構造化可逆言語から非構造化可逆言語への翻訳系の試作

2017SE050 宮本 昌武　2017SE051 水野 幹大　2017SE058 野端 祐人
指導教員：横山哲郎

1　はじめに

従来のプログラミング言語の計算は不可逆的に実行される．可逆とは，計算過程において，常に直前と直後の状態が一意に定まり，情報が消失しないことをいう．すなわち，不可逆的に実行されるということは．計算は前方には決定的だが，後方には決定的ではない．逆に，そのプログラムの実行過程が必ず可逆になるような言語設計がなされているプログラミング言語を可逆プログラミング言語という．可逆プログラミング言語には計算機の消費電力を削減できるという利点がある．どういうことかというと，ランダウアーの原理に従うと情報が消失する時，エネルギーを消費すると結論付けられている．よって，可逆の特性を利用することにより消費エネルギーをいくらでも小さくすることが可能である．

プログラミング言語は，高水準言語と低水準言語の大きく２つに分類することができる．高水準言語は，人間の言葉に近いプログラム方法で記述されているので，人間にとって直観的に分かりやすく，抽象的に記述されている．一方，低水準言語は，機械語または，機械語に近い言語であり，機械にとって分かりやすく記述されている．しかし，高水準言語とは反対に，詳細に書かれているので，人間にとっては複雑で理解しにくい言語である．高水準言語から，低水準言語に翻訳する基準として，正確性と効率性の２点が挙げられる．正確性とは，翻訳が意味論を維持する必要があることを意味し，効率性は，翻訳が複雑性を維持することを意味し，ソースプログラムと，ターゲットプログラムは，リソースの使用量に関して同じ漸近する複雑さを持っている必要がある．[1] このような翻訳が可能であることは，可逆言語の有用性にとって非常に重要である．

本研究では，可逆プログラミング言語に関する翻訳器を，純粋な関数型プログラミング言語である Haskell を用いて実装する．手順として，SRL と RL の構文を BNF で書く，すなわち， 字句解析器・構文解析器・プリティプリンタをプログラムする，次に，双方の表示的意味論，操作的意味論や，翻訳規則をプログラムし，翻訳器を実装する．更に， SRL/RL の評価規則，逆変換規則をプログラムし，インタープリタや逆変換規則を実装する．最後に、[2]，[3] で行われた SRL/RL の拡張を行う．

2　関連研究

2.1　プログラム意味論 [4]

言語学には，構文論，意味論のお互いを補い合う言語が存在する．構文論とは，アルファベット，ひらがななどの要素を並べて文章が完成するかのどうかの仕組みのことをいい，意味論は，構文論から成る文字の列について意味を与えるものをいう．プログラム言語では，この構文論及び意味論を展開していく．これから紹介する構文領域，構文規則，意味領域，意味関数の順に定義する方法は，表示的意味論という．これは，コンピュータサイエンスの中では最も数学的に解釈できる理論の一つであり，様々なプログラミング言語に対して有効である．

2.1.1　構文領域

まず，プログラミング言語を定義するには，構文規則（文法）というものを決めなければならないが，その前に，構文領域を導入する必要がある．構文領域とは，その記号が意味する範囲全体を表すものである．

例として，簡単なプログラミング言語 petit の構文領域を図 1 に示す．

ξ : Var　プログラミング（の中で用いられる）変数の記号全体
ϵ : Exp　式（expression) の全体
τ : Pro　プログラムの全体

図 1　petit の構文領域 [4]

2.1.2　構文規則

構文領域を導入した後，その構文領域の記号が示す構文の定義を定める構文規則を与える．その例を 2 に示す．

構文規則の与え方として形式言語の理論には，句構造文法，文脈依存文法，文脈自由文法，正規文法などがある．図 2 では，文脈自由文法を用いており，これは，プログラミング言語の設計や，コンパイラの作成に非常に有用である．

2.1.3　意味領域

意味領域とは，構文で書かれた要素に意味を記述させるために，それぞれ数学的領域に対応させたものいう．その例を図 3 に示す．A → B は集合 A から集合 B への関数全体を表す．

$$
\begin{aligned}
\xi ::= &\quad \mathrm{A} \mid \mathrm{B} \mid \cdots \mid \mathrm{Z} \\
\epsilon ::= &\quad 0 \mid \xi \mid \mathrm{suc}\ \epsilon \\
\tau ::= &\quad \xi\ :=\ \epsilon \mid \tau\ ;\ \tau \mid \\
&\quad \mathrm{for}\ \epsilon\ \mathrm{times\ do}\ \tau\ \mathrm{end}
\end{aligned}
$$

図 2 petit の構文規則 [4]

N 自然数の全体 (0 を含む)
$S = (\mathrm{Var} \rightarrow \mathrm{N})$ コンピュータの状態
$C = (\mathrm{S} \rightarrow \mathrm{S})$ コンピュータの状態の変換全体

図 3 petit の意味領域 [4]

2.1.4 意味関数

意味関数とは，構文領域から意味領域への関数を示す．意味領域が定められ，構文の要素とどのように対応するのかを決めなくてはならないので，意味領域上を動くメタ変数と，意味関数を導入しなければならない．その例を，図 4 に示し，意味関数の定義の例を，図 5，6 に示す．図 6 において，update: Var × N → C と，iterate: C × N → C は，図 7 のように定める．

$$
\begin{aligned}
&\mathcal{E} : \mathrm{Exp} \rightarrow (\mathrm{S} \rightarrow \mathrm{N}) \\
&\mathcal{C} : \mathrm{Pro} \rightarrow \mathrm{C} \\
&\nu : \mathrm{N} \qquad \sigma : \mathrm{S} \qquad \theta : \mathrm{C}
\end{aligned}
$$

図 4 関数 $\mathcal{S}$,$\mathcal{C}$ とメタ変数 ν,σ,θ[4]

$$
\begin{aligned}
\mathcal{E}[\![0]\!](\sigma) &= 0 \\
\mathcal{E}[\![\xi]\!](0) &= \sigma(\xi) \\
\mathcal{E}[\![\mathrm{suc}\ \varepsilon]\!](\sigma) &= \mathcal{E}[\![\varepsilon]\!](\sigma) + 1
\end{aligned}
$$

図 5 petit の意味関数 $\mathcal{E}$ の定義 [4]

$$
\begin{aligned}
\mathcal{C}[\![\xi\ :=\ \varepsilon]\!](\sigma) &= \mathit{update}(\xi, \mathcal{E}[\![\varepsilon]\!](\sigma))(\sigma) \\
\mathcal{C}[\![\tau_0\ ;\ \tau_1]\!](\sigma) &= \mathcal{C}[\![\tau_1]\!](\mathcal{C}[\![\tau_0]\!](\sigma)) \\
\mathcal{C}[\![\mathrm{for}\ \varepsilon\ \mathrm{times\ do}\ \tau\ \mathrm{end}]\!](\sigma) &= \mathit{iterate}(\mathcal{C}[\![\tau]\!], \mathcal{E}[\![\varepsilon]\!](\sigma))(\sigma)
\end{aligned}
$$

図 6 petit の意味関数 $\mathcal{C}$ の定義 [4]

$$
\begin{aligned}
\mathit{update}(\xi, \nu)(\sigma) &= \sigma\{\xi, \theta\} \\
\mathit{iterate}(\theta, \nu) &= \theta \bigcirc \theta \bigcirc \cdots \bigcirc \theta \\
&\quad (\nu\text{回}, \bigcirc\text{は関数の合成関数})
\end{aligned}
$$

$$
\sigma\{\xi, \nu\}(\eta) = \begin{cases} \nu & \text{if} \quad \eta = \xi \\ \sigma(\eta) & \text{if} \quad \eta \neq \xi \end{cases}
$$

図 7 update,iterate の定義 [4]

2.2 コンパイラ

コンパイラ [5] とは高水準言語のソースコードから機械語やアセンブリ言語などの低水準言語への変換器である．コンパイラはいくつかの工程を経て，高水準言語を低水準言語に変換する．また，各工程を経ている最中に 2 つの処理が並行して行われている．1 つは「Table Management」または「Book Keeping」という．これはプログラムで使用されている名前,，型などの本質的な情報を記憶する処理である. もう 1 つは「Error Handler」で，これはソースプログラム中に欠陥が検出されたときにプログラマへ報告をしたり，情報の調整を行う．

2.2.1 字句解析

ソースコードを字句（トークン）という単語単位に刻んでいく. トークンの例として，キーワード，識別子，定数, 演算子，カンマや括弧などの区切り記号などがある．解析された結果は次の工程である構文解析に渡される．

2.2.2 構文解析

A+B といったまとまった式などを解析．トークンが言語仕様によって許されているパターンと合致しているかを調べる．トークンを葉とする構文木にはめ込む．

2.2.3 中間コード生成

解析木をソースプログラムの中間言語表現に変換.

2.2.4 最適化

実行スピードを速くするために不要なコードを消したり，コードを変換することによってコードを小さくする．
例：A=2+3*4-5 → A=9

2.2.5 コード生成

最後の工程であり、ソースコードの意味を実現した目的のコードを生成する．本研究ではソースコードは SRL で，目的のコードは RL である．

2.2.6 例

以下にコンパイルの 1 例としてラムダ式からコンビネータ式へのコンパイルを示す．

コンビネータとは、実行環境に依存せずに完全にその振る舞いが定義された関数のことである．また、ラムダ計算で定義可能な任意の関数を表現可能な少数のコンビネータ集合が存在するときラムダ式のコンビネータへの変換が可能である．
コンビネータと対応するラムダ式を図 8 に示す．
コンビネータにおける式を図 9 に示す．

$$
\begin{aligned}
&\mathrm{S} : \lambda_x.\lambda_y.\lambda_z.(xz)(yz)\\
&\mathrm{k} : \lambda_x.\lambda_y.x\\
&\mathrm{P} : \lambda_x.\lambda_y(xy)\\
&\mathrm{F} : \lambda_x.x[1]\\
&\mathrm{N} : \lambda_x.[2]\\
&\mathrm{L} : \lambda_x.1(x)\\
&\mathrm{R} : \lambda_x.2(x)\\
&\mathrm{A} : \lambda_x.\lambda_y.\lambda_z(casexof1(x_1) \to yx_1, 2(x_2) \to zx_2\\
&\mathrm{X} : \lambda_x.fix(x)
\end{aligned}
$$

図 8　コンビネータと対応するラムダ式

$$
\begin{aligned}
&C ::= x \mid c|A|CC\\
&A ::= |S|K|P|F|N|L|R|A|X|
\end{aligned}
$$

x は変数集合、c は定数集合を表す．

図 9　コンビネータ式

各コンビネータについて対応する閉じたラムダ式の簡約と一致するように，簡約公理を図 10 に定義する．

$$
\begin{aligned}
\mathbf{S}\quad M_1\quad M_2\quad M_3\quad &\Rightarrow\quad (M_1M_3)\quad (M_2M_3)\\
\mathbf{K}\quad M_1\quad M_2\quad &\Rightarrow\quad M_1\\
\mathbf{F}\quad (\mathbf{P}\quad M_1\quad M_2)\quad &\Rightarrow\quad M_1\\
\mathbf{N}\quad (\mathbf{P}\quad M_1\quad M_2)\quad &\Rightarrow\quad M_2\\
\mathbf{A}\quad (\mathbf{L}\quad M_1)M_2\quad M_3\quad &\Rightarrow\quad M_2\quad M_1\\
\mathbf{A}\quad (\mathbf{R}\quad M_1)M_2\quad M_3\quad &\Rightarrow\quad M_3\quad M_1\\
\mathbf{X}\quad M\quad &\Rightarrow\quad M(\mathbf{X}\quad M)
\end{aligned}
$$

図 10　簡約公理

コンビネータ式 C に対して $\lambda *_x .C$ を図 11 に定義する．

これを用いてラムダ式のコンビネータ式への翻訳 $\overline{M}$ は

$$
\begin{aligned}
\lambda *_x .C &= KC \quad (x \notin FV(C) \text{ の場合})\\
\lambda *_x .x &= SKK\\
\lambda *_x .(C_1C_2) &= S(\lambda *_x .C_1)(\lambda *_x .C_2)
\end{aligned}
$$

図 11　$\lambda *_x .C$

図 12 のように与えられる．

$$
\begin{aligned}
\overline{x} &= x\\
\overline{c} &= c\\
\overline{\lambda_x.M} &= \lambda *_x .\overline{M}\\
\overline{M_1M_2} &= \overline{M_1}\overline{M_2}\\
\overline{(M_1\ M_2)} &= P\overline{M_1}\overline{M_2}\\
\overline{M[1]} &= F\overline{M}\\
\overline{M[2]} &= N\overline{M}\\
\overline{1(M)} &= L\overline{M}\\
\overline{2(M)} &= R\overline{M}\\
(\mathrm{case}M_1\mathrm{of}1(x) \Rightarrow M_2, 2(y) \Rightarrow M_3) &=\\
&A\overline{M_1}(\lambda *_x .\overline{M_2})(\lambda *_y .\overline{M_3})\\
\overline{\mathrm{fix}(M)} &= X\overline{M}
\end{aligned}
$$

図 12　ラムダ式のコンビネータ式への翻訳 $\overline{M}$

2.3　可逆フローチャート

可逆フローチャートは，従来のフローチャートが，幅広く使われている従来のプログラミング言語のモデルとして使用されているように，従来のフローチャートは，制御フローや，命令型のプログラムの構造を表している．但し，常に標準計算の基本理論を可逆言語に直接引き継げるとは限らない．更に，可逆フローチャートは，来のフローチャートと表面的に似ているにもかかわらず，重要な違いが存在する．それは，どのステップの計算過程でも，情報が失われることがなく，全てのステップが，順方向か逆方向に決定的になることである．これにより，可逆フローチャートの，計算能力や，可逆プログラミングの特性，理論的で，実用的な従来のフローチャートの形式との違いを正確に捉えることができる．可逆フローチャートは，幅広い目的で使用される．コンパイラによって生成されたマシンコードの低レベルな側面と，高いレベルの汎用的なブロック構造化言語や，反復や条件付き文に対応する．

2.3.1 SRL

可逆フローチャートの有用性を示すため，高水準言語である構造化可逆言語 (SRL) という具体的な可逆プログラミング言語が存在する．この言語は，横山らによって作成された可逆プログラミング言語である．[6]

構文規則については，図 13，図 15，図 16 に示す．SRL は，プログラムが再帰的に繰り返されている構文形式のブロックで構成されている．ブロックは，ステップ演算，ブロックのシーケンス，if 文から成る条件付き文，ループの 4 つの構文がある．ステップ操作に関して，push x1 x2 は，x1 の値を x2 の先頭に移動し，x1 をゼロクリアする．pop x1 x2 は,x2 の先頭を，ゼロクリアする必要がある最初の引数 x1 に移動する．skip については，何もしない．x ^= 0 の省略形とみなせる．

2.3.2 RL

SRL と同じく，可逆フローチャートの有用性を示すため，低水準言語である非構造化可逆言語 (RL) という具体的な可逆プログラミング言語が存在する．この言語も，横山教授らによって作成された可逆プログラミング言語である．[6]

構文規則については，図 14，図 15，図 16 に示す．RL のブロックはラベル l，アサーション k から来るステップ演算，ジャンプの要素から構成されている．アサーションからの取得は無条件がある．つまり，ブロック l から行われる，非構造化プログラムには，l つの entry と l つの exit が含まれている．

SRL,RL の翻訳規則を図 17、図 18 に示す．このように，相互に翻訳規則が存在し，翻訳可能であるが，SRL から RL へ翻訳する機械は，現状存在していない．

2.3.3 構文

SRL,RL の 2 つの可逆フローチャート言語の構文を以下図 13，14，15，16 に示す.

$$
\begin{array}{lcl}
p & ::= & b \\
b & ::= & a \\
 & | & b \quad b \\
 & | & \text{if } e \text{ then } b \text{ else } b \text{ fi } e \\
 & | & \text{from } e \text{ do } b \text{ loop } b \text{ until } e
\end{array}
$$

図 13 SRL のブロックの構文 ([6] より)

$$
\begin{array}{lcl}
q & ::= & d^{+} \\
d & ::= & l \ : \ k \ a^{*} \ j \\
k & ::= & \text{from } l \\
 & | & \text{fi } e \text{ from } l \text{ else } l \\
 & | & \text{entry} \\
j & ::= & \text{goto } l \\
 & | & \text{if } e \text{ goto } l \text{ else } l \\
 & | & \text{exit}
\end{array}
$$

図 14 RL の構文 ([6] より)

$$
\begin{array}{rcl}
a & ::= & x \oplus = \ e \ \mid \ x[e] \oplus = \ e \ \mid \ \text{push } x \ x \ \mid \ \text{pop } x \ x \ \mid \ \text{skip} \\
e & ::= & c \ \mid \ x \ \mid \ x[e] \ \mid \ e \otimes e \ \mid \ \textbf{top} \ x \ \mid \ \textbf{empty} \ x \\
c & ::= & 0 \ \mid \ 1 \ \mid \ \cdots \ \mid \ 4294967295 \\
\otimes & ::= & \oplus \ \mid \ * \ \mid \ / \ \mid \ = \ \mid \ < \ \mid \ > \ \mid \ <= \ \mid \ > \ \mid \ != \\
\oplus & ::= & + \ \mid \ - \ \mid \ \hat{} \ \ (\text{xor})
\end{array}
$$

図 15 可逆ステップ演算と式の構文 ([6] より)

$$
\begin{array}{rl}
\text{SRL}: & p \in \text{SRL} \quad b \in \text{Blk} \\
\text{RL}: & q \in \text{RL} \quad d \in \text{RLblk} \quad j \in \text{Jump} \quad k \in \text{From} \quad l \in \text{Label} \\
\text{SRL, RL}: & a \in \text{Step} \quad e \in \text{Exp} \quad c \in \text{Const} \quad x \in \text{Var} \quad \oplus, \otimes \in \text{Op}
\end{array}
$$

図 16 SRL と RL の構文ドメイン ([6] より)

2.3.4 SRL,RL の翻訳規則

SRL から RL への翻訳規則を図 17 に，RL から SRL への翻訳規則を図 18 に示す.

$$
\begin{array}{l}
\mathcal{T}_{SRL}[\![b]\!] \ = \\
\quad l_0 \ : \quad \text{entry} \\
\qquad\quad \text{goto } l_1 \\
\quad \mathcal{T}[\![b]\!] \ (\ l_0\ ,\ l_1\ ,\ l_2\ ,\ l_3\ ) \\
\quad l_3 \ : \quad \text{from } l_2 \\
\qquad\quad \text{exit} \\
\quad \text{where } l_0\ ,\ l_1\ ,\ l_2\ ,\ l_3 \text{ are fresh}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{T}[\![b_1\ b_2]\!] \ (\ l_0\ ,\ l_1\ ,\ l_4\ ,\ l_5\ ) = \\
\quad \mathcal{T}[\![b_1]\!] \ (\ l_0\ ,\ l_1\ ,\ l_2\ ,\ l_3\ ) \\
\quad \mathcal{T}[\![b_2]\!] \ (\ l_2\ ,\ l_3\ ,\ l_4\ ,\ l_5\ ) \\
\quad \text{where } l_2\ ,\ l_3 \text{ are fresh}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{T}[\![a]\!] \ (\ l_0\ ,\ l_1\ ,\ l_2\ ,\ l_3\ ) = \\
\quad l_1 \ : \quad \text{from } l_0 \\
\qquad\quad a \\
\qquad\quad \text{goto } l_2 \\
\quad l_2 \ : \quad \text{from } l_1 \\
\qquad\quad \text{goto } l_3 \\
\mathcal{T}[\![\text{if } e_1 \text{ then } b_1 \text{ else } b_2 \text{ fi } e_2\ ]\!] \ (\ l_0\ ,\ l_1\ ,\ l_6\ ,\ l_7\ ) = \\
\quad l_1 \ : \quad \text{from } l_0 \\
\qquad\quad \text{if } e_1 \text{ goto } l_2 \text{ else } l_4 \\
\mathcal{T}[\![b_1]\!] \ (\ l_1\ ,\ l_2\ ,\ l_3\ ,\ l_6\ ) \\
\mathcal{T}[\![b_2]\!] \ (\ l_1\ ,\ l_4\ ,\ l_5\ ,\ l_6\ ) \\
\quad l_6 \ : \text{fi } e_2 \text{ from } l_3 \text{ else } l_5 \\
\qquad \text{goto } l_7 \\
\quad \text{where } l_2\ ,\ l_3\ ,\ l_4\ ,\ l_5 \text{ are fresh}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{T}[\![\text{from } e_1 \text{ do } b_1 \text{ loop } b_2 \text{ until } e_2\ ]\!] \ (\ l_0\ ,\ l_1\ ,\ l_6\ ,\ l_7\ ) = \\
\quad l_1 \ : \text{ fi } e_1 \text{ from } l_0 \text{ else } l_6 \\
\qquad\quad \text{goto } l_2 \\
\quad \mathcal{T}[\![b_1]\!] \ (\ l_1\ ,\ l_2\ ,\ l_3\ ,\ l_4\ ) \\
\quad \mathcal{T}[\![b_2]\!] \ (\ l_4\ ,\ l_5\ ,\ l_6\ ,\ l_1\ ) \\
l_4 \ : \quad \text{from } l_3 \\
\qquad \text{if } e_2 \text{ goto } l_7 \text{ else } l_5 \\
\text{where } l_2\ ,\ l_3\ ,\ l_5\ , l_6 \text{ are fresh}
\end{array}
$$

図 17 SRL から RL への翻訳規則 ([6] より)

$$
\begin{aligned}
\mathcal{T}_{RL}[\![q]\!] \;=\;& \text{if } \wedge_{i=0}^{n} \wedge_{j=1}^{n+1} \wedge_{k=0}^{2} \neg x_{i,j}^{k} \text{ fi true} \\
& x_{0.1}^{0} \mathbin{\hat{}}= \text{true} \\
& \text{from } \mathrm{x}_{0.1}^{0} \text{ do} \\
& \quad \mathcal{T}_{blks}[\![q]\!] \\
& \text{until } \mathrm{x}_{n,n+1}^{0} \\
& x_{n,n+1}^{0} \mathbin{\hat{}}= \text{true} \\
& \text{if } \wedge_{i=1}^{n} \wedge_{j=0}^{n+1} \wedge_{k=0}^{2} \neg x_{i,j}^{k} \text{ fi true}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}_{blks}[\![d_1\ d_2 \cdot\cdot\cdot d_n]\!] &= \mathcal{T}_{blks}[\![d_1]\!](\mathcal{T}_{blks}[\![d_2]\!](\cdot\ \cdot\ (\mathcal{T}_{blks}[\![d_n]\!](E_{\text{false}}))\ \cdot\ \cdot)) \\
\mathcal{T}_{blk}[\![l_i\ :\ k\ a^*\ j]\!](F) &= \mathcal{T}_{flow}[\![k]\!](i\ , \mathcal{T}_{step}[\![a^*]\!](i\ , \mathcal{T}_{flow}[\![j]\!](i\ , \mathrm{F})))
\end{aligned}
$$

$$
\mathcal{T}_{flow}[\![\text{fi } e \text{ from } l_j \text{ else } l_k]\!](i\ , F) = \text{if } x_{j,i}^{0} \vee x_{k,i}^{0} \text{ then if } x_{j,i}^{0} \text{ then } P_{j,i} \text{ else } P_{k,i} \text{ fi } e \text{ else } F \text{ fi } x_{i,i}^{1}
$$

$$
\mathcal{T}_{flow}[\![\text{if } e \text{ goto } l_j \text{ else } l_k]\!](i\ , F) = \text{if } x_{i,i}^{2} \text{ then if } e \text{ then } R_{i,j} \text{ else } R_{i,k} \text{ fi } x_{i,j}^{0} \text{ else } F \text{ fi } x_{i,j}^{0} \vee x_{i,k}^{0}
$$

$$
\begin{aligned}
\mathcal{T}_{flow}[\![\text{from } l_j]\!](i\ , F) &= \text{if } x_{j,i}^{0} \text{ then } P_{j,i} \text{ else } F \text{ fi } x_{i,i}^{1} \\
\mathcal{T}_{flow}[\![\text{goto } l_j]\!](i\ , F) &= \text{if } x_{i,i}^{2} \text{ then } R_{i,j} \text{ else } F \text{ fi } x_{i,j}^{0}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}_{flow}[\![\text{entry}]\!](i\ , F) &= \text{if } x_{j,i}^{0} \text{ then } P_{j,i} \text{ else } F \text{ fi } x_{i,i}^{1} \\
\mathcal{T}_{flow}[\![\text{exit}]\!](n\ , F) &= \text{if } x_{n,n}^{2} \text{ then } R_{n,n+1} \text{ else } F \text{ fi } x_{n,n+1}^{0}
\end{aligned}
$$

$$
\mathcal{T}_{steps}[\![a^*]\!](i\ , F) = \text{if } x_{i,i}^{1} \text{ then } a^*\ ;\ P_{i,i} \text{ else } F \text{ fi } x_{i,i}^{2}
$$

図 18 RL から SRL への翻訳規則

([6] より)

図 17 について，SRL のブロック b が 2 つで構成されている場合，2 つの新しいラベルを用いて，個別に変換される．ステップ演算である a は，l_0 から受け取るブロック l_1 と，制御を l_3 に渡すブロック l_2 に変換される．条件文と，ループ文には，互いに類似しているが，b_1 と b_2 によるブロック間の制御フローつなぎが異なる．2 つのブロックは，内部の制御フローをつなぐ 4 つの新しいラベルを使用して，$\mathcal{T}$ への再帰的な呼び出しによって個別に変換される．

また，本研究では，SRL から RL への翻訳器を実装することを目的とするため，図 18 については利用しない．

2.3.5 SRL/RL の評価規則

2.3.6 SRL/RL の逆変換規則

2.3.7 SRL/RL の拡張

2.4 可逆プログラミング言語

1 章で述べたように，プログラミング言語には，高水準言語と低水準言語が存在し，可逆プログラミング言語にもその分類の仕方が適用できる．零例として，可逆プログラミング言語には，SRL，RL の他に，高水準の命令型可逆言語である Janus[7], R[8], SyReC[9], 低水準の命令型可逆言語である Bob[10], PISA[8] などが存在する．

2.4.1 Janus

Janus とは初の可逆構造化プログラミング言語であると思われる．シンプルでかつ強力であり，可逆言語を設計するうえで便利なモデルとなる．文は可逆制御フロー演算子（条件分岐，ループ），スタック操作（push,pop），ローカル変数ブロック，プロシージャ呼び出し，スキップまたはステートメントシーケンスから構成されている．

2.4.2 R

R は，手続き型言語であり，厳密な関数型言語ではなく，可逆機械用のプログラミング言語である．言語は現在不完全であり，安定していない．この言語は，C 言語の様な単純な配列，for ループ，if 文，および引数のある再帰サブルーチンを支持している．更に，Lisp の様に，記号と数値の括弧で囲まれ，ネストされたリストを使用して表される．R コンパイラは，R ソースを Pendulum アセンブリコードに変換する．

2.4.3 SyReC

SyReC は複雑なロジックの合成を一般的な HDL 記述を介して可能にする．SyReC プログラムは本質的に可逆的であり、同時に仕様も保証される．可逆プログラミング言語の Janus を取り入れたことにより以前よりも可逆回路に焦点を当てたハードウェア関連の言語が強化された．

2.4.4 Bob

Bob とは，BobISA という命令セットを備えた可逆計算アーキテクチャである．命令セットは算術・論理命令，分岐命令，メモリ命令の 3 つのタイプの命令をもつ．このアーキテクチャの特徴は，シンプルで表現力があるローカルで反転可能な命令セットや，可逆な制御ロジックとアドレス計算が導入されていることである．さまざまな計算機はエネルギー消費という問題を抱えているが，Bob のような可逆計算アーキテクチャにより消費電力や放熱を削減することが可能である．

2.4.5 PISA

PISA は，Pendulum リバーシブルマイクロプロセッサ用のアセンブリ言語である．この言語の命令セットは，R のコンパイラを対象としていた．

3 目的

SRL と RL の 字句解析器・構文解析器・プリティプリンタを書き、表示的意味論/操作的意味論も書く．更に，SRL から RL への翻訳規則を書き，翻訳器を Haskell で実装する．また、SRL/RL の評価規則や逆変換規則を書き，インタープリタや逆変換規則も実装する．

4 実装

4.1 SRL から RL への翻訳機の実装

4.2 インタープリタの実装

4.3 逆変換規則の実装

4.4 SRL/RL の拡張部分の実装

5 おわりに

参考文献

[1] (eds), K.J.: Compiler Construction, *Clean Translation of an Imperative Reversible Programming Language*, Vol.6601, pp.144–163 (2011).

[2] Moriyama, K.: Theoretical properties of reversible flowchart programming languages, Master's thesis, University of Copenhagen (2009).

[3] Moriyama, K.: An Introduction to Reversible Programming Using Simple Reversible Flowchart Languages, Master's thesis, University of Copenhagen (2009).

[4] 田辺　誠，中島玲二，長谷川真人：コンピュータサイエンス入門〈論理とプログラム意味論，岩波書店 (1999).

[5] 大堀　淳，ジャックガリグ，西村　進：コンピュータサイエンス入門≪アルゴリズムとプログラム言語≫，岩波書店 (1999).

[6] Tetsuo Yokoyama, Holger Bock Axelen, R.G.: Fundamentals of reversible flowchart languages, *Theoretical Computer Science*, Vol.611, pp.87–115 (2016).

[7] Tetsuo Yokoyama, Holger Bock Axelen, R.G.: Principles of a Reversible Programming Language, *Proc.Computing frontiers*, pp.43–54 (2008).

[8] Frank, M.P.: Reversibility for Efficient Computing, *PhD Thesis, MIT* (1999).

[9] Wille, R., O.S.D.R.: SyReC: A Programming Language for Synthesis of Reversible Circuits, Kaźmierski, T. J. and Morawiec, A., *System Specification and Design Languages*, Vol.106, pp.207–222 (2012).

[10] Thomsen, M. K., A.H. and Glück, R.: A Reversible Processor Architecture and Its Reversible Logic Design, Reversible Computation, *A. Vos and R. Wille (Eds.), Springer-Verlag*, Vol.7165, pp.30–42 (2012).