引数渡し機構を拡張した 可逆プログラミング言語の可逆性の保証

2010SE199 新海由侑　2010SE237 田中秀明
指導教員：横山哲郎

1　はじめに

現在，提案されている可逆命令型プログラミング言語では，高水準プログラミング言語に通常は備わっている表現力を高めるための機構が備わっていない．可逆命令型プログラミング言語のひとつである Janus[1][2][3] には，引数渡し機構が提案されているが，不十分であると考えられる点がある．たとえば，提案されている Janus には，変数名と記憶場所との対応関係を表す環境という概念が導入されていない．これにより，可逆性を保ったまま引数を渡すとき，実引数が変数に限定されるため，一般に用意すべき変数が増えてしまう．ここで，可逆性とは，任意の状態において，その直前と直後の状態がそれぞれ一意に定まることをいう．また，可逆プログラミング言語とは，そのプログラムが必ず可逆性をもつような言語設計がなされているプログラミング言語である．

我々は本研究において，環境の概念を追加した Janus の引数渡し機構を提案し，その可逆性を証明する．これにより，変数に加えて，実引数として式を渡すことができるようになり，一般に変数の数を減らすことができる．また，環境の導入に伴い，配列を連続的に扱うことができるようになる．

引数渡し機構を拡張した Janus で，実引数として式を渡すプログラムを実現し，現在提案されている Janus によるプログラムとの比較を行う．現在提案されている Janus では，可逆性を保ったまま式を渡すことができない．これに対して，我々の提案する Janus では，引数として式を渡すことによって変数の数を減らし，表現力の高いプログラム記述に繋げることができる．

2　関連研究

2.1　Janus

前述のような不十分な点は存在するが，Janus は我々の研究におけるアイデアを試す可逆プログラミング言語として適した特性を備えている．以下に Janus が備える主要な特性を 4 つ紹介する．第 1 に，Janus は，可逆であることの形式的証明がなされており，非可逆なプログラムを記述することはできない．これにより，答えを返した任意のプログラムの可逆性が必ず保証される．第 2 に，Janus では同一のプロシージャを順，逆の両方向に実行できる．第 3 に，Janus では可逆性を保つために構文に対していくつかの条件や制約が設けられているが，これらによってプログラマの意図しないゴミ情報が発生することのないように設計がされている．なお，ゴミ情報とは，プログラムに期待される本来の出力以外に発生する出力のことである．第 4 に，Janus は汎用的な可逆プログラミング言語である．

2.2　Jensen's Device

Jensen's Device とは，命令型プログラミング言語 Algol60 において提案された総和を求める計算を行うプログラムである．文献 [4] においても，Jensen's Device を参照し，文献中で用いられている．Jensen's Device は，名前渡しのもつ力を最大限生かすような構造をしており，名前渡しを実装していないプログラミング言語では記述することができない．

3　環境記憶域モデルを適用した可逆プログラミング言語

我々は，Janus に環境記憶域モデルを適用した．環境 $\gamma :: \mathrm{Vars} \cup \{next\} \cup \mathrm{Cons} \rightharpoonup \mathrm{Lvals} \cup \mathrm{Vals}$ は変数 x, $next$ または定数 c と左辺値 l または値 v とを対応付ける部分関数である：

$$x(\gamma[x' \mapsto l']) = \begin{cases} l' & \textbf{if } x' = x \\ x\gamma & \textbf{otherwise} \end{cases}$$

γ の一番右側が必ず $[next \mapsto \cdot]$ になるようにすることで，可逆な意味規則にしている．なお，$next$ は現在使用されている記憶域の次に使用されていない記憶域を指す特別な変数名である．記憶域 $\sigma :: \mathrm{Lvals} \rightharpoonup \mathrm{Vals}$ は左辺値 l と値 v を対応付ける部分関数であり，後置演算子 $[l, v_1, v_2] :: \mathrm{Stores} \rightharpoonup \mathrm{Stores}$ は，記憶域の記憶場所 l にある値 v_1 を v_2 に更新する：

$$(\sigma[l' \mapsto v'])[l, v_1, v_2] = \begin{cases} \sigma[l' \mapsto v_2] & \textbf{if } l' = l \text{ and } v' = v_1 \\ (\sigma[l, v_1, v_2])[l' \mapsto v'] & \textbf{otherwise} \end{cases}$$

この演算子には必ず逆関数が存在する：

$$[l, v_1, v_2]^{-1} = [l, v_2, v_1]$$

ここで，環境記憶域モデルを適用した Janus の構文の一部を紹介する．本稿で取り扱う構文領域を図 1, 意味領域を図 2, 構文規則を図 3, 操作的意味論を図 4 に示す．

まず，操作的意味論における式と文の判断についての説明を記す．ただし，e は式，Γ はプロシージャ名とプロシージャ本体を対応させる関数である．式の評価は，

$$\gamma, \sigma \vdash_{expr} e \Rightarrow v$$

で表され，環境 γ, 記憶域 σ において，式 e の評価値が v になることを意味する．文の実行は，

$$\gamma, \sigma \vdash^{\Gamma}_{stmt} s \Rightarrow \gamma, \sigma'$$

$s \in \mathrm{Stms}$ $x \in \mathrm{Vars}$ $t \in \mathrm{Type}$ $\odot \in \mathrm{ModOps}$
$e \in \mathrm{Exps}$ $d \in \mathrm{Vdecs}$ $c \in \mathrm{Cons}$ $\otimes \in \mathrm{Ops}$
$p \in \mathrm{Procs}$ $q \in \mathrm{PIds}$ $ct \in \mathrm{CallTypes}$

図 1 構文領域

$v \in \mathrm{Vals} = \mathbb{Z}_{32}$
$l \in \mathrm{Lvals} = \mathbb{N}$
$x \in \mathrm{Vars} = \{$ a, b, ... $\}$
$\gamma \in \mathrm{Envs} = \mathrm{Vars} \cup \{next\} \cup \mathrm{Cons} \rightharpoonup \mathrm{Lvals} \cup \mathrm{Vals}$
$\sigma \in \mathrm{Stores} = \mathrm{Lvals} \rightharpoonup \mathrm{Vals}$
$\Gamma \in \mathrm{Pmaps} = \mathrm{PIds} \rightharpoonup \mathrm{Procs}$
$\xi \in \mathrm{Substs} = \mathrm{Stms} \to \mathrm{Stms}$

図 2 意味領域

$d ::= x \mid x\texttt{[}c\texttt{]}$
$t ::= \texttt{int}$
$s ::= x \odot \texttt{=}\ e \mid x\texttt{[}e\texttt{]} \odot \texttt{=}\ e \mid$
$\texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s\ \texttt{fi}\ e \mid$
$\texttt{local}\ t\ x\ \texttt{=}\ e\ s\ \texttt{delocal}\ t\ x\ \texttt{=}\ e \mid$
$\texttt{call}\ q\texttt{(}e\texttt{)} \mid \texttt{uncall}\ q\texttt{(}e\texttt{)} \mid \cdots$
$e ::= c \mid x \mid x\texttt{[}e\texttt{]} \mid e \otimes e$
$c ::= \texttt{-2147483648} \mid \cdots \mid \texttt{2147483647}$
$\odot ::= \texttt{+} \mid \texttt{-} \mid \texttt{\^{}}$
$\otimes ::= \odot \mid \texttt{*} \mid \texttt{/} \mid \cdots$
$ct ::= \texttt{val} \mid \texttt{name} \mid \texttt{ref}$
$p ::= \texttt{procedure}\ q\texttt{(}ct\ d\texttt{)}\ s$

図 3 構文規則

$$\frac{x\gamma = l \quad l\sigma = v_1 \quad \gamma, \sigma[l, v_1, undef] \vdash_{expr} e \Rightarrow v \quad v_2 = [\![\odot]\!](v_1, v)}{\gamma, \sigma \vdash_{stmt} x \odot \texttt{=} e \Rightarrow \gamma, \sigma[l, v_1, v_2]} \ \textsc{AssVar}$$

$$\frac{\gamma, \sigma \vdash_{expr} e_1 \not\Rightarrow 0 \quad \gamma, \sigma \vdash^{\Gamma}_{stmt} s_1 \Rightarrow \gamma, \sigma' \quad \gamma, \sigma' \vdash_{expr} e_2 \not\Rightarrow 0}{\gamma, \sigma \vdash^{\Gamma}_{stmt} \texttt{if}\ e_1\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ e_2 \Rightarrow \gamma, \sigma'} \ \textsc{IfTrue}$$

$$\frac{\gamma, \sigma \vdash_{expr} e_1 \Rightarrow 0 \quad \gamma, \sigma \vdash^{\Gamma}_{stmt} s_2 \Rightarrow \gamma, \sigma' \quad \gamma, \sigma' \vdash_{expr} e_2 \Rightarrow 0}{\gamma, \sigma \vdash^{\Gamma}_{stmt} \texttt{if}\ e_1\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ e_2 \Rightarrow \gamma, \sigma'} \ \textsc{IfFalse}$$

$$\frac{\begin{array}{c}\gamma, \sigma \vdash_{expr} e \Rightarrow v \\ \gamma[x \mapsto l][next \mapsto new(l)], \sigma[l, undef, v] \vdash^{\Gamma}_{stmt} s \Rightarrow \gamma[x \mapsto l][next \mapsto new(l)], \sigma' \\ \gamma, \sigma' \vdash_{expr} e' \Rightarrow v'\end{array}}{\gamma[next \mapsto l], \sigma \vdash^{\Gamma}_{stmt} \texttt{local}\ t\ x\ \texttt{=}\ e\ s\ \texttt{delocal}\ t\ x\ \texttt{=}\ e' \Rightarrow \gamma[next \mapsto l], \sigma'[l, v', undef]} \ \textsc{LocMem}$$

$$\frac{\begin{array}{c}\Gamma(q) = \texttt{procedure}\ q\texttt{(val}\ y\texttt{)}\ s \quad \gamma, \sigma \vdash_{expr} e \Rightarrow v \quad \gamma, \sigma' \vdash_{expr} e \Rightarrow v \\ \gamma[y \mapsto l][next \mapsto new(l)], \sigma[l, undef, v] \vdash^{\Gamma}_{stmt} s \Rightarrow \gamma[y \mapsto l][next \mapsto new(l)], \sigma'\end{array}}{\gamma[next \mapsto l], \sigma \vdash^{\Gamma}_{stmt} \texttt{call}\ q\texttt{(}e\texttt{)} \Rightarrow \gamma[next \mapsto l], \sigma'[l, v, undef]} \ \textsc{CallVal}$$

$$\frac{\Gamma(q) = \texttt{procedure}\ q\texttt{(ref}\ y\texttt{)}\ s \quad x\gamma = l \quad \gamma[y \mapsto l], \sigma \vdash^{\Gamma}_{stmt} s \Rightarrow \gamma[y \mapsto l], \sigma'}{\gamma, \sigma \vdash^{\Gamma}_{stmt} \texttt{call}\ q\texttt{(}x\texttt{)} \Rightarrow \gamma, \sigma'} \ \textsc{CallRef}$$

$$\frac{\Gamma(q) = \texttt{procedure}\ q\texttt{(name}\ y\texttt{)}\ s \quad \gamma, \sigma \vdash^{\Gamma}_{stmt} s[e/y] \Rightarrow \gamma, \sigma'}{\gamma, \sigma \vdash^{\Gamma}_{stmt} \texttt{call}\ q\texttt{(}e\texttt{)} \Rightarrow \gamma, \sigma'} \ \textsc{CallName} \qquad \frac{\gamma, \sigma' \vdash^{\Gamma}_{stmt} \texttt{call}\ q\texttt{(}e\texttt{)} \Rightarrow \gamma, \sigma}{\gamma, \sigma \vdash^{\Gamma}_{stmt} \texttt{uncall}\ q\texttt{(}e\texttt{)} \Rightarrow \gamma, \sigma'} \ \textsc{Uncall}$$

図 4 操作的意味論

で表され，環境 γ, 記憶域 σ において，文 s を実行すると，記憶域 σ' に更新されることを意味する．なお，判断は文献 [5] のように記述することもできる．ある文 s が

$$\gamma, \sigma \vdash_{stmt} s \Rightarrow \gamma, \sigma' \wedge \gamma, \sigma \vdash_{stmt} s \Rightarrow \gamma, \sigma'' \Longrightarrow \sigma' = \sigma''$$

を満たすとき，その文 s は前方決定性があるといい，

$$\gamma, \sigma' \vdash_{stmt} s \Rightarrow \gamma, \sigma \wedge \gamma, \sigma'' \vdash_{stmt} s \Rightarrow \gamma, \sigma \Longrightarrow \sigma' = \sigma''$$

を満たすとき，その文 s は後方決定性があるという．また，ある文 s に前方決定性と後方決定性があるとき，その文 s は可逆性をもつという．

まず，規則 AssVar で表現される変数代入について記す．一般的な命令型プログラミング言語で用いられている代入をそのまま可逆プログラミング言語に適用することはできない．なぜなら，代入は情報を上書きする操作であり，代入後の値を知ることができず，非可逆となってしまうためである．そこで，Janus では代入操作を加算代入 `+=`, 減算代入 `-=`, 排他的論理和代入 `^=` に限定している．また，代入の左辺の式に出現する変数 x の左辺値と同じ左辺値に対応付けられている変数は，右辺の式に出現してはならないという制約を加えている．これは，出力値から入力値を計算できない記述を避けるために設けられている．たとえば，`x -= x` のような代入文を記述することはできない．この代入文に相当する C 言語の代入文 `x -= x;` では，変数 x の値に依らず，代入文実行後の変数 x の値は 0 となる．計算結果の 0 という値から，変数 x の元々の値は知りえない．したがって，この代入文は後方決定性を持たず，可逆であるとはいえない．このような記述をした場合，実行時エラーとなる．

次に，規則 IfTrue と規則 IfFalse で表現される条件分岐について記す．一般的な命令型プログラミング言語の条件分岐文との違いは，アサーションが加えられている点である．アサーションとは必ず満たすべき条件を意味し，条件分岐文においては，条件分岐文を抜けるときに満たすべき条件として用いられている．これによって，逆実行の際にも真と偽のどちらに分岐するのかが一意に定まるため，可逆性をもつことになる．構文としては，`if` に続く式 e_1 の評価値が真であれば `then` に続く文 s_1 を，偽であれば `else` に続く文 s_2 を実行する．なお，Janus では真を非 0, 偽を 0 としている．その後，アサーションである `fi` に続く式 e_2 が評価されることになる．式 e_1 を真で通過していれば式 e_2 も真となるとき，式 e_1 を偽で通過していれば式 e_2 も偽となるときのみ条件分岐文を抜け，これを満たさない場合は実行時エラーとなる．なお，逆実行の際は `fi` で条件を与えて分岐を行い，`if` をアサーションとして機能させている．

最後に，規則 LocMem で表現される局所変数ブロックについて記す．一般的な命令型プログラミング言語と違い，Janus では割り当てられた局所変数について条件付きで解放を行う．具体的には，`local` x=e と記述すると，局所変数 x が定義され，式 e の評価値が初期値として格納される．その後，`delocal` x=e' という記述によって，局所変数 x が解放される．このとき，局所変数 x は式 e' の評価値と一致していなければならず，不一致の場合は実行時エラーとなる．これが，局所変数ブロックに可逆性をもたせるための条件である．もし，定義から解放の間で局所変数の値が変更されていた場合，逆実行の際の初期値が断定できない．そこで，このような条件を加えている．なお，逆実行の際の初期値は `delocal` によって与えられ，`local` によって条件付きで解放が行われる．

4 可逆プログラミング言語における引数渡し

既存の言語では実引数と仮引数を結合するために，様々な引数渡しの方法が用いられている．可逆プログラミング言語においては，引数渡し機構によって非可逆とならないように，制約を加える必要もある．本研究では，一般的に用いられている値渡し，参照渡し，名前渡しについて Janus で実現した．

値渡しは実引数の評価値を仮引数に渡す引数渡しの方法であり，規則 CallVal によって表現されている．値渡しの場合，仮引数と実引数で割り当てられる記憶場所が異なり，プロシージャを抜けたときに仮引数に割り当てられた記憶場所が解放される．このため，仮引数の値が何であったかが分からなくなり，一般に非可逆となる．そこで，プロシージャを抜ける際の仮引数 y が実引数 e の評価値 v に対応付けられていない場合，構文エラーとなるようにした．このような記述をする必要はあるが，この制限によって可逆な値渡しを実現することができた．

参照渡しは実引数の左辺値を仮引数に渡す引数渡しの方法であり，規則 CallRef で表現されている．参照渡しの場合，仮引数が実引数の値を間接参照するため，仮引数の値の変更が実引数の値にも反映されることになる．

名前渡しは実引数の式を仮引数に渡す引数渡しの方法であり，規則 CallName で表現されている．名前渡しでは，呼び出されたプロシージャ内の仮引数は実引数のテキストがそのまま置換される．また，プロシージャ呼出し文には，呼び出されたプロシージャの仮引数が実引数に置換されたテキストがそのまま置換される．このため仮引数の値の変更が，そのまま実引数の値の変更となる．

ただし，参照渡しや名前渡しによって引数を渡す場合，複数の仮引数に同じ実引数が渡されることを制限する必要がある．これは，この行為によってエイリアシングが発生し，エイリアシングによって非可逆となる可能性があるためである．非可逆となる具体的な例を，図 5 に示す．なお，ここでは参照渡しで引数が渡されているものとする．プロシージャ`sub` は，引数として 2 つの整数 `a` と `b` を受け取り，`a` を左辺値として，`b` を減算代入するプロシージャである．ここで問題となるのが，`sub` の呼び出しにおいて，仮引数 `a` と `b` の両方に，実引数として同じ `x` の左辺値を渡している点である．これは，プロシージャ`sub` の仮引数 `a` と `b` が，同じ `x` の記憶場所を参照していることを意味している．つまり，プロシージャ`sub` 内の演算 `a -= b` における左辺値 `a` も，右辺値 `b` も，`x` として計算が行われている．これは 3 章で示した代入における問題と同様に，非可逆となることになる．

なお，Janus では `uncall` によってプロシージャを逆呼出しすることができる．これについては，どの呼出し方

```
procedure sub(ref a, ref b)
  a -= b

call sub(x, x)
```

図 5 エイリアシングの問題が発生する例

法に対しても規則 UNCALL で実現することができている．また，これらの引数渡し機構を応用することによって，3 種類の引数渡し方法の組み合わせ自由な複数引数渡しも実現可能である．

5 可逆性の証明

我々は，導出木に対する構造帰納法を用いて，環境記憶域モデルを適用した Janus のすべての文が前方後方決定性をもつことを証明し，以下の定理 1 を得た．これによって，Janus の可逆性を保証することができた．

定理 1 Janus のすべての文が可逆性をもつ．

また，定理 1 と局所的な逆文の生成によって，逆プログラムを常に生成可能であるため，Janus は可逆プログラミング言語であるといえる．

6 おわりに

現在の Janus と我々の提案する Janus についての比較を具体的なプログラムを用いて行った．整数 n の階乗を求めるプロシージャ`factorial` を，文献 [1] で提案されている Janus で記述したものが図 6 である．なお，`a` は階乗の計算結果が格納される変数であり，`tmp1`, `tmp2` は計算過程を一時的に保存する変数である．また，`n` の初期値は 1 以上に限定し，`tmp1` の初期値は `n` が，`a` の初期値は 0 が与えられるものとする．このプログラムを，我々の提案する Janus では図 7 のように記述可能である．変数だけでなく，場合に応じて式を実引数に記述できるようになった．この例では，プロシージャ本体の行数が 8 行から 4 行になり，変数も 1 つ減らすことができている．

```
procedure factorial(n, tmp1, a)
  local tmp2 = tmp1*(n-1)
    if n = 1
    then a += tmp1
    else n -= 1
         call factorial(n, tmp2, a)
         n += 1
    fi n = 1
  delocal tmp2 = tmp1*(n-1)
```

図 6 状態モデルの Janus で記述したプログラム

我々は，Janus に環境記憶域モデルを適用し，引数渡し機構の拡張を提案した．新たに左辺値を定義したことによって，実引数として変数だけでなく，式を渡すことも可能となった．また，配列を連続的に扱うことも可能となった．さらに，値渡し，参照渡し，名前渡しをそれぞれ可逆化し，Janus で実現した．これにより，これらの引数渡し方法を場合によって使い分けることができるようになった．また，これら 3 種類の引数渡し方法の組み合わせ自由な複数引数渡し機構を実現することができた．これらの成果として，一般に変数の数を減らすことができるようになった．また，状態モデルの Janus では記述不可能な Jensen's Device のようなプログラムを記述できるようにもなった．以上の結果より，我々の提案によって Janus の表現力は向上したといえる．

```
procedure factorial(val n, val tmp1, ref a)
  if n = 1
  then a += tmp1
  else call factorial(n-1, tmp1*(n-1), a)
  fi n = 1
```

図 7 環境記憶域モデルの Janus で記述したプログラム

また，我々は，環境記憶域モデルの Janus についての可逆性を保証した．これにより，3 種類の引数渡し方法の可逆化に成功したことが確認できた．また，本稿で提案する Janus についても，同一のプロシージャにおいて呼出しと逆呼出しが可能である．プロシージャ逆呼出しの規則の前提に，プロシージャ呼出しの判定を用いることは，プログラミング言語全体を注意深く可逆に設計してはじめてできることである．さらに，Janus は汎用的な可逆プログラミング言語であるため，別の可逆プログラミング言語にも本研究の成果を適用することが可能である．

今後の課題としては，本稿で表現力を向上させた Janus に対して，一般的な命令型プログラミング言語で用いられている機構を更に可逆化し，導入することが挙げられる．たとえば，副作用をもたない標準ライブラリ関数や，スタックなどが考えられる．

参考文献

[1] Yokoyama, T., Axelsen, H.B. and Glück, R.: Principles of a Reversible Programming Language, Proc. *Computing frontiers*, pp.43–54, ACM (2008).

[2] Yokoyama, T. and Glück, R.: A Reversible Programming Language and its Invertible Self-Interpreter, Proc. *Partial evaluation and semantic-based program manipulation*, pp.144–153, ACM (2007).

[3] Lutz, C.: Janus: a time-reversible language. *Letter to R. Landauer* (1986).

[4] Dijkstra, E.W.: Letter to the Editor: Defense of ALGOL 60, *Communications of the ACM*, Vol.4, No.11, pp.502–503 (1961).

[5] 田中秀明，新海由侑，横山哲郎：引数渡し機構をもつ可逆プログラミング言語の可逆性（to be published）．