情報処理学会論文誌　プログラミング　Vol.7　No.4　21–36 (Aug. 2014)

可逆プログラミング言語の引数渡し機構の拡張

新海 由侑[1]　田中 秀明[1]　横山 哲郎[2,a)]

受付日 2014年1月14日, 採録日 2014年6月5日

概要：本稿では，可逆プログラミング言語 Janus の引数渡し機構を拡張して，その拡張言語が可逆であることを示す．参照渡しのモデル化には，同一の参照を持つかもしれない複数の変数名が同じ記憶場所を指せるメモリモデルが必要である．これは既存の Janus で用いられていた状態モデルでは直接には扱うことができない．この問題を解決するために，我々は，環境・記憶域モデルを導入して意味論を改良した．この結果，プロシージャの実引数には，局所変数や局所配列変数だけでなく大域変数や添字付き配列変数の参照，および同一の参照を持つ複数の構文対象も渡せるようになった．拡張言語が可逆であることの保証には，環境と記憶域の更新が可逆であることの保証が必要である．これらの更新は，制限された可逆操作のみ用いて実現されることで，可逆であることが保証される．我々は，拡張言語がほかにも既存の Janus の良い性質を保っていることを示す．すなわち，任意の文やプロシージャ呼び出しの逆実行ができること，および任意の文に対して逆文が存在してそれを求めるプログラム逆変換器が構成できることを示す．さらに，我々は，プログラミング言語を可逆にするために課されていた代入の構文上の制限を緩和する．このことが原因で起こる記憶域の不正な更新は，大域のプログラム解析をしなくても，効率的に検知することができる．

キーワード：可逆プログラミング言語，Janus，参照渡し，環境・記憶域モデル

Extended Parameter Passing Mechanisms of a Reversible Programming Language

Yoshiyuki Shinkai[1]　Hideaki Tanaka[1]　Tetsuo Yokoyama[2,a)]

Received: January 14, 2014, Accepted: June 5, 2014

Abstract: The extended pass-by-reference parameter passing mechanism of reversible programming language Janus is presented and its reversibility is shown. The states used in the memory model of the existing Janus cannot directly deal with the multiple variables sharing the same memory locations. To remedy this, the state model is replaced with the environment-store model and the semantics of the language is refined. In the extended language, we can pass not only local variables and local array variables to the arguments of procedures but the references of global variables, indexed array variables and such syntactic objects that have the same references. We make the extended Janus reversible by imposing restrictions on the updates of environments and stores. The extended language also preserves several properties of Janus; any sentences and procedure calls can be inversely executed, and the inverse sentences for any sentences exist and can be constructed by simple program transformation. Furthermore, we relax the restrictions on the syntax of reversible assignments. The invalid store updates because of the extension can be efficiently detected during the execution without the necessity of global program analysis. The refined semantics is a simple extension to the existing one in the sense that the programs in the existing syntax have the same meaning under both semantics.

Keywords: reversible programming languages, Janus, pass by reference, environment-store model

[1] 南山大学情報理工学部ソフトウェア工学科
Department of Software Engineering, Faculty of Information Sciences and Engineering, Seto, Aichi 489–0863, Japan
[2] 南山大学理工学部ソフトウェア工学科
Department of Software Engineering, Faculty of Science and Engineering, Seto, Aichi 489–0863, Japan
[a)] tyokoyama@acm.org

1. はじめに

現在のほとんどすべての高水準プログラミング言語では，引数を渡すことのできるプロシージャや関数によって，再利用が容易なソフトウェアの部品を作ることができる．た

とえば，実引数を仮引数に渡す方式の 1 つである参照渡しは，FORTRAN や Pascal をはじめとして広く使われている．一方，可逆プログラミング言語では参照渡しが採用されたとされる言語は，我々の知る範囲では Janus [39] と R 言語 [9] のみである．しかし，文献 [39] の Janus には，大域変数が使えず，プロシージャ呼び出しの実引数は互いに異なる変数もしくは配列変数でなければならないといった制限がある．また，R 言語は，形式的意味論が定義されていないので実際の動作は明確ではなく，可逆性の保証はコンパイル後の低レベル可逆プログラミング言語の可逆性に依っている．

可逆プログラミング言語 Janus において，もし配列変数への代入とプロシージャ呼び出しの構文上の制約が緩められたら，より多くの文を可逆プログラミングに用いることができる．また，スコープの異なる変数を使用できるようにすることでより適切な結合度のプログラムを書くことが容易になる．このことにより，より Janus を用いた可逆プログラミングが容易になることが期待できる．

本稿では，大域変数と局所変数を用いられ，プロシージャ呼び出しの引数に添字付き配列変数を渡したり同一の配列変数や変数を渡したりでき，配列変数の代入に同一の配列変数が用いられるようにするような拡張を Janus に行う．このことで，返り値の使えない Janus において何重にもネストしたプロシージャ呼び出しの計算結果を大域変数に格納して利用したり，配列全体ではなく配列の特定の要素を渡すことでより適切な結合度のプロシージャが実現できたりするようになることが期待される．

しかし，こうした構文上の制約の緩和は変数のエイリアスを出現させてしまうという問題がある．既存の Janus は，エイリアスが出現すると，可逆ではなくなってしまう．また，既存の状態モデルを用いた Janus の意味論では，可逆性を保証しながら効率的にエイリアスを扱うことができない．我々は，環境・記憶域モデル [30] を用いた形式的意味論を用いて，拡張された Janus が可逆であることの証明を行う．

以下，それぞれの点をくわしく見ていくことにする．

配列変数への代入文の構文制約の緩和

可逆なプログラミング言語には，実行の任意の時点の計算状態から，その直後の計算状態が一意に定まるとともに，その直前の計算状態が一意に定まるよう構文上の制約が与えられている．たとえば，非可逆な C 言語プログラムにおける文 `x -= x;` を考える．変数 `x` の値は，この文の実行直前に 2 であろうが 3 であろうが，この文の実行直後に必ず 0 になり実行直前の値は一意に定まらない．したがって，この文の実行直後の計算状態から，実行直前の計算状態は一意に定まらない．可逆プログラミング言語である Janus は，代入の構文において，左辺で代入が行われる変数が右辺に出現してはならないという制約があり，文 `x -= x` は記述できないようになっている．

同様に，Janus では，代入の構文において，左辺で代入が行われる配列変数が，左辺の添字や右辺に出現してはならないという制約がある．したがって，フィボナッチ数列を配列を用いて計算するときに出現する，`fib[i]` と `fib[i+1]` の合計値を `fib[i+2]` に格納するような

```
fib[i+2] += fib[i+1] + fib[i]
```

という文は記述できない．

この構文上の制約はプログラミング言語を可逆にするという目的に対しては厳しすぎる．たとえば，上記の文の実行直後に

```
fib[i+2] -= fib[i+1] + fib[i]
```

を実行すれば必ず元の計算状態を一意に求めることができるからである．我々は，代入の行われる配列変数が右辺式にも現れる文も実行できるように，Janus の構文の制約を緩めてその部分の意味論を拡張したい．

エイリアスの導入

プログラムの実行中に，複数の変数が記憶域の同一の記憶場所を指すことがある．こうした変数はエイリアスと呼ばれる．参照渡しのプログラミング言語では，プロシージャ呼び出しによってエイリアスが生まれることがある．

可逆プログラミング言語 Janus は，エイリアスが導入されると文が可逆でなくなることがあるという問題がある．たとえば，単純な複合代入演算文

```
x -= y
```

の実行において，変数 `x` と `y` が同一の記憶場所を指していた場合を考える．この文は，上記の C 言語の文 `x -= x;` と同様な理由で可逆でない．すなわち，任意の `x` と `y` の値に対して，この文の実行直後に `x` と `y` の値はともに 0 になる．したがって，この文の実行直前の計算状態は一意に定まらない．この原因は，`x` と `y` がエイリアスであること，つまり `x` と `y` が同一の左辺値を持つことである．エイリアスの解析方法は数十年以上前から研究されてきているが [5]，添字付き配列変数への代入が行われる場合は，動的な情報を用いなければ保守的にしか解析をすることしかできない．

文献 [39] の Janus では，引数付きプロシージャがあるにもかかわらず，エイリアスは出現することがない．これは次の 3 つの理由による．第 1 に，大域変数が存在しない．第 2 に，プロシージャの仮引数には同一の変数を渡すことが制限されている．第 3 に，プロシージャ呼び出し以外の構文ではエイリアスが生み出されることはない．一方，文献 [42] の Janus では，大域変数はあるが引数付きプロシー

ジャが存在しないため，エイリアスが生み出されることはない．これらの条件は，可逆性を保証することができている一方で，プログラミングの表現を制約している．

我々は，文献 [39] の Janus に，大域変数を導入し，同一の変数をプロシージャに複数個渡せるようにする．これらの構文に対応する推論規則を追加するような Janus の意味論の拡張を行う．

環境・記憶域モデルの導入

非可逆プログラミング言語においても，長年，エイリアスは静的解析の対象となってきた [5]．たとえば，エイリアスは，コンパイラによる最適化の妨げとなるからである．プログラムの実行前に静的にエイリアスを解析することでレジスタに割り付ける変数を効果的に決められるようになる．

可逆プログラミング言語においては，このような目的のエイリアスの静的解析に加えて，実行時に可逆性を確認するための左辺値の静的・動的解析が有用である．たとえば，上で出てきた文 `x -= y` が実行されるとき，静的に `x` と `y` がエイリアスであることが分かれば，コンパイラはプログラマに修正をもとめることができ，また静的に `x` と `y` がエイリアスであるかもしれないことが分かれば，コンパイラはプログラマに警告をすることができ，さらに動的に `x` と `y` がエイリアスであるかもしれないことが分かれば，ランタイムシステムはこの文を可逆に実行できないことをユーザに示し計算を停止することができる．同様な理由から，可逆プログラミング言語においては，配列変数への代入においても左辺値の解析が有用である．

文献 [39] の Janus では，意味論において状態モデルが用いられていた．状態は，変数と値の対応関係を管理するのに用いられる．状態モデルでは，左辺値を明示的に記憶しておくことができない．したがって，実行時にはランタイムシステムがエイリアスや左辺値の解析を行うことができず，非可逆な計算を検知することができなかった．

我々は，変数と値の対応関係しか管理できない状態モデルの代わりに，左辺値をうまくあつかえる環境・記憶域モデルを用いる．このモデルでは，左辺値が意味領域に加えられているので，このような解析を効率的に行うことができる．複合代入文は，実行時に環境と記憶域だけの情報から可逆でないことの判定が行えるようになる．制約なしに環境と記憶域を用いた推論規則は，一般に可逆にはならない．我々は，環境の形を制限し，記憶域の更新に可逆な更新のみを用いるといった制約を与えることで，推論規則の可逆性を保証する．

さらに，本稿では，文に関する推論規則を可逆に実現することはもちろん，これまで形式化されてこなかった Janus における変数の初期化，プログラム実行の初期化に関する可逆な推論規則を環境・記憶域モデルを用いて形式化する．これらの可逆性は可逆な処理系を実現する際に必要になるものであり重要である．

論文の構成

2 章では，本稿で提案する言語の構文と意味論を与える．3 章では，この提案言語が可逆であることや可逆言語として良い性質を持っていることを示す．4 章では，複合代入文の形によって行える最適化を議論する．5 章では，関連研究を述べる．6 章では，本稿のまとめを行う．

2. 言語

本稿で扱う可逆プログラミング言語 Janus は，文献 [42] と文献 [39] のものをもとにしている．構文は，文献 [42] のものに文献 [39] で導入されたプロシージャの引数を構文上の制約を緩めて追加した．意味論は，文献 [42] と文献 [39] で用いられていた状態モデルを環境・記憶域モデル [30] に拡張した．なお，繰返しの推論規則は文献 [42] のものをもとにした．

2.1 構文

本稿で用いる Janus の構文規則を**図 1** に示す．

Janus プログラム $d^*\ p^+$ は，0 個以上の変数と配列の宣言 d^*，および 1 個以上のプロシージャの定義 p^+ からなる．プロシージャの定義 `procedure` $q(x_1,\ldots,x_n)$ s は，プロシージャ名 q，0 個以上の仮引数の列 $x_1,\ldots,x_n$，およびプロシージャの本体である文 s からなる．文 s は，代入 e_l $\odot$`=` e，条件 `if` e_1 `then` s_1 `else` s_2 `fi` e_2，繰返し `from` e_1 `do` s_1 `loop` s_2 `until` e_2，プロシージャ呼び出し `call` $q(e_{l1},\ldots,e_{ln})$ と逆呼び出し `uncall` $q(e_{l1},\ldots,e_{ln})$，文の繰返し $s_1\ s_2$，および空文 `skip` からなる．式 e は，整定数 c，変数 x，添字式付き配列変数 x`[`e`]`，および二項演算式 $e_1 \otimes e_2$ からなる．左辺式 e_l は，変数 x および添字式付き配列変数 x`[`e`]` からなる．

Janus プログラムには次のような構文上の制約がある．Janus プログラムには，必ず 1 つ `main` プロシージャが含まれる．`main` プロシージャは仮引数を持たない．プログラムの実行は `main` プロシージャの本体から始まる．同一プログラムにおいて各プロシージャのプロシージャ名 q は，互いに異なっていなければならない．同一プロシージャの仮引数は互いに異なっていないとならない．プロシージャ呼び出し `call` $q(e_{l1},\ldots,e_{ln})$ およびプロシージャ逆呼び出し `uncall` $q(e_{l1},\ldots,e_{ln})$ で用いられるプロシージャ名 q は，同一プログラムで定義されていなければならない．プロシージャ q の呼び出しおよび逆呼び出しの実引数の個数 n は，対応するプロシージャ q の定義 `procedure` $q(x_1,\ldots,x_n)$ s における仮引数の個数 n と一致しなければならない．

任意の 2 つの識別子は，同一であるかを判定できるものとする．可逆代入で e_l $\odot$`=` e_1 は，左辺式 x_1，x_2`[`e_2`]` に現れる変数 x_1 や配列変数 x_2 が右辺の式 e_1 に現れてもかま

$prog$	$\in$ Progs	::=	$d^*\ p^+$	Janus プログラム
d	$\in$ Vdecs	::=	$x \mid x\texttt{[}c\texttt{]}$	スカラーと配列
x, y	$\in$ Vars	::=	`a` $\mid$ `b` $\mid \cdots$	変数
p	$\in$ Procs	::=	`procedure` $q\texttt{(}x\texttt{,}\ldots\texttt{,}x\texttt{)}\ s$	プロシージャの定義
q	$\in$ PIds	::=	`a` $\mid$ `b` $\mid \cdots$	プロシージャ名
s	$\in$ Stms	::=	$e_l\ \odot\texttt{=}\ e$	代入
		$\mid$	`if` e `then` s `else` s `fi` e	条件
		$\mid$	`from` e `do` s `loop` s `until` e	繰返し
		$\mid$	`call` $q\texttt{(}e_l\texttt{,}\ldots\texttt{,}e_l\texttt{)}$	プロシージャ呼び出し
		$\mid$	`uncall` $q\texttt{(}e_l\texttt{,}\ldots\texttt{,}e_l\texttt{)}$	プロシージャ逆呼び出し
		$\mid$	$s\ s$	文の並び
		$\mid$	`skip`	空文
e	$\in$ Exps	::=	$c \mid x \mid x\texttt{[}e\texttt{]} \mid e \otimes e$	式
e_l	$\in$ Lexps	::=	$x \mid x\texttt{[}e\texttt{]}$	左辺式
c	$\in$ Cons	::=	$\cdots \mid$ `-1` $\mid$ `0` $\mid$ `1` $\mid$ `2` $\mid \cdots$	整定数
$\odot$	$\in$ ModOps	::=	`+` $\mid$ `-`	演算子
$\otimes$	$\in$ Ops	::=	$\odot \mid$ `*` $\mid$ `/` $\mid$ `%` $\mid$ `&&` $\mid$ `||` $\mid$ `<` $\mid$ `>` $\mid$ `=` $\mid$ `!=` $\mid$ `<=` $\mid$ `>=`	演算子

図 1　構文規則
Fig. 1　Syntax rules.

$v \in$ Vals	$=$	$\mathbb{Z}$	値
$l \in$ Lvals	$=$	$\mathbb{N}$	左辺値
$\gamma \in$ Envs	$=$	[Vars $\times$ Lvals]	環境
$\sigma \in$ Stores	$=$	Lvals $\to$ Vals	記憶域
$\Gamma \in$ Pmaps	$=$	PIds $\rightharpoonup$ Procs	プロシージャ環境

図 2　意味領域
Fig. 2　Semantic domains.

わない．また，左辺式 $x_2\texttt{[}e_2\texttt{]}$ 中の配列変数 x_2 がその添字 e_2 に現れてもかまわない．配列変数の添字式 e_2 は負の値や配列 x のサイズを超える値をとりうるが，構文上は任意の式が添字に現れてかまわない．プロシージャの実引数は，左辺式であり，変数に加えて配列変数や添字付き配列変数も許される．

簡単にするために，データ型は整数型のみが用いられている．配列の宣言 $x\texttt{[}c\texttt{]}$ の添字 c は正の定数でなければならない．定数には，整数を用いる．配列の添字は 0 から始まる．すべての大域変数の値および配列の要素の値は，実行前には 0 に初期化されている．式において配列の添字に現れるのは整数を値として持つ式のみが許される．

既存の Janus [19], [39], [42] からの拡張部分は，大域変数とプロシージャの引数が両方導入されていること，プロシージャ呼び出しおよびプロシージャ逆呼び出しの実引数に左辺式を渡せること，可逆代入の左辺値の制限が緩められていること，および整定数の上限と下限がないことである．

2.2　意味論

本稿で用いる Janus の意味論を定める．

意味領域

意味領域を**図 2** に示す．値は，整数とする．左辺値は，記憶域上の記憶場所を表す自然数（0 以上の整数）とする．

引数渡し機構を自然に実現するために，環境および記憶域を導入する．

環境は，変数と記憶場所の組の列である．変数 x と記憶場所 l の組は $x \mapsto l$ と表記される．本稿の Janus では，変数と配列変数の初期化はプログラムの実行の初期にのみ行われ，文の実行中に新たな左辺値が生成されることがない．

記憶域 σ :: Lvals $\rightharpoonup$ Vals は記憶場所から値への関数である．記憶域の読み出し $\sigma(l)$ は，記憶場所 l にある値を返す．記憶域の更新には，直和 $\uplus$ を用いる：

前方決定的な判断：

$$\text{式の評価の判断} \quad \gamma, \sigma \vdash_{expr} e \Rightarrow v$$
$$\text{左辺式の評価の判断} \quad \gamma, \sigma \vdash_{lexpr} e_l \Rightarrow l$$

可逆な判断：

$$\text{変数と配列変数の初期化の判断} \quad \vdash_{defv} \langle e_l, (\gamma, \sigma)\rangle \Rightarrow \gamma', \sigma'$$
$$\text{文の実行の判断} \quad \gamma \vdash^{\Gamma}_{stmt} \langle s, \sigma\rangle \Rightarrow \sigma'$$
$$\text{プログラムの実行の判断} \quad \vdash^{\Gamma}_{prog} \langle prog, (\gamma, \sigma)\rangle \Rightarrow \gamma', \sigma'$$

条件付きで可逆な判断（条件は $\gamma, \sigma \vdash_{expr} e_1 \not\Rightarrow 0$ および $\gamma, \sigma' \vdash_{expr} e_2 \not\Rightarrow 0$）：

$$\text{繰返しの実行の判断} \quad \gamma \vdash^{\Gamma}_{loop1} \langle (e_1, s_1, s_2, e_2), \sigma\rangle \Rightarrow \sigma'$$
$$\gamma \vdash^{\Gamma}_{loop2} \langle (e_1, s_1, s_2, e_2), \sigma\rangle \Rightarrow \sigma'$$

図 3　判断
Fig. 3　Judgments.

$$(\sigma \uplus \{l \mapsto v\})(l') = \begin{cases} v & \textbf{if } l = l' \\ \sigma(l') & \textbf{otherwise} \end{cases}$$

ここで，記憶場所と値の組の集合を記憶場所から値への関数と同一視している．記憶場所 *next* は，以降がまだ割当ての行われていない記憶域を $\sigma(next)$ によって指すために用いることにする．以降で見る推論規則は，この制約が入力において満たされるのであれば，出力においても満たされるように設計する．

プロシージャ環境は，プロシージャ名とプロシージャの対応を表す関数である．プロシージャ環境 Γ の更新は後置演算子 $[q \mapsto p]$ で行う：

$$(\Gamma[q \mapsto p])(q') = \begin{cases} p & \textbf{if } q = q' \\ \Gamma(q') & \textbf{otherwise} \end{cases}$$

判断

図 3 のように判断を定義する．式の評価の判断 $\gamma, \sigma \vdash_{expr} e \Rightarrow v$ は，環境 γ と記憶域 σ の下で式 e が値 v に評価されることを示す．左辺式の評価の判断 $\gamma, \sigma \vdash_{lexpr} e_l \Rightarrow l$ は，環境 γ と記憶域 σ の下で左辺式 e_l が記憶場所 l に評価されることを示す．変数と配列変数の初期化の判断 $\vdash_{defv} \langle d, (\gamma, \sigma)\rangle \Rightarrow \gamma', \sigma'$ は，d で表される変数もしくは配列のために記憶域上の記憶場所が確保され，環境を用いて変数名から値を得たり更新したりできるようにされることを示す．文の実行の判断 $\gamma \vdash^{\Gamma}_{stmt} \langle s, \sigma\rangle \Rightarrow \sigma'$ は，プロシージャ環境 Γ と環境 γ の下で，文 s が実行された結果，記憶域 σ が σ' に更新されることを示す．繰返しの実行の判断 $\gamma \vdash^{\Gamma}_{loop1} \langle (e_1, s_1, s_2, e_2), \sigma\rangle \Rightarrow \sigma'$ および $\gamma \vdash^{\Gamma}_{loop2} \langle (e_1, s_1, s_2, e_2), \sigma\rangle \Rightarrow \sigma'$ によって，プロシージャ環境 Γ と環境 γ の下で，繰返し文 `from` e_1 `do` s_1 `loop` s_2 `until` e_2 の実行過程で，記憶域 σ が σ' に更新されることを示す．プログラムの実行の判断 $\vdash^{\Gamma}_{prog} \langle prog, (\gamma, \sigma)\rangle \Rightarrow \gamma', \sigma'$ は，プロシージャ環境 Γ の下で，プログラム *prog* が実行された結果，環境 γ と記憶域 σ がそれぞれ，γ' と σ' に更新されることを示す．

$\gamma, \sigma \vdash_{expr} e \not\Rightarrow v$ は，v 以外の値に式 e が評価されることを表す．すなわち，$\gamma, \sigma \vdash_{expr} e \not\Rightarrow v$ は，ある v'（$\neq v$）が存在して，$\gamma, \sigma \vdash_{expr} e \Rightarrow v'$ であることを表す．

式の評価の判断では γ と σ を入力と呼び，v を出力と呼ぶ．左辺式の評価の判断では γ と σ を入力と呼び，l を出力と呼ぶ．また，$\cdot \vdash_{\cdot} \langle \cdot, in\rangle \Rightarrow out$ という形をした判断では，*in* を入力と呼び，*out* を出力と呼ぶ．

判断は，他の要素が決まったとき，入力から出力が一意に定まるのであれば前方決定的，出力から入力が一意に定まるのであれば後方決定的という．前方決定的かつ後方決定的な判断は可逆という．また，結論の判断が前方決定的，後方決定的，および可逆になる推論規則はそれぞれ前方決定的，後方決定的，および可逆という．可逆な判断や可逆な推論規則を持つことが可逆プログラミング言語の特徴である．

式の評価

式の評価の推論規則を**図 4** に示す．

推論規則 Con では定数の評価が行われている．意味関数 $[\![\cdot]\!] :: \text{Cons} \to \text{Vals}$ は，任意の定数 c に対して，$[\![c]\!]$ は定数 c に対応する値を返すものとする．関数 $[\![\cdot]\!]$ は全域で定義されているものとする．

推論規則 Var では変数の評価が行われている．前提では，左辺式の評価の判断 $\gamma \vdash_{lexpr} x \Rightarrow l$ が出現している．変数に対する左辺式の評価の判断は，**図 5** の LVar1 および LVar2 に定義されている．環境に格納されている変数と記憶場所の組を右側から順に見ていく．最初に評価を行っている変数 x と一致する変数を持つ組 $x \mapsto l$ が見つかったら，l を出力とする．記憶域 σ においてその記憶場所の対応する値が得られる．

推論規則 BinOp では二項演算子 $\otimes$ の評価が行われている．算術演算子 $\odot$ の意味は，それぞれの演算子に対応する

$$\frac{}{\gamma,\sigma \vdash_{expr} c \Rightarrow [\![c]\!]}\textsc{Con} \qquad \frac{\gamma,\sigma \vdash_{lexpr} x \Rightarrow l}{\gamma,\sigma \vdash_{expr} x \Rightarrow \sigma(l)}\textsc{Var}$$

$$\frac{\gamma,\sigma \vdash_{expr} e_1 \Rightarrow v_1 \quad \gamma,\sigma \vdash_{expr} e_2 \Rightarrow v_2}{\gamma,\sigma \vdash_{expr} e_1 \otimes e_2 \Rightarrow [\![\otimes]\!](v_1,v_2)}\textsc{BinOp} \qquad \frac{\gamma,\sigma \vdash_{lexpr} x \Rightarrow l \quad \gamma,\sigma \vdash_{expr} e \Rightarrow v}{\gamma,\sigma \vdash_{expr} x\texttt{[}e\texttt{]} \Rightarrow \sigma(l+v)}\textsc{Arr}$$

図 4　式の評価の推論規則

Fig. 4　Inference rules for the evaluation of expressions.

$$\frac{\gamma, x \mapsto l,\ \sigma \vdash_{lexpr} x \Rightarrow l}{\gamma,\sigma \vdash_{lexpr} x \Rightarrow l}\textsc{LVar1} \qquad \frac{x \neq y \quad \gamma,\sigma \vdash_{lexpr} x \Rightarrow l}{\gamma, y \mapsto l',\ \sigma \vdash_{lexpr} x \Rightarrow l}\textsc{LVar2}$$

$$\frac{\gamma,\sigma \vdash_{lexpr} x \Rightarrow l \quad \gamma,\sigma \vdash_{expr} e \Rightarrow v}{\gamma,\sigma \vdash_{lexpr} x\texttt{[}e\texttt{]} \Rightarrow l+v}\textsc{LArr}$$

図 5　左辺式の評価の推論規則

Fig. 5　Inference rules for the evaluation of left expressions.

数学上の演算を行う．すなわち，

$$[\![\texttt{+}]\!](v_1,v_2) = v_1 + v_2$$
$$[\![\texttt{-}]\!](v_1,v_2) = v_1 - v_2$$

と定義する．関数 $[\![\texttt{+}]\!]$ および $[\![\texttt{-}]\!]$ は，第 1 引数について双射になっている．任意の v に対して $\lambda v_1.\ [\![\texttt{+}]\!](v_1,v)$ と $\lambda v_1.\ [\![\texttt{-}]\!](v_1,v)$ は互いに逆関数になっていることを Ass の後方決定性を議論するときに用いる．論理演算の意味は

$$[\![\texttt{>}]\!](v_1,v_2) = \begin{cases} 0 & \textbf{if } v_1 \leq v_2 \\ 1 & \textbf{if } v_1 > v_2 \end{cases}$$

$$[\![\texttt{==}]\!](v_1,v_2) = \begin{cases} 0 & \textbf{if } v_1 \neq v_2 \\ 1 & \textbf{if } v_1 = v_2 \end{cases}$$

と定義する．他の演算子も同様に定義する．関数 $[\![\otimes]\!]$ は全域で定義されているものとする．

推論規則 Arr は，添字付き配列変数の値を評価する．配列変数 x の記憶場所 l から添字の値 v の分だけずれた記憶場所の値を記憶域 σ から得る．配列外を参照している場合でも，指定された記憶域の記憶場所の値を得ることに注意してほしい．ただし，負の値で示される記憶場所は存在せず，$\sigma(l+v)$ が未定義になることで，この推論規則が適用できないことがあることに注意してほしい．

左辺式の評価

左辺式の評価の推論規則を図 5 に示す．推論規則 LVar1 および LVar2 は変数の左辺値を，推論規則 LArr は添字付き配列変数の左辺値をそれぞれ出力とする．

文の実行

文の実行の推論規則を**図 6** に示す．

代入の意味は，推論規則 Ass で定める．直感的には，代入文 `x += e` は C 言語における複合代入演算の文 `x += e;` すなわち `x = x + e;` に対応する．代入では，記憶域を変化させる前後の左辺の左辺値が一致することを確認する．また，記憶域を変化させる前後の右辺の値が一致することも確認する．まず，左辺式が評価されると記憶場所 l が得られ，右辺式の評価では値 v が得られる．次に，この値 v を用いて記憶場所 l に格納された値 $\sigma(l) = v_1$ を $[\![\odot]\!]$ で更新して，記憶域 σ' を得る．さらに，更新後の記憶域の下で左辺式と右辺式を評価し直して，左辺の左辺値と右辺の値は，記憶域の更新の前後で一致することを検査する．

たとえば，$\gamma_1 \vdash_{lexpr} \texttt{x} \Rightarrow l_1$ および $\sigma_1(l_1) = 2$ となる環境 γ_1 と記憶域の下で `x += 1` が実行された場合を考える，直感的には `x` へ `x+1` の値を代入することになる．具体的には，推論規則 Ass に従って計算は進む．推論規則 LVar1 と LVar2 より `x` の左辺値である記憶場所 l_1 が得られる．右辺式 `1` は推論規則 Con より値 1 を持つ．記憶域は $\sigma_1 = \sigma_1'' \uplus \{l_1 \mapsto 2\}$ から $\sigma_1' = \sigma_1'' \uplus \{l_1 \mapsto 2+1\}$ に更新される．左辺は変数 `x` であるので記憶場所は l_1 のままであり，記憶域の更新の前後で右辺の定数は値は変化しない．

`x` が `y` のエイリアスであるとき，`x -= y` は，`x` が 0 である場合を除くと，右辺の値が変化してしまうので Ass を適用できない．左辺が添字付き配列変数 x`[`e`]` の場合は，左辺の左辺値の計算が変数の場合と異なる．左辺値は x の左辺値に e の値を足したものである．したがって，e の値が記憶域の更新の前後で異なれば，この左辺値が変化してしまう可能性がある．たとえば，`x[0]` の値が 0 であるような環境と記憶域の下で `x[x[0]] += 1` を実行することはできない．

推論規則 IfTrue, IfFalse, LoopMain, Loop1Base, Loop2Base, Loop1Rec, Loop2Rec, Skip, Seq は，文献 [39], [42] の対応する推論規則をもとに，状態モデルを環境・記憶モデルに拡張することで，単純に作成した．条件文と繰返し文については，文献 [39], [40], [42] において用いられている可逆制御流図を用いて直感的な説明を行う．以下では，C 言語と同様に，非ゼロの値が真を，値 0 が偽を表すものとする．

条件文 `if` e_1 `then` s_1 `else` s_2 `fi` e_2 について考える．**図 7** (a) に条件文の制御流図を示す．通常の制御流図と同

$$\frac{\begin{array}{c}\gamma,\sigma\vdash_{lexpr} e_l\Rightarrow l \qquad \gamma,\sigma\vdash_{expr} e\Rightarrow v \\ \sigma=\sigma''\uplus\{l\mapsto v_1\} \qquad \sigma'=\sigma''\uplus\{l\mapsto [\![\odot]\!](v_1,v)\} \\ \gamma,\sigma'\vdash_{lexpr} e_l\Rightarrow l' \qquad \gamma,\sigma'\vdash_{expr} e\Rightarrow v' \\ l=l' \qquad\qquad v=v'\end{array}}{\gamma\vdash^{\Gamma}_{stmt}\langle e_l \odot\!\texttt{=}\ e,\sigma\rangle\Rightarrow\sigma'}\textsc{Ass}$$

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

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

$$\frac{\gamma,\sigma\vdash_{expr} e_1\not\Rightarrow 0 \quad \gamma\vdash^{\Gamma}_{loop1}\langle(e_1,s_1,s_2,e_2),\sigma\rangle\Rightarrow\sigma' \quad \gamma,\sigma'\vdash_{expr} e_2\not\Rightarrow 0}{\gamma\vdash^{\Gamma}_{stmt}\langle \texttt{from}\ e_1\ \texttt{do}\ s_1\ \texttt{loop}\ s_2\ \texttt{until}\ e_2,\sigma\rangle\Rightarrow\sigma'}\textsc{LoopMain}$$

$$\frac{\gamma\vdash^{\Gamma}_{stmt}\langle s_1,\sigma\rangle\Rightarrow\sigma'}{\gamma\vdash^{\Gamma}_{loop1}\langle(e_1,s_1,s_2,e_2),\sigma\rangle\Rightarrow\sigma'}\textsc{Loop1Base} \qquad \frac{\gamma\vdash^{\Gamma}_{stmt}\langle s_2,\sigma\rangle\Rightarrow\sigma'}{\gamma\vdash^{\Gamma}_{loop2}\langle(e_1,s_1,s_2,e_2),\sigma\rangle\Rightarrow\sigma'}\textsc{Loop2Base}$$

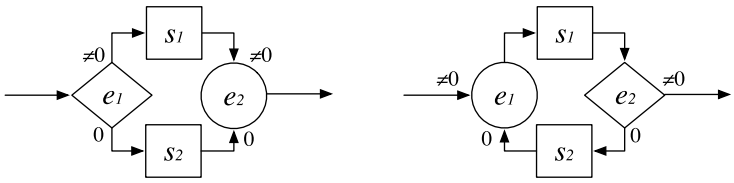
$$\frac{\begin{array}{c}\gamma\vdash^{\Gamma}_{stmt}\langle s_1,\sigma\rangle\Rightarrow\sigma' \qquad \gamma,\sigma'\vdash_{expr} e_2\Rightarrow 0 \\ \gamma\vdash^{\Gamma}_{loop2}\langle(e_1,s_1,s_2,e_2),\sigma'\rangle\Rightarrow\sigma'' \\ \gamma,\sigma''\vdash_{expr} e_1\Rightarrow 0 \quad \gamma\vdash^{\Gamma}_{stmt}\langle s_1,\sigma''\rangle\Rightarrow\sigma'''\end{array}}{\gamma\vdash^{\Gamma}_{loop1}\langle(e_1,s_1,s_2,e_2),\sigma\rangle\Rightarrow\sigma'''}\textsc{Loop1Rec} \quad \frac{\begin{array}{c}\gamma\vdash^{\Gamma}_{stmt}\langle s_2,\sigma\rangle\Rightarrow\sigma' \qquad \gamma,\sigma'\vdash_{expr} e_1\Rightarrow 0 \\ \gamma\vdash^{\Gamma}_{loop1}\langle(e_1,s_1,s_2,e_2),\sigma'\rangle\Rightarrow\sigma'' \\ \gamma,\sigma''\vdash_{expr} e_2\Rightarrow 0 \quad \gamma\vdash^{\Gamma}_{stmt}\langle s_2,\sigma''\rangle\Rightarrow\sigma'''\end{array}}{\gamma\vdash^{\Gamma}_{loop2}\langle(e_1,s_1,s_2,e_2),\sigma\rangle\Rightarrow\sigma'''}\textsc{Loop2Rec}$$

$$\frac{}{\gamma\vdash^{\Gamma}_{stmt}\langle\texttt{skip},\sigma\rangle\Rightarrow\sigma}\textsc{Skip} \qquad \frac{\gamma\vdash^{\Gamma}_{stmt}\langle s_1,\sigma\rangle\Rightarrow\sigma' \quad \gamma\vdash^{\Gamma}_{stmt}\langle s_2,\sigma'\rangle\Rightarrow\sigma''}{\gamma\vdash^{\Gamma}_{stmt}\langle s_1\ s_2,\sigma\rangle\Rightarrow\sigma''}\textsc{Seq}$$

$$\frac{\begin{array}{c}\Gamma(q)=\texttt{procedure}\ q(y_1,\ldots,y_n)\ s \\ (\gamma,\sigma\vdash_{lexpr} e_{li}\Rightarrow l_i)\ \text{for}\ 1\le i\le n \\ \gamma,y_1\mapsto l_1,\ldots,y_n\mapsto l_n\vdash^{\Gamma}_{stmt}\langle s,\sigma\rangle\Rightarrow\sigma' \\ (\gamma,\sigma'\vdash_{lexpr} e_{li}\Rightarrow l'_i)\ \text{for}\ 1\le i\le n \\ (l_i=l'_i)\ \text{for}\ 1\le i\le n\end{array}}{\gamma\vdash^{\Gamma}_{stmt}\langle\texttt{call}\ q(e_{l1},\ldots,e_{ln}),\sigma\rangle\Rightarrow\sigma'}\textsc{Call} \qquad \frac{\gamma\vdash^{\Gamma}_{stmt}\langle\texttt{call}\ q(e_{l1},\ldots,e_{ln}),\sigma'\rangle\Rightarrow\sigma}{\gamma\vdash^{\Gamma}_{stmt}\langle\texttt{uncall}\ q(e_{l1},\ldots,e_{ln}),\sigma\rangle\Rightarrow\sigma'}\textsc{Uncall}$$

図 6　文の実行の推論規則

Fig. 6　Inference rules for the execution of statements.



(a) 条件文　　(b) 繰返し文

図 7　可逆制御流演算子

Fig. 7　Reversible control flow operators.

様に，菱形はテストを，四角は文の実行を表す．丸は，可逆制御流図に特有のもので，アサーションを表す．テスト e_1 から上に伸びる辺には e_1 の値が非ゼロであるときに制御が通過することを示すラベル $\neq 0$ が付属している．同様に，テスト e_1 から下に伸びる辺には e_1 の値が 0 であるときに制御が通過することを示すラベル 0 が付属している．また，アサーションに伸びる辺にもそれぞれ制御が通過するときに満たすべきアサーションの値がラベルとして付属している．アサーションのおかげで条件文を通過した後において，どちらの節を制御が通過したかを知ることができる．条件文の形式的意味を推論規則 IfTrue と IfFalse で定める．テスト e_1 の評価結果が非ゼロである場合は then 節の文 s_1 を，ゼロである場合は else 節の文 s_2 を実行する．その後，then 節が実行された場合はアサーション e_2 が非ゼロの値であり，else 節が実行された場合はアサーション e_2 が 0 でなければならない．このようにアサーションは単なる注釈ではなく意味が定義されていることに注意されたい．

同様に，繰返し文 `from` e_1 `do` s_1 `loop` s_2 `until` e_2 について考える．図 7 (b) に繰返し文の制御流図を示す．繰返しに入るとき，アサーション e_1 の値は非ゼロでなければならない．次に do 節の s_1 が実行され，テスト e_2 の値に

$$\frac{\sigma = \sigma'' \uplus \{next \mapsto l\} \quad \sigma' = \sigma'' \uplus \{next \mapsto l+1\}}{\vdash_{defv} \langle x, (\gamma, \sigma)\rangle \Rightarrow \gamma, x \mapsto l,\ \sigma'}\ \textsc{DefVar}$$

$$\frac{[\![c]\!] = v \quad \sigma = \sigma'' \uplus \{next \mapsto l\} \quad \sigma' = \sigma'' \uplus \{next \mapsto l+v\}}{\vdash_{defv} \langle x[c], (\gamma, \sigma)\rangle \Rightarrow \gamma, x \mapsto l,\ \sigma'}\ \textsc{DefArr}$$

図 8　変数と配列変数の初期化の推論規則

Fig. 8　Inference rules for initializing variables and array variables.

よって制御流が分岐する．テスト e_2 の値が非ゼロであるときは，繰返しを抜ける．しかし，テスト e_2 が値 0 のときは，loop 節の s_2 を実行してアサーション e_1 に戻る．このときアサーション e_1 は，0 でなければならない．このことが繰返しの外からきた制御流と内からきた制御流を区別している．繰返し文の形式的意味は 5 つの推論規則で定める．推論規則 LoopMain は，繰返しに入るときと出るときにアサーション e_1 とテスト e_2 がそれぞれ非ゼロの値を持つことを示している．残りの推論規則は，s_1 と s_2 を交互に実行する繰返しについてのものである．添字が *loop1* である判断は s_1 から始まり s_1 で終わる繰返しを，添字が *loop2* である判断は s_2 から始まり s_2 で終わる繰返しを示す．LoopMain の前提で用いられた場合に繰返しの実行の判断が可逆になるために，結論の繰返しの実行の判断の入力と出力が，前提の繰返しの実行の判断の入力と出力にならないように設計されている．

推論規則 Skip と Seq によって，文のならびの意味を定める．

推論規則 Call と Uncall は，0 個以上の引数をともなったプロシージャ呼び出しと逆呼び出しに関するものである．

推論規則 Call は，n 個の変数の束縛と解放を行っている．それぞれの実引数 e_{li} は評価され対応する左辺値 l_i が求められる．それぞれの仮引数 y_i と対応する左辺値 l_i の組を環境に追加してプロシージャの本体を実行する．プロシージャの実行前後で，それぞれの実引数 e_{li} の左辺値が同じである必要がある．たとえば，

```
procedure f(a,b)
  a += 1
  b += 2
```

というプロシージャを `call f(i,x[i])` という文で呼び出したとする．ただし，`i` と配列 `x` のすべての要素は 0 であるとする．すると，呼び出し後には，`i` は 1，`x[0]` は 2 となるが，`x[i]` の左辺値が `x[0]` から `x[1]` の指す記憶場所に変化してしまう．この場合，推論規則の前提が満たされないので，プロシージャ呼び出しを行うことができない．

推論規則 Uncall は，引数付きプロシージャの逆呼び出しを実現するものである．プロシージャ逆呼び出しの意味は，プロシージャ呼び出しの入力と出力の記憶域を交換した判断で定められる．

文献 [39] では，可逆 FFT に頻出する加算と減算を同時に行う演算をマクロによって実現していた．`+-=(x,y)` は `x` を `x+y` に，`y` を `x-y` に更新する．さらにこれらの引数に添字付き配列を渡していた．これは既存の Janus のプロシージャ呼び出し `call addsub(re[k+j],e[k+j+m])` で代用することができない．添字付き配列変数を実引数とすることができないからである．本稿では，プロシージャの実引数の条件を緩めて，局所変数や局所配列変数だけでなく大域変数や添字付き配列変数の参照，および同一の参照を持つ複数の構文対象も渡せるようにした．その結果，上記のプロシージャ `addsub` の呼び出しも実現できるようになった．

変数と配列変数の初期化

図 8 は，変数と配列変数の初期化の推論規則である．推論規則 DefVar では，1 つの変数が新たに環境に割り当てられる．2 章では，記憶場所 *next* は，以降の記憶域上にまだ割当てが行われていない記憶場所を指すのに用いるとした．変数と初期化では，具体的には，まず，記憶場所 *next* に保持された記憶場所 l を求める．次に，記憶場所 *next* に保持された値を 1 つ増加させて，x と l の組を環境に追加する．推論規則 DefArr では，配列の初期化が行われており，一度に v 個の記憶場所が割り当てられている．それにともなって *next* の値が v だけ加算されている．また，配列名と配列の先頭の記憶場所の組が環境に追加される．変数と配列変数の初期化の判断は，逆方向に用いた場合，変数や配列の全要素の解放を行う．

プログラムの実行の判断

図 9 は，プログラムの実行の推論規則を示したものである．推論規則 Main では，前提を簡潔にするため，プロシージャ q_i の仮引数と文は省略した．

プログラム *prog* の実行の判断は以下のとおり：

$$\vdash^{\Gamma_{init}}_{prog} \langle prog, (\gamma_{init}, \sigma_{init})\rangle \Rightarrow \gamma, \sigma$$

ただし，γ_{init}，σ_{init}，および Γ_{init} は，それぞれプログラム実行前の初期の環境，記憶域およびプロシージャ環境である．

可逆プログラミング言語ではプログラム実行前には環境や記憶域にあたるものがゼロクリアされているものとする慣習がある．これにならって本稿では，環境には空列を，記憶域には任意の記憶場所 l に対して 0 であるようなものをそれぞれ用いることにする．ただし，任意のプログラム

$$\frac{(\vdash_{defv} \langle d_i, (\gamma_{i-1}, \sigma_{i-1})\rangle \Rightarrow \gamma_i, \sigma_i) \text{ for } 1 \le i \le k \quad (p_i = \texttt{procedure}\ q_i(\cdots)\ \cdots) \text{ for } 1 \le i \le n \quad \Gamma' = \Gamma[q_1 \mapsto p_1] \cdots [q_n \mapsto p_n] \quad \Gamma'(\texttt{main}) = \texttt{procedure main()}\ s \quad \gamma_k \vdash^{\Gamma'}_{stmt} \langle s, \sigma_k\rangle \Rightarrow \sigma}{\vdash^{\Gamma}_{prog} \langle d_1 \cdots d_k\ p_1 \cdots p_n, (\gamma_0, \sigma_0)\rangle \Rightarrow \gamma_k, \sigma}\ \textsc{Main}$$

図 9　プログラムの実行の推論規則
Fig. 9　Inference rules for the execution of programs.

実行前の初期の環境，記憶域およびプロシージャ環境に対して，後で見るようにプログラムの実行は可逆である．

3. 拡張した言語の可逆性

本章では，拡張した言語が，可逆プログラミング言語において重要な性質を持つことを示す．

まず，値の評価の一意性を示すために必要な，変数の左辺値の評価の一意性を示す．

補題 1（変数の左辺値の評価の一意性）．任意の変数 x，環境 γ，記憶域 σ，記憶場所 l，l' に対して，$\gamma, \sigma \vdash_{lexpr} x \Rightarrow l$ かつ $\gamma, \sigma \vdash_{lexpr} x \Rightarrow l'$ ならば，$l = l'$ である．

証明.　γ の構造に関する帰納法より明らか．　□

この補題を用いて式の評価の一意性を示す．

補題 2（式の評価の一意性）．任意の式 e，環境 γ，記憶域 σ，値 v，v' に対して，$\gamma, \sigma \vdash_{expr} e \Rightarrow v$ かつ $\gamma, \sigma \vdash_{expr} e \Rightarrow v'$ ならば，$v = v'$ である．

文献 [42] における状態モデルのもとでの式の評価の一意性の証明と同様にして，この補題は，式 e に関する構造帰納法で証明できる．任意の推論規則において，結論に現れる式よりも前提に現れる式の方が小さいからである．

証明.　式 e に関する構造帰納法より明らか．　□

式の評価は後方決定的ではない．たとえば，`x + y` は，`x` が 2，`y` が 1 であっても，`x` が 1，`y` が 2 であっても，3 になる．

値の評価で値が一意に定まったとの同様に，左辺式の評価において左辺値はつねに一意に定まる．

補題 3（左辺式の評価の一意性）．任意の左辺式 e_l，環境 γ，記憶域 σ，記憶場所 l，l' に対して，$\gamma, \sigma \vdash_{lexpr} e_l \Rightarrow l$ かつ $\gamma, \sigma \vdash_{lexpr} e_l \Rightarrow l'$ ならば，$l = l'$ である．

証明.　左辺式の場合分けで示す．

左辺式が変数 x である場合．補題 2 より明らかである．

左辺式が添字付き配列変数である場合．任意の γ，σ を仮定する．補題 1 より l が，前提と補題 2 より v が，それぞれ一意に定まる．$+$ が関数であることから $l + v$ は値であるならば一意に定まる．

以上で題意は示された．　□

左辺式の評価の判断は，その前提で出現している式の評価の判断が後方決定的でないことから，後方決定的でない．

補題 4（変数と配列変数の初期化の可逆性）．任意の変数もしくは定数の添字を持つ配列変数 d，式 e，環境 γ，γ'，γ''，および記憶域 σ，σ'，σ'' に対して，$\vdash_{defv} \langle d, (\gamma, \sigma)\rangle \Rightarrow \gamma', \sigma'$ かつ $\vdash_{defv} \langle d, (\gamma, \sigma)\rangle \Rightarrow \gamma'', \sigma''$ ならば，$\gamma' = \gamma''$ かつ $\sigma' = \sigma''$ である．また，任意の変数もしくは定数の添字を持つ配列変数 d，式 e，環境 γ，γ'，γ''，および記憶域 σ，σ'，σ'' に対して，$\vdash_{defv} \langle d, (\gamma', \sigma')\rangle \Rightarrow \gamma, \sigma$ かつ $\vdash_{defv} \langle d, (\gamma'', \sigma'')\rangle \Rightarrow \gamma, \sigma$ ならば，$\gamma' = \gamma''$ かつ $\sigma' = \sigma''$ である．

証明.　第 1 文は，記憶域 σ および記憶域の更新が関数であることと $[\![c]\!]$ の評価の一意性より成り立つ．

第 2 文の後方決定性は，推論規則によって場合分けする．

推論規則 DefVar の場合．$l' = \sigma'(next)$ とおくと $\sigma'' \uplus \{next \mapsto l' - 1\}$ によって入力の記憶域が一意に得られる．また，$\gamma, x \mapsto l$ から最後の組を取り除くことで γ が一意に得られる．

推論規則 DefArr の場合．$[\![\cdot]\!]$ が関数であることから c の値 v は一意に定まる．後は，推論規則 DefVar の場合と同様の議論により γ と σ が一意に定まる．

以上より，第 2 文も成り立つ．　□

式の評価の判断および左辺式の評価の判断が後方決定的でないにもかかわらず，文の実行の判断は可逆である．

補題 5（文の実行の判断の可逆性）．任意の文 s，プロシージャ環境 Γ，環境 γ，および記憶域 σ，σ'，σ'' に対して $\gamma \vdash^{\Gamma}_{stmt} \langle s, \sigma\rangle \Rightarrow \sigma'$ かつ $\gamma \vdash^{\Gamma}_{stmt} \langle s, \sigma\rangle \Rightarrow \sigma''$ ならば，$\sigma' = \sigma''$ である．また，任意の文 s，プロシージャ環境 Γ，環境 γ，および記憶域 σ，σ'，σ'' に対して $\gamma \vdash^{\Gamma}_{stmt} \langle s, \sigma'\rangle \Rightarrow \sigma$ かつ $\gamma \vdash^{\Gamma}_{stmt} \langle s, \sigma''\rangle \Rightarrow \sigma$ ならば，$\sigma' = \sigma''$ である．

この補題は，文 s と組 (e_1, s_1, s_2, e_2) に関する構造帰納法では，導出の最後の推論規則が Loop1Rec であるときうまく証明できない．結論にある組 (e_1, s_1, s_2, e_2) をともなう Loop1Rec が前提の判断の推論規則 Loop2Rec の前提にも現れているかもしれないので，適切な整礎順序集合を定義できないからである．導出の最後の推論規則が Loop2Rec であるときも同様の問題がある．そこで導出に関する帰納法を使って証明を行う．

注意が必要なのは，繰返しの実行の判断の導出が無条件には一意に定まらないことである．たとえば，$\gamma \vdash^{\Gamma}_{loop1} \langle(\texttt{false}, \texttt{skip}, \texttt{skip}, \texttt{false}), \sigma\rangle \Rightarrow \sigma$ は，任意の回数の Loop1Rec および Loop2Rec が用いられた導出木の根になることができる．しかし，繰返しの実行の判断が，文の実行の判断を結論に持つ推論規則の前提に出現するのは，LoopMain の場合のみである．

LoopMain の前提で現れた場合，繰返しの実行の判断 $\gamma \vdash^{\Gamma}_{loop1} \langle (e_1, s_1, s_2, e_2), \sigma\rangle \Rightarrow \sigma'$ は，$\gamma, \sigma \vdash_{expr} e_1 \not\Rightarrow 0$ および $\gamma, \sigma' \vdash_{expr} e_2 \not\Rightarrow 0$ という判断とともに出現しており，これらの条件の下で LoopMain が導出の最後の推論規則であるような導出木は一意に定まる．このことを以下の証明では用いる．

また，帰納法の前提として前方決定性と後方決定性の両方が同時に仮定される必要があることに注意してほしい．これは，推論規則 Uncall の前方決定性と後方決定性を証明するときにそれぞれ逆方向の決定性を帰納法の仮定とする必要があるからである．

証明. まず，導出の最後の推論規則の一意性を示す．すなわち，任意の文 s，プロシージャ環境 Γ，環境 γ，および記憶域 σ もしくは σ' に対して，文の実行の判断 $\gamma \vdash^{\Gamma}_{stmt} \langle s, \sigma\rangle \Rightarrow \sigma'$ を導出する推論規則が一意に定まることと，任意の式 e_1, e_2，文 s_1, s_2，プロシージャ環境 Γ，環境 γ，および記憶域 σ もしくは σ' に対して，$\gamma, \sigma \vdash_{expr} e_1 \not\Rightarrow 0$ および $\gamma, \sigma' \vdash_{expr} e_2 \not\Rightarrow 0$ であるならば，判断 $\gamma \vdash^{\Gamma}_{loop1} \langle (e_1, s_1, s_2, e_2), \sigma\rangle \Rightarrow \sigma'$ および $\gamma \vdash^{\Gamma}_{loop2} \langle (e_1, s_1, s_2, e_2), \sigma\rangle \Rightarrow \sigma'$ を導出する推論規則が一意に定まることを示す．

推論規則 Ass, LoopMain, Skip, Seq, Call, Uncall は，文の形によって適用できるものが一意に定まる．

文 s が条件文の場合．導出の最後の推論規則が IfTrue もしくは IfFalse である．記憶域 σ が与えられた場合は，$\gamma, \sigma \vdash_{expr} e_1 \not\Rightarrow 0$ もしくは $\gamma, \sigma \vdash_{expr} e_1 \Rightarrow 0$ が排反に成り立つため導出の最後の推論規則は一意に定まる．同様に，記憶域 σ' が与えられた場合は，$\gamma, \sigma \vdash_{expr} e_2 \not\Rightarrow 0$ もしくは $\gamma, \sigma \vdash_{expr} e_2 \Rightarrow 0$ が排反に成り立つため導出の最後の推論規則は一意に定まる．

$\gamma, \sigma \vdash_{expr} e_1 \not\Rightarrow 0$ および $\gamma, \sigma' \vdash_{expr} e_2 \not\Rightarrow 0$ という条件の下，判断 $\gamma \vdash^{\Gamma}_{loop1} \langle (e_1, s_1, s_2, e_2), \sigma\rangle \Rightarrow \sigma'$ が成り立った場合を考える．ここでその導出木の根からたどって最初に出現した文の実行の判断および式の評価の判断を葉と呼ぶことにする．葉を左から右に順番にならべると，ある n が存在して，$0 \le i < n$ に対して，$\sigma_0 = \sigma$，$\sigma_{2n+1} = \sigma'$，

$$\gamma \vdash^{\Gamma}_{stmt} \langle s_1, \sigma_{2i}\rangle \Rightarrow \sigma_{2i+1} \qquad \gamma, \sigma_{2i+1} \vdash_{expr} e_2 \Rightarrow 0$$
$$\gamma \vdash^{\Gamma}_{stmt} \langle s_2, \sigma_{2i+1}\rangle \Rightarrow \sigma_{2i+2} \quad \gamma, \sigma_{2i+2} \vdash_{expr} e_1 \Rightarrow 0$$

および，$\gamma \vdash^{\Gamma}_{stmt} \langle s_1, \sigma_{2n}\rangle \Rightarrow \sigma_{2n+1}$ となる．この葉の列の中では式 e_1 と e_2 の判断がすべて成立しているのに対して，$\gamma, \sigma_0 \vdash_{expr} e_1 \not\Rightarrow 0$ および $\gamma, \sigma_{2n+1} \vdash_{expr} e_2 \not\Rightarrow 0$ であることから n は一意に定まる．一方，このような葉を持つような導出木は Loop1Base, Loop2Base, Loop1Rec, Loop2Rec からしか構成されない．Loop1Base と Loop1Rec および Loop2Base と Loop2Rec では，それぞれ同じ形をした判断が結論に出現している．しかし，Loop1Base と Loop2Base の前提に出現する葉の数は 1 つであり，Loop1Rec と Loop2Rec の前提に出現する葉の数は 5 つ以上であり，それぞれ葉の数が異なる．また，Loop1Rec と Loop2Rec の前提に出現する葉でない判断はそれぞれ 1 つしかない．よって，それぞれの判断に対して使用できる推論規則はつねに 1 つである．したがって，任意の $\gamma \vdash^{\Gamma}_{loop1} \langle (e_1, s_1, s_2, e_2), \sigma\rangle \Rightarrow \sigma'''$ を導出する推論規則は一意に定まる．同様にして判断 $\gamma \vdash^{\Gamma}_{loop2} \langle (e_1, s_1, s_2, e_2), \sigma\rangle \Rightarrow \sigma'''$ を導出する推論規則も一意に定まる．

以上で，導出の最後に使われる推論規則の一意性が示された．以降では断りなく，導出の最後の推論規則の前提にある判断を導出する推論規則が一意であることを使う．

以降では，文の実行の判断 $\gamma \vdash^{\Gamma}_{stmt} \langle s, \sigma\rangle \Rightarrow \sigma'$ および繰返しの実行の判断 $\gamma \vdash_{loop1} \langle (e_1, s_1, s_2, e_2), \sigma\rangle \Rightarrow \sigma'$ および $\gamma \vdash_{loop2} \langle (e_1, s_1, s_2, e_2), \sigma\rangle \Rightarrow \sigma'$ の導出に関する帰納法を使う．ただし，ここで考える繰返しの実行の判断には，LoopMain の前提に出現する繰返しの実行の判断 $\gamma \vdash^{\Gamma}_{loop1} \langle (e_1, s_1, s_2, e_2), \sigma\rangle \Rightarrow \sigma'$ を根とする導出木の中に現れたものという条件を課す．それぞれの場合では，導出の最後の推論規則の前提に出現するすべての文の実行の判断および繰返しの実行の判断が可逆であることを帰納法の仮定として，導出の最後の推論規則の可逆性を示す．また，自明な前方決定性の証明は省略する．

導出の最後の推論規則が Ass の場合．後方決定性を示す．式の評価の判断 $\gamma, \sigma' \vdash_{expr} e \Rightarrow v'$ から v' が求まる．ここで，右辺の値が代入の前後で変化しないという条件 $v = v'$ から v が一意に定まる．また，左辺式の評価の判断 $\gamma, \sigma' \vdash_{lexpr} e_l \Rightarrow l'$ から l' が求まる．ここで，左辺の左辺値が代入の前後で変化しないという条件 $l = l'$ から l が一意に定まる．$[\![\odot]\!]$ は第 1 引数について単射な関数であるので，$\sigma'(l)$ と v から v_1 が一意に定まる．よって，$\sigma = \sigma'' \uplus \{l \mapsto v_1\}$ が一意に定まる．したがって，推論規則 Ass の結論は後方決定的である．

導出の最後の推論規則が IfTrue もしくは IfFalse の場合．帰納法の仮定より判断 $\gamma \vdash^{\Gamma}_{stmt} \langle s_1, \sigma\rangle \Rightarrow \sigma'$ は後方決定的である．また，補題 2 より，2 つの式の評価の判断は一意性を持つ．したがって，推論規則 IfTrue, IfFalse は後方決定的である．

導出の最後の推論規則が Skip の場合．前提がないので基底ケースであり，推論規則は後方決定的である．

導出の最後の推論規則が Loop1Base, Loop2Base の場合．前提は文の実行の判断のみなので帰納法の仮定より推論規則は可逆である．

導出の最後の推論規則が LoopMain, Loop1Rec, Loop2Rec, Seq の場合．補題 2 と帰納法の仮定より推論規則が可逆であることを示せる．ただし，導出の最後の推論規則が Loop1Rec もしくは Loop2Rec の場合，文

の実行の判断および繰返しの実行の判断の両方の仮定を使う．ここで考えている導出木は，上で議論したように一意に定まっていることに注意してほしい．

導出の最後の推論規則が Call の場合．プロシージャ環境 Γ が関数であること，左辺式の評価が一意であること（補題 3）および帰納法の仮定により，この規則は後方決定的である．

導出の最後の推論規則が Uncall の場合．帰納法の仮定より推論規則が可逆であることを示せる．ただし，前方決定性の証明に前提の後方決定性の仮定を使い，後方決定性の証明に前提の前方決定性の仮定を使う．　□

可逆な判断は，対応する入力が存在する場合には，出力から入力を一意に求めることができる．逆方向の実行を行うときも順方向で用いた推論規則を用いることができる．推論規則の中には前提に後方決定的ではない式の評価の判断や左辺式の評価の判断が出てくるが，これらは逆方向に推論規則を用いる場合においても順方向に用いられるので問題はない．注意が必要なのは，推論規則 Ass の中で，σ' から σ を求めるところである．ここで $v_2 = [\![\odot]\!](v_1, v)$ とおくことにする．関数 $[\![\odot]\!]$ が第 1 引数について単射であることを利用すると，v_2 と v から v_1 が一意に求まる．

文の実行だけでなく，変数宣言なども含めてプログラム全体が可逆であることを示す．

定理 6（プログラミング言語の可逆性）．任意の Janus のプログラム $prog$，環境 $\gamma, \gamma', \gamma''$，および記憶域 $\sigma, \sigma', \sigma''$ に対して $\vdash^{\Gamma}_{prog} \langle prog, (\gamma, \sigma)\rangle \Rightarrow \gamma', \sigma'$ かつ $\vdash^{\Gamma}_{prog} \langle prog, (\gamma, \sigma)\rangle \Rightarrow \gamma'', \sigma''$ ならば，$\gamma' = \gamma''$ かつ $\sigma' = \sigma''$ である．また，任意の Janus のプログラム $prog$，環境 γ，γ'，γ''，記憶域 σ，σ'，および σ'' に対して $\vdash^{\Gamma}_{prog} \langle prog, (\gamma', \sigma')\rangle \Rightarrow \gamma, \sigma$ かつ $\vdash^{\Gamma}_{prog} \langle prog, (\gamma'', \sigma'')\rangle \Rightarrow \gamma, \sigma$ ならば，$\gamma' = \gamma''$ かつ $\sigma' = \sigma''$ である．

証明． 推論規則 Main の前提は，変数と配列変数の初期化の判断と文の実行の判断からなる．前者は補題 4 より，後者は補題 5 より可逆である．よって，推論規則 Main は可逆である．　□

すでに `uncall` によってプロシージャの逆実行ができることをみたが，プログラム逆変換をしてその結果を順実行しても同じ振舞いが得られる．次の定理は逆文の存在を示している．

定理 7（逆文の存在）．任意のプロシージャ環境 Γ，文 s に対して，ある文 s' が存在して，任意の環境 γ，γ'，および記憶域 σ，σ' に対して，$\vdash^{\Gamma}_{stmt} \langle s, (\gamma, \sigma)\rangle \Rightarrow \gamma', \sigma'$ ならば，$\vdash^{\Gamma}_{stmt} \langle s', (\gamma', \sigma')\rangle \Rightarrow \gamma, \sigma$ である．

文献 [42] で定義された文の逆変換器と同様なものが定義できる．ただし，本稿で拡張した引数付きプロシージャ呼び出しと逆呼び出しの引数は変更する必要がある：

$$\mathcal{I}[\![\texttt{call } q(e_{l1}, \ldots, e_{ln})]\!] = \texttt{uncall } q(e_{l1}, \ldots, e_{ln})$$
$$\mathcal{I}[\![\texttt{uncall } q(e_{l1}, \ldots, e_{ln})]\!] = \texttt{call } q(e_{l1}, \ldots, e_{ln})$$

この逆変換器 $\mathcal{I}$ によって，上記の s から s' が一意に求まる：

$$s' = \mathcal{I}[\![s]\!] \tag{1}$$

証明のスケッチ． 文の実行の判断と繰返しの実行の判断が導出の最後の推論規則となる導出木を考えて，導出木に関する帰納法を行う．　□

とくに，本稿では，プロシージャの実引数は左辺値を持つ式であるが，同一の実引数を持つ同一のプロシージャ呼び出しと逆呼び出しは逆になる．

系 8. 任意のプロシージャ環境 Γ，プロシージャ名 q，左辺式 $e_{l1}, \ldots, e_{ln}$（n はプロシージャ q の引数の数），環境 γ，γ'，および記憶域 σ，σ' に対して，

$$\vdash^{\Gamma}_{stmt} \langle \texttt{call } q(e_{l1}, \ldots, e_{ln}), (\gamma, \sigma)\rangle \Rightarrow \gamma', \sigma'$$

ならば，またそのときに限り

$$\vdash^{\Gamma}_{stmt} \langle \texttt{uncall } q(e_{l1}, \ldots, e_{ln}), (\gamma', \sigma')\rangle \Rightarrow \gamma, \sigma$$

である．

式 (1) より

$$\texttt{uncall } q(e_{l1}, \ldots, e_{ln}) = \mathcal{I}[\![\texttt{call } q(e_{l1}, \ldots, e_{ln})]\!] \tag{2}$$

であるので，この系は成り立つ．

この系より実用上に役立つ示唆は，プロシージャ呼び出しの直後にその同一の実引数をともなうプロシージャ逆呼び出しがされたら，環境と記憶域はこれらの呼び出しの前後で変化しないということである．

4. 代入の最適化

代入では，左辺において代入される変数や配列変数が右辺に現れると可逆性を持つ意味を自然に定義することが難しい．たとえば，`x -= x` という文は直前の `x` の値が何であってもこの文の実行後には値が 0 になってしまうように見える．文献 [39], [42] では，この問題を構文を制限することで回避していた．変数への代入文では，左辺の変数が右辺に現れることを禁止していた．また，配列変数への代入では，配列変数が左辺の添字式および右辺に現れることを禁止していた．

環境・記憶域モデルを採用した本稿の Janus でもこのような制限を課すことで代入文に可逆性を持たせることができる．右辺式の評価では，記憶域において左辺の変数に対応する値域を未定義にすればよい．ただし，環境の定義域から左辺の変数を除くだけでは不十分である．左辺の参照先のオブジェクトと右辺に現れた参照先のオブジェクト

が一致してしまうことがありうるからである．

本稿では，代入の前後で，右辺式の値と左辺式の左辺値が変化しないという条件が課されている．一方，この条件の判定は，それぞれの式の評価を代入の実行の前後に 2 回ずつ行わなければならないという点で効率が悪い．代入においてエイリアスが出現しない場合は，コンパイラによる最適化ができる．

ここで, 式 e において左辺式 e_l のエイリアスが出現することを形式的に表しておく．環境 γ および記憶域 σ の下で, 左辺式 e_l が左辺式 e'_l のエイリアスであるとは，$\gamma, \sigma \vdash e_l \Rightarrow l$ かつ $\gamma, \sigma \vdash e'_l \Rightarrow l$ であることをいう．集合 $Lexp(e)$ は式 e の中において左辺式でもある部分式をすべて集めたものとする．環境 γ および記憶域 σ の下で，式 e の中に左辺式 e_l のエイリアスが出現するとは，$\gamma, \sigma \vdash e_l \Rightarrow l$ であり，ある $e'_l \in Lexp(e)$ に対して $\gamma, \sigma \vdash e'_l \Rightarrow l$ となるものがあることをいう．

たとえば，代入が行われる変数のエイリアスが右辺に出現しない場合に用いることができる変数への代入の推論規則は

$$\frac{\begin{array}{c}\forall e'_l \in Lexp(e).\ \gamma, \sigma \vdash e'_l \not\Rightarrow l \\ \gamma \vdash_{lexpr} x \Rightarrow l \\ \sigma = \sigma'' \uplus \{l \mapsto v_1\} \\ \sigma' = \sigma'' \uplus \{l \mapsto [\![\odot]\!](v_1, v)\} \\ \gamma, \sigma'' \vdash_{expr} e \Rightarrow v\end{array}}{\gamma \vdash^{\Gamma}_{stmt} \langle x \odot\!\!= e, \sigma\rangle \Rightarrow \sigma'}\ \textsc{AssVar2}$$

と表すことができる．推論規則の前提における一番上の判断 $\forall e'_l \in Lexp(e).\ \gamma, \sigma \vdash e'_l \not\Rightarrow l$ が静的に判定できる場合，すなわち動的にはこの判断を検査しなくてよい場合がある．この場合，この推論規則には，前提に，式 e の評価が 1 回しか出現しておらず，推論規則 Ass よりも効率的である．記憶域 σ' から σ は次のように一意に求められる．まず，$\gamma \vdash_{lexpr} x \Rightarrow l$ で l が一意に定まる．次に，$\sigma' = \sigma'' \uplus \{l \mapsto v'\}$ で σ'' が，$\gamma, \sigma'' \vdash_{expr} e \Rightarrow v$ から v が，さらに $[\![\odot]\!]$ が第 1 引数について単射な関数であることから v_1 がそれぞれ一意に求まる．最後に $\sigma = \sigma'' \uplus \{l \mapsto v_1\}$ により σ が一意に求まる．

また，配列変数への代入において左辺式のエイリアスが右辺にも左辺の配列の添字にも現れない場合に用いることができる添字付き配列変数への代入の推論規則は

$$\frac{\begin{array}{c}\forall e'_l \in Lexp(e) \cup Lexp(e_l).\ \gamma, \sigma \vdash e'_l \not\Rightarrow l + v_l \\ \gamma \vdash x \Rightarrow l \qquad \sigma = \sigma'' \uplus \{l + v_l \mapsto v_1\} \\ \sigma' = \sigma'' \uplus \{l + v_l \mapsto [\![\odot]\!](v_1, v)\} \\ \gamma, \sigma'' \vdash_{expr} e_l \Rightarrow v_l \qquad \gamma, \sigma'' \vdash_{expr} e \Rightarrow v\end{array}}{\gamma \vdash^{\Gamma}_{stmt} \langle x[e_l] \odot\!\!= e, \sigma\rangle \Rightarrow \sigma'}\ \textsc{AssArr2}$$

と表すことができる．前提の 1 番上の判断 $\forall e'_l \in Lexp(e) \cup Lexp(e_l).\ \gamma, \sigma \vdash e'_l \not\Rightarrow l + v_l$ が静的に判定できれば，前提に式 e の評価と左辺式 e_l の評価がそれぞれ 1 回しかでてきていないので推論規則 Ass よりも効率的である．この推論規則の前提では，σ'' と v_l は相互に依存してしまっている．すなわち，σ から σ' を求めるときも，σ' から σ を求めるときも，σ'' を求めるには v_l が必要で v_l を求めるには σ'' が必要である．しかし，以下の補題から式 e の任意の部分式が記憶場所 l を左辺値として持たなければ，記憶域から l を除いても式の評価に影響がないことがいえる．

補題 9. 任意の $e_l \in Lexp(e)$ に対して $\gamma, \sigma \vdash_{lexpr} e_l \not\Rightarrow l$ であるとする．ある v が存在して，$\gamma, \sigma \vdash_{expr} e \Rightarrow v$ ならば，$\sigma = \sigma' \uplus \{l \mapsto v'\}$ となる σ' に対して，$\gamma, \sigma' \vdash_{expr} e \Rightarrow v$ である．

証明. 式 e の上の構造帰納法による． □

この補題から AssArr2 は可逆であることは明らかである．順実行のときは $\gamma, \sigma'' \vdash_{expr} e_l \Rightarrow v_l$ の σ'' の代わりに σ を，逆実行のときは σ' を使用して v_l を求めることができる．これにより σ'' と v_l の相互に依存する問題は解決できた．

ほかにも e_l のみもしくは e のみ 2 回評価する推論規則も同様に定めることができる．こうした推論規則は，AssVar2 や AssArr2 と同様に，適用できたらその入出力は Ass に一致する．

5. 関連研究

可逆論理回路 [6], [36] では，任意の演算が可逆に実行される．可逆論理回路は，その内部で情報が消されることが原因による電力消費がない [4], [18] ことなどから，低消費電力設計に応用される可能性がある．量子論理回路における任意の演算は可逆であるので，可逆論理回路の分野の成果は量子論理回路に適用することができる．量子計算機は，素因数分解などの実用的な問題が伝統的な回路よりも高速に解けることが知られている [29]．ロータリー素子と呼ばれる可逆論理素子は，自身は単純な構造であるが，万能な可逆コンピュータを簡潔に構成できることが知られている [24]．

こうした計算デバイスにおける計算をうまく記述するために適したプログラミング言語を開発するのは重要な研究課題である．本稿では，高水準プログラミング言語において，可逆性を自然にあつかえる再利用可能な部品，すなわち引数付きプロシージャを考えた．

可逆性を持つプログラミング言語の研究は古くからある．約半世紀前の文献 [27] においてすでに，アセンブリ言語の命令の実行順序を逆にしたりそれぞれの命令を対応する命令に置き換えて実行したりすることで，逆実行を行うことが考えられていた．彼らの考えた MOHAC（Mohawk-Hudson automatic computer）という仮想計算機において，たとえば，メモリの値をアキュムレータに足す `ADD` という命令の共役（conjugate）はメモリの値をアキュムレータから引く `SUB` という命令であると定義されている．命令の順序を逆にされたサブルーチンは転置（transposed），命令が共

役に置き換えられたサブルーチンは共役（conjugate），あるサブルーチンの行った仕事を元に戻すサブルーチンは逆（inverse）と呼ばれる．さらに，共役転置が逆になるサブルーチンはユニタリ（unitary）と呼ばれる．ユニタリなサブルーチンは本稿でのべたような可逆性を持つ．マルチプログラミング環境において，ユニタリサブルーチンを使うメリットには，サブルーチンとその逆サブルーチンを両方とも使うよりも，1965 年においては今よりずっと貴重な資源であったメモリの使用効率が良いことがあげられている．

局所的に逆変換ができるプリミティブや制御流演算子から構成されるプログラミング言語は，実行環境の可逆的な構造を利用することを支援する．このような局所的に逆化可能なプログラミング言語には，高水準言語では，Janus [19], R 言語 [9], SRL [20], ESRL [20], Inv [25], Type Isomorphisms に基づく Π や Π^o などの言語 [16] が，アセンブリ言語などの低水準言語では，PISA [9]，BobISA [33], Hall の命令セットアーキテクチャ [12]，プログラム逆変換において中間表現を行うためのスタック言語 [10] がある．Gries の逆化可能言語 [11] は，注釈も言語の一部として解釈実行されるものとするとこの分類に含まれる．我々が知る範囲では，Janus は最も古い高水準可逆命令型プログラミング言語である．PISA の `EMIT`，および Hall の命令セットアーキテクチャの `ERASE` はそれぞれビットを消去するが，これらの命令を除くと，ここであげたすべての言語は実行中にデータを消去したり制御情報を上書きしたりすることはない．また，ランタイムシステムによって実行時間に比例するようなゴミ情報を暗黙的に管理されることもない．可逆回路を設計するための言語でも局所的な逆変換が支援されており，コンビネータ言語 [31], [32] や命令型言語 SyReC [36], [38] が知られている．

引数渡しの方法には，値渡しや参照渡しが広く使われている．たとえば，値渡しは，Algol-60，Pascal，および C 言語などで用いられており，参照渡しは，Fortran や Pascal などで用いられている．値渡しでは，プロシージャ呼び出しなどから制御が戻ったときに仮引数に保持されている値へアクセスできなくなる．同様に，参照渡しでも，プロシージャ呼び出しなどから制御が戻ったときに仮引数に保持されている参照が計算できなくなる．たとえば，ある変数 `i` を添字としてともなった配列変数 `a[i]` が参照として渡されたプロシージャの中で，変数 `i` の値が変化したときにこの問題が起こる．したがって，一般には，その値や参照を復元することができないので，制限なしに値渡しや参照渡しを導入するとプロシージャ呼び出しは可逆ではなくなる．

引数渡し機構を持つ可逆プログラミング言語には，Ψ-lisp, R 言語，Janus，RFUN，SyReC がある．

Ψ-lisp [3] は，逆実行可能な Lisp に似た言語である．条件式においてランタイムシステムが計算履歴を記憶することを許していることから，Ψ-lisp はランタイムシステムのクリーンさを保証しておらず，局所的に逆化可能な言語ではない．Ψ-lisp では，値渡しや参照渡しではなく，スワップ渡し（pass by swapping）[13] が用いられている．スワップ渡しでは，仮引数は対応する実引数とスワップされる．スワップ渡しでは，関数の本体に現れる変数を実引数として渡すことや同一の記憶場所を指す構文対象を関数呼び出しの実引数として 2 つ以上渡すことができない．Ψ-lisp では，変数の出現は線形的である—すなわちすべての変数がちょうど 1 回アクセスされることが保証されている—ので，これらの違反はおきることがない．

R 言語 [9] は，引数付きの再帰サブルーチンを備えた高水準言語であり，Pendulum アーキテクチャ [34], [35] 上で動作するクリーンな可逆プログラムを効率的に開発するために設計された．R コンパイラ RCOMP [9] は，R 言語をマクロ展開のようなコード変換によって PISA（Pendulum instruction set architecture）アセンブリ言語に翻訳する．PISA アセンブリ言語が可逆であることによって R 言語は可逆であることが保証されている．しかし，制約を満たさない R プログラムは，可逆ではあるかもしれないが，正しく動作することが保証されておらず，また R コンパイラの翻訳が適切に意味を保持しているかも保証されていない．R 言語では，複数引数付きのサブルーチン呼び出しができる．実引数が変数やメモリ参照である場合，対応する仮引数の値を変えることができる．また，実引数が定数や式である場合，サブルーチン呼び出しから戻るときの対応する仮引数の値は元の実引数の値でなければその後の動作は保証されない．R 言語では，実引数が変数の場合は参照渡し，式の場合は値渡しが行われていると見受けられる．しかし，このことは形式的意味論が与えられていないので明確ではない．

文献 [39] では，Janus に引数付きプロシージャを導入しており，実引数を変数に制限して参照渡しを実現している．エイリアスは，大域変数を実引数にしたり，同じ参照を複数引数にしたりすることが禁止されているので，現れないことが保証されている．プロシージャ呼び出しの意味規則では，呼び出されたプロシージャ本体の仮引数が対応する実引数に書き換えられているという意味では，名前渡しの実現方法に類似している．

RFUN [2], [41] は可逆な関数型言語であり，引数をともなった関数が用いられている．RFUN では，実引数は必ず関数の本体内で使われ，また関数呼び出しの返り値から実引数が必ず復元できるように設計されている．

文献 [42] の Janus をもとにして設計された可逆回路の合成のためのプログラミング言語に SyReC がある [36], [38]．文献 [38] では，既存の方法に比べて，複雑な可逆回路を効率的に合成することができたと報告されている．最近の拡張によって SyReC はモードつきの引数が渡せるように

なった [28], [37]．モードには，仮引数を，入力，出力，および入出力にそれぞれ用いることや局所モジュールのみで用いること，順序回路における状態を表すことを指定することができる．

双方向変換に用いるプログラミング言語も，前方と後方の意味を持っており，ビュー更新問題の解決に用いられている [8], [15]．しかし，こうした言語は必ずしも可逆やクリーンである必要がない．

可逆プログラミング言語に関連する研究として，逆計算やプログラム逆変換も広く研究されている（たとえば，文献 [1], [7], [11], [14], [22]）．プログラム逆変換における問題の 1 つに局所的な後方非決定性がある．とくに末尾再帰プログラムでは，後方決定性が単純な条件で記述できないことが多いという点で困難な問題であることが知られている．文献 [10] のプログラム逆変換は一部の末尾再帰プログラムにも適用可能である．この方法は，スタック言語の命令を終端記号，関数呼び出しを非終端記号と見なした文脈自由文法で表現されたプログラムに LR 法の構文解析器の手法を適用して，右再帰を行う文法に対応する決定性を持つ逆プログラムを生成するものである．文法の種類によって問題の難しさは異なるが，文法に基づくプログラム逆変換の枠組みは文献 [21] において整理されている．文献 [23] では，末尾再帰のプログラムを繰返しに変換し，そのプログラムの呼び出しに用いられている実引数の情報を用いて繰返しの停止を判定している．このアイデアをもとに，項書換系を用いて形式化されたプログラム逆変換法は，既存の方法と比較してより多くの末尾再帰プログラムをプログラム逆変換することができたとの報告がある [26]．

6. おわりに

本稿では，大域変数がある可逆プログラミング言語 Janus のプロシージャの実引数に，大域変数や添字付き配列変数の参照や同一の参照を持つ変数や添字付き配列変数を渡せるようにした．この結果，プロシージャ呼び出しを行うプログラムをより簡潔に記述できるようになった．拡張によって導入されたエイリアスは，元々の Janus を非可逆にしてしまう．可逆性を保証するために，我々は，状態モデルを環境・記憶域モデルに拡張して，環境と記憶域の更新に制約を設けた．代入やプロシージャ呼び出しと逆呼び出しの推論規則に課された重要な制約は，代入の左辺の左辺値やプロシージャ呼び出しの引数の左辺値がその文の実行前後で変化しないというものである．また，記憶域を導入したことで，代入時において動的に大域プログラム解析を行うことなく，参照が同一のオブジェクトを指すかを判定することができるようになった．このことにともなって，代入は，左辺の左辺値が出現しない場合は，その推論規則をより簡単なもので実現できた．さらに，環境・記憶域モデルを用いることで同一の記憶場所を複数の識別子で表すことができるようになったので，文献 [39] で用いられていた置換は本稿では不要になった．

我々は，拡張した Janus が可逆であることを形式的に示した．とくに，文献 [39], [42] では示されていなかった変数と配列変数の初期化やプログラムの実行の判断が可逆であることについても示した．また，文献 [39], [42] では，スペースの制限から証明の概略しか記述されていなかったが，本稿では後方決定性に関して完全な証明を与えた．

ほとんどの推論規則は可逆であるという点で，本稿の Janus を可逆言語でクリーンに実装することは容易であると考えられる．記憶域の更新は非単射であるが，それぞれの推論規則の中では，更新後の記憶域から更新前の記憶域を一意に再構築できた．また，意味関数 $[\![\odot]\!]$ は第 1 引数について単射であり，出力と第 2 引数から第 1 引数を一意に再構築できた．したがって，意味規則において非可逆なのは式の評価の判断および左辺式の評価の判断の推論規則だけである．

本稿では，4 章において，代入を部分的に効率化することには成功した．しかし，提案する推論規則を可逆なランタイムシステムにおいてより効率良く実現することは今後の課題である．たとえば，仮引数によって示される記憶場所の書き換えがプロシージャの本体で行われないことが保証できれば，Ada の `in` モードの仮引数や初期の Fortran の定数渡し（pass as constant）の実現のように値のコピーで引数を渡した方が実行効率が良くなることが期待される．

謝辞　南山大学情報理工学部ソフトウェア工学科卒業研究発表会審査員，横山研究室各位，および匿名の査読者には，有益なコメントを多数いただいた．本研究の一部は，JSPS 科研費 25730049 の助成を受けたものである．記して感謝の意を表する．本研究は，情報処理学会第 76 回全国大会で発表した内容の一部を拡張したものを含む．

参考文献

[1] Abramov, S.M. and Glück, R.: The universal resolving algorithm: Inverse computation in a functional language, *Proc. Mathematics of Program Construction* (*MPC 2000*), Backhouse, R. and Oliveira, J.N. (Eds.), Lecture Notes in Computer Science, Vol.1837, pp.187–212, Springer-Verlag, DOI: 10.1007/10722010_13 (2000).

[2] Axelsen, H. and Glück, R.: Reversible Representation and Manipulation of Constructor Terms in the Heap, *Proc. Reversible Computation* (*RC 2013*), Dueck, G.W. and Miller, D.M. (Eds.), Lecture Notes in Computer Science, Vol.7948, pp.96–109, Springer-Verlag, DOI: 10.1007/978-3-642-38986-3_9 (2013).

[3] Baker, H.G.: NREVERSAL of Fortune — The Thermodynamics of Garbage Collection, *Proc. International Workshop on Memory Management* (*IWMM 1992*), Lecture Notes in Computer Science, Vol.637, pp.507–524, Springer-Verlag, DOI: 10.1007/BFb0017210 (1992).

[4] Bennett, C.H.: Logical Reversibility of Computation, *IBM Journal of Research and Development*, Vol.17, No.6, pp.525–532, DOI: 10.1147/rd.176.0525 (1973).

[5] Cooper, K.D.: Analyzing Aliases of Reference Formal Parameters, *Proc. Symposium on Principles of Programming Languages* (*POPL 1985*), pp.281–290, ACM Press, DOI: 10.1145/318593.318658 (1985).

[6] De Vos, A.: *Reversible Computing: Fundamentals, Quantum Computing, and Applications*, Wiley-VCH (2010).

[7] Dijkstra, E.W.: Program inversion, *Program Construction: International Summer School*, Bauer, F.L. and Broy, M. (Eds.), Lecture Notes in Computer Science, Vol.69, pp.54–57, Springer-Verlag, DOI: 10.1007/BFb0014657 (1978).

[8] Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C. and Schmitt, A.: Combinators for Bi-Directional Tree Transformations: A Linguistic Approach to the View Update Problem, *ACM Trans. Prog. Lang. Syst.*, Vol.29, No.3, pp.1–65, DOI: 10.1145/1232420.1232424 (2007).

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

[10] Glück, R. and Kawabe, M.: A Method for Automatic Program Inversion Based on LR(0) Parsing, *Fundamenta Informaticae*, Vol.66, pp.367–395 (2005).

[11] Gries, D.: *The Science of Programming*, Texts and Monographs in Computer Science, chapter 21 Inverting Programs, pp.265–274, Springer-Verlag (1981).

[12] Hall, J.S.: A Reversible Instruction Set Architecture and Algorithms, *Proc. Workshop on Physics and Computation* (*PhysComp 1994*), pp.128–134, IEEE Press, DOI: 10.1109/PHYCMP.1994.363690 (1994).

[13] Harms, D.E. and Weide, B.W.: Copying and Swapping: Influences on the Design of Reusable Software Components, *IEEE Trans. Softw. Eng.*, Vol.17, No.5, pp.424–435, DOI: 10.1109/32.90445 (1991).

[14] Hou, C., Vulov, G., Quinlan, D., Jefferson, D., Fujimoto, R. and Vuduc, R.: A New Method for Program Inversion, *Proc. Compiler Construction* (*CC 2012*), O'Boyle, M. (Ed.), Lecture Notes in Computer Science, Vol.7210, pp.81–100, Springer-Verlag, DOI: 10.1007/978-3-642-28652-0_5 (2012).

[15] Hu, Z., Mu, S.-C. and Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations, *Higher-Order and Symbolic Computation*, Vol.21, No.1-2, pp.89–118, DOI: 10.1007/s10990-008-9025-5 (2008).

[16] James, R.P. and Sabry, A.: Information effects, *Proc. Symposium on Principles of Programming Languages*, (*POPL 2012*), pp.73–84, ACM Press, DOI: 10.1145/2103656.2103667 (2012).

[17] Jones, N.D.: *Computability and Complexity: From a Programming Perspective*, MIT Press (1997). Revised version, available from ⟨http://www.diku.dk/~neil/Comp2book.html⟩.

[18] Landauer, R.: Irreversibility and Heat Generation in the Computing Process, *IBM Journal of Research and Development*, Vol.5, No.3, pp.183–191, DOI: 10.1147/rd.53.0183 (1961).

[19] Lutz, C.: Janus: A time-reversible language (1986). Letter to R. Landauer.

[20] Matos, A.B.: Linear programs in a simple reversible language, *Theoretical Computer Science*, Vol.290, No.3, pp.2063–2074, DOI: 10.1016/S0304-3975(02)00486-3 (2003).

[21] Matsuda, K., Mu, S.-C., Hu, Z. and Takeichi, M.: A Grammar-based Approach to Invertible Programs, *Proc. European Conference on Programming Languages and Systems* (*ESOP 2010*), Springer-Verlag, DOI: 10.1007/978-3-642-11957-6_24 (2010).

[22] Matsuda, K. and Wang, M.: FliPpr: A Prettier Invertible Printing System, *Proc. Programming Languages and Systems* (*ESOP 2013*), pp.101–120, Springer-Verlag, DOI: 10.1007/978-3-642-37036-6_6 (2013).

[23] Mogensen, T.A.: Report on an Implementation of a Semi-inverter, *Proc. International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics*, Springer-Verlag, DOI: 10.1007/978-3-540-70881-0_28 (2007).

[24] Morita, K.: A Simple Universal Logic Element and Cellular Automata for Reversible Computing, *Proc. Machines, Computations, and Universality* (*MCU 2001*), Margenstern, M. and Rogozhin, Y., (Eds.), Lecture Notes in Computer Science, Vol.2055, pp.102–113, Springer-Verlag, DOI: 10.1007/3-540-45132-3_6 (2001).

[25] Mu, S.-C., Hu, Z. and Takeichi, M.: An Injective Language for Reversible Computation, *Proc. Mathematics of Program Construction* (*MPC 2004*), Kozen, D. (Ed.), Lecture Notes in Computer Science, Vol.3125, pp.289–313, Springer-Verlag, DOI: 10.1007/978-3-540-27764-4_16 (2004).

[26] Nishida, N. and Vidal, G.: Program Inversion for Tail Recursive Functions, *Proc. International Conference on Rewriting Techniques and Applications* (*RTA 2011*), Schmidt-Schauß, M. (Ed.), Leibniz International Proceedings in Informatics (LIPIcs), Vol.10, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, DOI: 10.4230/LIPIcs.RTA.2011.283 (2011).

[27] Reilly, E.D. and Federighi, F.D.: On reversible subroutines and computers that run backwards, *Comm. ACM*, Vol.8, No.9, pp.557–558, 578, DOI: 10.1145/365559.365593 (1965).

[28] RevLib: File Format Documentation (2010). available from ⟨http://www.informatik.uni-bremen.de/rev_lib/doc/docu/revlib_2_0.pdf⟩.

[29] Shor, P.: Algorithms for Quantum Computation: Discrete Logarithms and Factoring, *Foundations of Computer Science*, pp.124–134, DOI: 10.1109/SFCS.1994.365700 (1994).

[30] Strachey, C.: Fundamental Concepts in Programming Languages, *Higher-Order and Symbolic Computation*, Vol.13, No.1-2, pp.11–49, DOI: 10.1023/A:1010000313106 (2000).

[31] Thomsen, M.K.: Describing and Optimising Reversible Logic Using a Functional Language, *Proc. Implementation and Application of Functional Languages* (*IFL 2012*), Gill, A. and Hage, J. (Eds.), Lecture Notes in Computer Science, Vol.7257, pp.148–163, Springer-Verlag, DOI: 10.1007/978-3-642-34407-7_10 (2012).

[32] Thomsen, M.K.: A functional language for describing reversible logic, *Proc. Specification and Design Languages*, pp.135–142 (2012).

[33] Thomsen, M.K., Axelsen, H.B. and Glück, R.: A Reversible Processor Architecture and Its Reversible Logic Design, *Proc. Reversible Computation* (*RC 2012*), De Vos, A. and Wille, R. (Eds.), Vol.7165, pp.30–42, Springer-Verlag, DOI: 10.1007/978-3-642-29517-1_3 (2012).

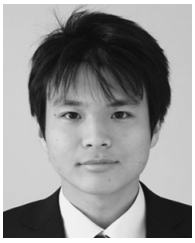
[34] Vieri, C.J.: Pendulum: A Reversible Computer Architecture, Master's thesis, Massachusetts Institute of Technology (1995).

[35] Vieri, C.J.: Reversible computer engineering and archi-

tecture, Ph.D. Thesis, Massachusetts Institute of Technology (1999).
[36] Wille, R. and Drechsler, R.: *Towards a Design Flow for Reversible Logic*, Springer-Verlag (2010).
[37] Wille, R. and Drechsler, R.: The SyReC hardware description language: Enabling scalable synthesis of reversible circuits, *Proc. International Midwest Symposium on Circuits and Systems* (*MWSCAS 2013*), DOI: 10.1109/MWSCAS.2013.6674836 (2013).
[38] Wille, R., Offermann, S. and Drechsler, R.: SyReC: A programming language for synthesis of reversible circuits, *Proc. Forum on Specification and Design Languages* (*FDL 2010*), pp.184–189, DOI: 10.1049/ic.2010.0150 (2010).
[39] Yokoyama, T., Axelsen, H.B. and Glück, R.: Principles of a Reversible Programming Language, *Proc. Computing Frontiers* (*CF 2008*), pp.43–54, ACM Press (2008).
[40] Yokoyama, T., Axelsen, H.B. and Glück, R.: Reversible Flowchart Languages and the Structured Reversible Program Theorem, *Proc. International Colloquium on Automata, Languages and Programming* (*ICALP 2008*), Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A. and Walukiewicz, I. (Eds.), Lecture Notes in Computer Science, Vol.5126, pp.258–270, DOI: 10.1007/978-3-540-70583-3_22 (2008).
[41] Yokoyama, T., Axelsen, H.B. and Glück, R.: Towards a Reversible Functional Language, *Proc. Reversible Computation* (*RC 2012*), De Vos, A. and Wille, R. (Eds.), Lecture Notes in Computer Science, Vol.7165, pp.14–29, Springer-Verlag, DOI: 10.1007/978-3-642-29517-1_2 (2012).
[42] Yokoyama, T. and Glück, R.: A Reversible Programming Language and its Invertible Self-Interpreter, *Proc. Partial Evaluation and Semantics-Based Program Manipulation* (*PEPM 2007*), pp.144–153, ACM Press, DOI: 10.1145/1244381.1244404 (2007).

新海 由侑

1992 年生．2014 年南山大学情報理工学部卒業．



田中 秀明　（正会員）

1992 年生．2014 年南山大学情報理工学部卒業．

横山 哲郎　（正会員）

2006 年東京大学大学院情報理工学系研究科博士課程修了．博士（情報理工学）．2009 年南山大学情報理工学部講師．2011 年同大学同学部准教授．2014 年同大学理工学部准教授．日本ソフトウェア科学会，ACM 各会員．