s]\!]$ は s の可逆化された文である．

次に，任意の式の評価が前方決定的であることを示す．前方決定的な式は，文に関する定義 5 と同様に，次のように定義する．

定義 8 (前方決定的な式)**.** 任意の E, M, v, v' について

$$\langle (E, M), a\rangle \to v \ \wedge \ \langle (E, M), a\rangle \to v' \implies v = v' \quad (10)$$

を満たす式 a は前方決定的な式である．

前方決定的な式は，任意の E, M に対して式の評価の値を一意に定める．すなわち，式 a の評価 $\langle (E, M), a\rangle \to v$ という関係によって，値 v は E, M, a の関数であることが定められる．

補題 9 (式の評価の前方決定性)**.** 任意の式の評価は前方決定的である．

なお，式は必ずしも後方決定的ではない．例えば，$a_1 = 2, a_2 = 3, \odot = +$ もしくは $a_1 = 1, a_2 = 4, \odot = +$ とすると，$\langle (E, M), a_1 \odot a_2\rangle \to 5$ となるので，式の評価により定まる関数は単射ではない．ただし，式の評価は局所環境および記憶域を変更しないため，文の可逆性を保証するためには式の評価が後方決定的である必要はない．

次に，任意の文について可逆化文生成器で得られた文が可逆文であることを示す．これを示すために，まず可逆化に用いられるマクロ `SAVE` の可逆性を示し，これを補題として用いて証明を行う．

補題 10 (`SAVE` の可逆性)**.** 任意の式 a に対して，`SAVE(` a `)` は，可逆文である．

定理 11. 任意の文 s について，$\mathcal{R}[\![s]\!]$ は可逆文である．ただし，s に関する導出木は有限であるとする．

文 s が “While” 文の場合，導出木の前提に結論と同じ大きさの導出木が出現する．このとき，構造帰納法による証明ができず，導出木が有限であることを前提とした数学的帰納法によって証明が与えられる．この前提は，導出に関する帰納法を用いた証明により省略できる．

最後に，任意の文について，左逆可逆化文生成器で得られた文が，可逆化文生成器で得られた文の左逆文となっていることを示す．これにより，対象とする言語において，任意の文は可逆実行可能であることが示される．

定理 12. 任意の文 s について，$\mathcal{I}[\![s]\!]$ は $\mathcal{R}[\![s]\!]$ の左逆文である．

なお，一般に，$\mathcal{I}[\![s]\!]$ は $\mathcal{R}[\![s]\!]$ の逆文ではない．実際，s が `x = 1` のとき，まず `SAVE` によって実行直前の `x` のデータを記録すれば，`RESTORE(x)` が左逆文となることが予想できるが，記憶域に記録されたデータがない場合，`RESTORE` を用いることができないので，右逆文ではない．

4 おわりに

本研究では，命令型プログラミング言語におけるプログラム可逆変換の形式化を行った．これにより，本研究の対象となる形式化された言語において，プログラム可逆変換が形式的に正しいことを保証した．

C 言語が対象であるが，形式的意味論が公式に存在しないことから，形式化された C 言語のサブセット Clight を用いた．構文と意味論を定め，対象となる文に対する可逆化文生成器および左逆可逆化文生成器を定義し，これにより得られる任意のプログラムが可逆実行可能であることを形式的に証明した．

既存研究では，C 言語プログラムに対するプログラム可逆変換器が提案されている．しかし，その手法には形式的証明が与えられておらず，結果として誤ったプログラム可逆変換が存在することを確認した．これに対して本研究では，形式的証明のもと，プログラム可逆変換が常に正しいことを保証した．さらに，既存研究の問題を解決する可能性がある．

構文は制限されており，実際の C 言語プログラムを可逆変換することは難しい．ただし，本研究で示した形式化の手順および証明すべき定理に従えば，機械的に構文の拡張を行うことができるといえる．また，記録すべきデータと制御を削減する効率化も可能であると考えており，この場合も同様に，生成器の定義を変更して定理を証明すればよい．

今後の課題は，まず，構文の拡張が挙げられる．本研究では構文を制限しており，Clight において手法の健全性を予想することはできるが，完全性を主張できない．次に，適用実験によってプログラム可逆実行の実行効率を確認し，最後に，手法の効率化を行う必要がある．

参考文献

[1] Blazy, S. and Leroy, X.: Mechanized semantics for the Clight subset of the C language, *Journal of Automated Reasoning*, Vol. 43, No. 3, pp. 263–288 (2009).

[2] Carothers, C. D., Perumalla, K. S. and Fujimoto, R. M.: Efficient Optimistic Parallel Simulations Using Reverse Computation, *ACM Transactions on Modeling and Computer Simulation*, Vol. 9, No. 3, pp. 126–135 (1999).

[3] Frank, M. P.: Reversibility for Efficient Computing, PhD Thesis, Massachusetts Institute of Technology (1999).

[4] Koju, T., Takada, S. and Doi, N.: An Efficient and Generic Reversible Debugger using the Virtual Machine based Approach, *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (*VEE '05*), pp. 79–88 (2005).

[5] Landauer, R.: Irreversibility and Heat Generation in the Computing Process, *IBM Journal of Research and Development*, Vol. 5, No. 3, pp. 183–191 (1961).

[6] Lutz, C.: Janus: A time-reversible language, *Letter to R. Landauer*. `http://www.cise.ufl.edu/~mpf/rc/janus.html` (1986).

[7] Perumalla, K. S. and Fujimoto, R. M.: Source-code Transformations for Efficient Reversibility, Technical Report GIT-CC-99-21, Georgia Institute of Technology (1999).