

卒 業 論 文

可逆プログラミング言語 R-WHILE による 万能可逆チューリング機械の構成

2014SE006 青木 峻
2014SE059 増田 大輝
2014SE089 柴田 心太郎

指導教員 横山 哲郎

2017 年 12 月

南山大学 理工学部 ソフトウェア工学科

Constructing of universal reversible Turing machine by using the reversible programming language R-WHILE

2014SE006 AOKI Ryo
2014SE059 MASUDA Hiroki
2014SE089 SHIBATA Shintaro

Supervisor YOKOYAMA Tetsuo

December 2017

Department of Software Engineering
Faculty of Science and Engineering
Nanzan University

要約

可逆プログラミング言語 R-WHILE の計算モデルは，万能可逆チューリング機械と同じ計算能力があるとされている．しかし，筆者の知る限りにおいて，その具体的な証明についてはこれまで報告が無かった．本稿では任意の可逆チューリング機械を，意味が同じ R-WHILE プログラムに変換できることを示す．このことにより，可逆万能チューリング機械の R-WHILE プログラムが構成できることを示す．

Abstract

目次

第 1 章	序論	1
1.1	導入	1
1.2	背景	1
1.3	期待される効果	1
1.4	役割分担	1
第 2 章	関連研究	2
2.1	可逆計算	2
2.2	構造化定理	2
2.3	可逆プログラミング言語	2
2.4	WHILE 言語	3
2.5	セルオートマトン	3
2.6	万能性	3
第 3 章	可逆チューリング機械	5
3.1	チューリング機械	5
3.2	可逆チューリング機械	8
3.3	3 テープ可逆チューリング機械	11
第 4 章	可逆チューリング機械から R-WHILE への変換	14
4.1	R-WHILE について	14
4.2	可逆チューリング機械から R-WHILE への変換	19
4.3	模倣できていることの証明	19
第 5 章	結論	23
5.1	結果	23
5.2	考察	23
	参考文献	24

第 1 章

序論

1.1 導入

可逆プログラミング言語 R-WHILE は、命令型の可逆プログラミング言語であり、意味論と構文規則がほかの可逆プログラミング言語より少ないため、単純な言語であるといえる。そのため、可逆なプログラムや、可逆なプログラムの振る舞いについての定理を証明する際に重宝される。しかし、R-WHILE の計算能力の高さは文献 [?] によって仮定されているが、筆者の知る範囲では R-WHILE で任意の可逆チューリング機械を構成できるほどの計算能力をもつという報告はない。そのため、R-WHILE は、可逆チューリング機械の知見を得ることが出来ていない。

1.2 背景

計算できるという概念をチューリング機械で計算できるということにしようという主張は広く認められている [1]。チューリング機械は計算の効率を問題としなければ現在のコンピュータをも模倣できるとされている強力な計算モデルであり、計算可能性理論を議論する際に重要である。任意のチューリング機械を模倣できる計算システムは計算万能性 (チューリング完全) をもつといわれる。プログラミング言語の計算モデルが計算万能性をもつことを示すことは、ほかのプログラミング言語との知見を共有する上で重要である。

1.3 期待される効果

我々は本研究において、R-WHILE で任意の可逆チューリング機械を模倣し、R-WHILE が可逆的計算万能性を持つことを証明する。これにより、R-WHILE は可逆チューリング機械や、可逆的計算万能性をもつ可逆プログラミング言語との知見を共有することができるようになる。

1.4 役割分担

増田大輝は第 1 章、第 2 章を、柴田心太郎は第 3 章と R-WHILE のプログラムを、青木峻は第 4 章、第 5 章を担当した。

第 2 章

関連研究

本章では、可逆性やプログラム言語において本稿に関連する研究、用語について説明する。

2.1 可逆計算

可逆計算とは、計算過程において、初期状態と最終状態を除いたすべての状態が、直前と直後にとり得る状態を高々一つもつものである。Landauer は、計算機において非可逆な演算はエネルギーの散逸を伴うことを指摘した。可逆的な演算はこのような不可避なエネルギーの散逸を減らす 1 つの解決策として用いられる。可逆計算では、入力から出力までの過程を出力から逆算して入力を求めることが可能であるため、情報を消失することなく出力結果を導くことが出来る [5]。

2.2 構造化定理

1 つの入り口と 1 つの出口を持つようなアルゴリズムには 3 つの基本構造がある。1 つは順次と呼ばれるプログラムに記されている順に随時処理を行う構造である。2 つ目は繰り返しと呼ばれ、ある条件が成立するならば処理 A を行い、そうでなければ処理 B を行なう。3 つ目は、一定の条件が満たされている間処理を繰り返す選択である。この 3 つの基本構造の組み合わせを利用してアルゴリズムを記述することができる [?]。

2.3 可逆プログラミング言語

可逆プログラミング言語とは、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である。可逆プログラミング言語の例を挙げると、Janus、R などが存在する。可逆プログラミング言語は、可逆であることの形式的証明されている。それらの言語は非可逆なプログラムを記述することができないため、可逆プログラミング言語で書かれた任意のプログラムの可逆性が保証される。

2.3.1 Janus

Janus とは、多重集合と配列の書換えに基づく制約処理モデルを持つ可逆プログラミング言語の一種。可逆プログラミング言語である Janus は、C 言語に似た構文に加えて可逆性を保証するための構文規則を持つ。Janus の変数の基本型は R-WHILE とは異なり、整数型、整数型の配列とスタックである。そのため各データへのアクセスが容易である。また、Janus には複合代入演算子 ($+=, -=, !=$) と交換演算子 ($:$) が存在するため、変数の変更が可能である。その場合、複合代入演算子は右辺の式を評価し、演算子に応じて左側の変数を変更する演算子であり、交換演算子は左右の変数の値を交換する演算子である。この 2 つの演算子を利用する場合、単射な計算を行う記述であるために、右側の式や変数に、左側の変数が利用されてはならない。これは添字演算子式の場合でも同様に、その添字が右側の式や変数に利用されてはならない。

2.4 WHILE 言語

WHILE 言語とは、ローカル変数、整数、真偽式をもつ単純な命令型言語である。WHILE は構文規則や意味論が他のプログラム言語と比べ少なく、チューリング機械を模倣できるくらいの計算能力を持っているため、プログラムやプログラムの振る舞いについての定理を証明する場合に重宝される。構造化定理の基本となる順次、分岐、繰り返しを表現することができ、分岐の場合 if-else 文、繰り返しの場合 while 文がある。その他には、変数に整数式を代入する割り当て、ユーザーからの入力を読み取って画面に出力する文の読み書き文、効果のないスキップ文がある。この言語には、0 から n の値または 0 と 1 の結果のパラメータを取ることができる手続きが含まれている [9]。

2.5 セルオートマトン

セルオートマトンとは 1950 年代に Neumann が自己増殖オートマトンを設計するための理論的枠組みとして提案されたモデルである。いくつかの状態を持つセルという単位によって構成され、事前に設定された規則に従い、そのセル自身や近傍の状態によって、時間発展と共に、その時間における各セルの状態によって決定される。現在では、計算システムの数理モデルの一種として扱われ、計算の基礎理論をはじめとして、交通流や生物などのシステムのシミュレーションに用いられている。標準的な CA は

$$(\mathbb{Z}^k, Q, N, f, \#)$$

として定める。ただし、 Q はセルの状態と呼ばれる空でない有限集合、 $N(= \{n_1, \dots, n_m\})$ は近傍と呼ばれる \mathbb{Z}^k の部分集合、局所関数 $f: Q^m \rightarrow Q$ は各セルの状態遷移を定めるものとする。なお、 $\# \in Q$ は静止状態と呼ばれ、 $f(\#, \dots, \#) = \#$ を満たす。集合 Q 上の k 次元の状相とは $\alpha: \mathbb{Z}^k \rightarrow Q$ であるような写像 α をいう。 Q 上の k 次元状相すべての集合を $\text{Conf}_k(Q) = \{\alpha \mid \alpha: \mathbb{Z}^k \rightarrow Q\}$ で表す。状相 α は、集合 $\{x \mid x \in \mathbb{Z}^k \wedge \alpha(x) \neq \#\}$ が有限であるとき、有限と呼ばれる。

大域関数 $F: \text{Conf}_k(Q) \rightarrow \text{Conf}_k(Q)$ を

$$\begin{aligned} \forall \alpha \in \text{Conf}_k(Q), \forall x \in \mathbb{Z}^k : \\ F(\alpha)(x) = f(\alpha(x + n_1), \dots, \alpha(x + n_m)) \end{aligned} \quad (2.1)$$

と定める [5]。

2.5.1 可逆セルオートマトン

可逆セルオートマトン (以下 RCA) とは、可逆性が保証されたセルオートマトン (以下 CA)、つまり、現在の状態から 1 つ前の状態を一意に決めることができる CA のことである。RCA は、以下の条件を満たしている。

1. 大域関数 F が単射であること。
2. 状相の遷移がちょうど逆であるような CA が存在すること。

1 次元、2 次元 RCA は計算可逆万能性をもつモデルが数多く発見されている。Toffoli は RCA の次元を 1 増やすことで、2 次元以上の RCA において任意の TM が模倣できることを示し、そこから分割 CA の枠組みを利用して、1 次元以上の RCA が計算可逆万能性を持つことを証明した [5]。

2.6 万能性

チューリング機械における万能性とは、任意の計算モデルをチューリング機械の動作に還元できるということである。また、チューリング機械で任意の計算モデルが模倣できるということから、任意の計算モデルやブ

プログラムにおいて、任意のチューリング機械を計算、若しくは記述できる能力があれば、任意の計算モデルも計算できる。このとき、任意のチューリング機械を計算、記述できる能力があることを、チューリング完全（計算万能性）という。

第 3 章

可逆チューリング機械

この章ではチューリング機械とチューリング機械に制約を加えた可逆チューリング機械の定義，及び具体例を記述する．

3.1 チューリング機械

あるシステムにおいて，実行すべきことに曖昧さがなく明確に記述されていて，記述されたことを機械的に実行すれば出力が得られるものとする．このときそのように記述されたものを機械的な手順，あるいはアルゴリズムと呼ぶ [3]．また，アルゴリズムの特徴として実効性，有限性がある．実効性は，その手順を実際に行うことのできる性質のことであり，有限性は，実行する時間が有限であることを示す．この機械的な手順は直感的な概念であるが，Turing はこの概念を厳密に定義することを試み，1936 年にチューリング機械と呼ばれる理論的計算モデルを導入した．チューリング機械は現在のコンピュータをも模倣できるとされている．そのため，任意のチューリング機械を模倣できる計算システムは計算万能性をもつといわれる．

また，1943 年に Kleen によって，機械的な手順で計算できるということはチューリング機械の枠組みの中で定式化できるという提唱 (チャーチ・チューリングのテーゼ) がなされた [1]．

3.1.1 チューリング機械の動作

チューリング機械はマス目に分割された左右に無限長のテープをもち，有限制御部，及びテープ上の記号を読み書きするためのヘッドから構成されている [図 3.1]．テープには予め入力情報 (2 進数，多数のアルファベット) が格納されており，ヘッドが位置するマス目の記号を読み取る．そして，この記号と現在の有限制御部の状態 (内部状態，図 3.1においては状態 q を指す) に依存して，マス目の記号を書き換え，ヘッドを左か右に 1 コマ移動もしくは不動にし，内部状態を遷移させる．チューリング機械は，この一連の動作を繰り返すことで計算を行う．

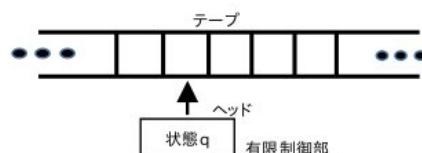


図 3.1: チューリング機械の模式図

3.1.2 定義

本稿では、1 テープチューリング機械を $T = (Q, \Sigma, b, \delta, q_s, Q_f)$ として定める。(以下、1 テープチューリング機械のことをチューリング機械と呼ぶ。) ただし Q は内部状態の空でない有限集合、 Σ はテープ記号の空でない有限集合、 $b(\in \Sigma)$ は空白記号でテープの有限個のマスを除く残り全てのマス目に b が格納されていると仮定する。 δ は $(Q \times [\Sigma \times \Sigma] \times Q) \cup (Q \times \{ \leftarrow, \rightarrow \} \times Q)$ の部分集合、 $q_s(\in Q)$ は初期状態、 $Q_f(\subset Q)$ は最終状態の集合とする。

δ は遷移規則の集合である。矢印 (\leftarrow, \rightarrow) はそれぞれヘッドの移動 (左, 不動, 右) を表す。遷移規則は 3 項組であり、 $(q, \langle s, s' \rangle, q')$ または (q, d, q') である ($q, q' \in Q, s, s' \in \Sigma, d \in \{ \leftarrow, \rightarrow \}$)。前者の 3 項組は T が状態 q で記号 s を読んだ場合、記号 s' に書き換え、状態を q' にすることを意味する。また、後者の 3 項組は T が状態 q の場合ヘッドを d の方向に動かし、状態を q' にすることを意味する。

ここで、チューリング機械が計算を行っているある時点での様子を様相とする。このとき、チューリング機械 $T = (Q, \Sigma, b, \delta, q_s, Q_f)$ の様相とは組 $(q, (l, s, r)) \in Q \times ((\Sigma \setminus \{b\})^* \times \Sigma \times (\Sigma \setminus \{b\})^*)$ である。ここで V^* は V 中の記号を 0 個以上並べたものの集合を表す。ただし q は内部状態、 s はヘッドの上にあるテープの中の記号、 l はヘッドの左側のテープを表す記号列、 r はヘッドの右側のテープを表す記号列を表す。

チューリング機械 $T = (Q, \Sigma, b, \delta, q_s, Q_f)$ の計算ステップは、 $c \xRightarrow{T} c'$ を満たすように様相 c を様相 c' に移すものとする。ただし、ここで、 b つまり空白記号が無限に続くときを λ として

$$\begin{aligned} (q, (l, s, r)) &\xRightarrow{T} (q', (l, s', r)) && \text{if } (q, \langle s, s' \rangle, q') \in \delta \\ (q, (\lambda, s, r)) &\xRightarrow{T} (q', (\lambda, b, sr)) && \text{if } (q, \leftarrow, q') \in \delta \\ (q, (ls', s, r)) &\xRightarrow{T} (q', (l, s', sr)) && \text{if } (q, \leftarrow, q') \in \delta \\ (q, (ls, b, \lambda)) &\xRightarrow{T} (q', (l, s, \lambda)) && \text{if } (q, \leftarrow, q') \in \delta \\ (q, (l, s, r)) &\xRightarrow{T} (q', (l, s, r)) && \text{if } (q, \downarrow, q') \in \delta \\ (q, (l, s, \lambda)) &\xRightarrow{T} (q', (ls, b, \lambda)) && \text{if } (q, \rightarrow, q') \in \delta \\ (q, (l, s, s'r)) &\xRightarrow{T} (q', (ls, s', r)) && \text{if } (q, \rightarrow, q') \in \delta \\ (q, (\lambda, b, sr)) &\xRightarrow{T} (q', (\lambda, s, r)) && \text{if } (q, \rightarrow, q') \in \delta \end{aligned}$$

である。 \xRightarrow{T} の反射推移閉包を \xRightarrow{T}^* と記す。

チューリング機械 $T = (Q, \Sigma, b, \delta, q_s, \{q_f\})$ の意味をとして、 $\llbracket T \rrbracket = \{(r, r') | (q_s, (\lambda, b, r)) \xRightarrow{T}^* (q_f, (\lambda, b, r'))\}$ とする。これは初期状態 q_s でテープ内が (λ, b, r) の様相であるとき、遷移を繰り返し行った結果が最終状態 q_f でテープ内が (λ, b, r') の様相になるということを表している。

3.1.3 チューリング機械の例

これまでに定義したチューリング機械に基づいて、簡単なチューリング機械を考えてみる。

例 与えられた 2 進数に 1 を加えるチューリング機械 $T_1 = (Q_1, \{b, 0, 1\}, b, \delta_1, q_s, \{q_f\})$ を考える。ただし、 $Q_1 = \{q_s, q_1, q_2, q_3, q_4, q_f\}$ であり、 δ_1 は以下の 3 項組の集合である。

$$\begin{aligned} \delta_1 = \{ & [q_s, \langle b, b \rangle, q_1], [q_1, \rightarrow, q_2], \\ & [q_2, \langle 1, 0 \rangle, q_1], \\ & [q_2, \langle 0, 1 \rangle, q_3], [q_3, \leftarrow, q_4], \end{aligned}$$

$$\begin{aligned}
&[q_2, \langle b, 1 \rangle, q_3], \\
&[q_4, \langle 0, 0 \rangle, q_3], \\
&[q_4, \langle 1, 1 \rangle, q_3], \\
&[q_4, \langle b, b \rangle, q_f]
\end{aligned}$$

T_1 は、非負整数 n の 2 進数表現が書かれたテープが与えられ、ヘッドを 2 進数表現の左隣のます目に置いて初期状態 q_s から動作を開始したとき、 n を n に 1 を加えたものに書き換え、2 進数表現の左隣のます目までヘッドを移動し、最終状態 q_f で停止するチューリング機械である（テープに書かれる 2 進数表現は反転されたものとする）。このとき、 T_1 は 2 進数表現の一番下位の桁から読んでいく。1 を読んだとき、1 を 0 に書き換える。また、0 または b を読み込んだときにそれを 1 に書き換える。各状態における T_1 の動作を以下に説明する。また実際の動作の例は（図 3.2）である。このとき、予め、テープに格納されている 2 進数を「101」とする。t は計算の進行（遷移）を表す。

q_s : 2 進数表現 n の左隣の b を読み、 q_1 へ。

q_1 : ヘッドを右に 1 コマ移動し、 q_2 へ。

q_2 : 1 を読んだとき、0 に書き換え q_1 へ。 b または 0 を読んだとき 1 に書き換え、 q_3 へ。

q_3 : ヘッドを左に 1 コマ移動し、 q_4 へ。

q_4 : 0 または 1 を読んだ場合、 q_3 へ、 b を読んだ場合 q_f へ。

また、この遷移規則による様相を 3.1.2 節の様相の計算ステップの記法を用いて表わす。

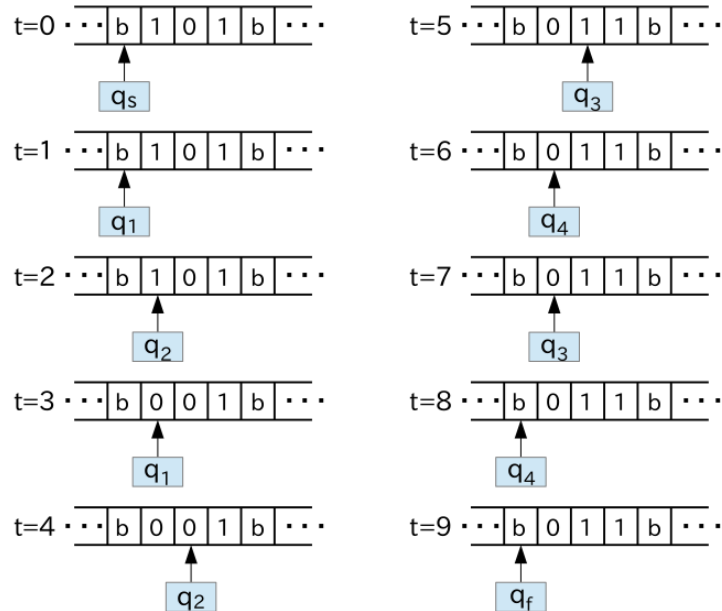


図 3.2: チューリング機械の動作例

$$\begin{array}{llll}
t = 0 & (q_s, (\lambda, b, 1)) & \xRightarrow{T_1} & (q_1, (\lambda, b, 1)) \\
t = 1 & & \xRightarrow{T_1} & (q_2, (b, 1, 0)) \\
t = 2 & & \xRightarrow{T_1} & (q_1, (b, 0, 0)) \\
t = 3 & & \xRightarrow{T_1} & (q_2, (0, 0, 1)) \\
t = 4 & & \xRightarrow{T_1} & (q_3, (0, 1, 1)) \\
t = 5 & & \xRightarrow{T_1} & (q_4, (b, 0, 1)) \\
t = 6 & & \xRightarrow{T_1} & (q_3, (b, 0, 1)) \\
t = 7 & & \xRightarrow{T_1} & (q_4, (\lambda, b, 0)) \\
t = 8 & & \xRightarrow{T_1} & (q_3, (\lambda, b, 0)) \\
t = 9 & & \xRightarrow{T_1} & (q_f, (\lambda, b, 0))
\end{array}$$

3.1.4 n テープチューリング機械

前小節までは、1 テープのみを考えたが、テープとヘッドがそれぞれ n 個ある n テープチューリング機械も存在する。 n テープチューリング機械 T' は $T' = (Q, (\Sigma_1, \Sigma_2, \dots, \Sigma_n), b, \delta, q_s, Q_f)$ として定める。 Q, b, q_s, Q_f は 1 テープチューリング機械と同様であり、 $\Sigma_1, \Sigma_2, \dots, \Sigma_n$ はそれぞれ n 本目のテープで利用されるテープ記号の空でない有限集合。 δ は $((Q \setminus Q_f) \times [\Sigma_1 \times \Sigma_2 \times \dots \times \Sigma_n] \times [\Sigma_1 \times \Sigma_2 \times \dots \times \Sigma_n]) \times Q \cup ((Q \setminus Q_f) \times [d_1 \times d_2 \times \dots \times d_n] \times Q)$ の部分集合である。ただし、 d はヘッド移動の表す矢印 $[\quad , \quad]$ である。また、任意の n テープチューリング機械に対して、それと等価な 1 テープチューリング機械が存在する [3]。

3.1.5 非決定性チューリング機械

ここまでは、現在の状態においてテープ内の記号が決まれば、次に行う遷移が一意に決まる決定性チューリング機械のみを扱ったが、そうではない非決定性チューリング機械も存在する。このとき、任意の非決定性チューリング機械に対して、それと等価なチューリング機械が存在することも知られている。[3] これは非決定的に選ばれる状態を計算木として表し、その計算木の全節点を巡回する順番を固定しておけば遷移は決定的になると言ったものである。

3.2 可逆チューリング機械

この節では、前節で定義したチューリング機械に制約を加えることで可逆性をもたせたチューリング機械である、可逆チューリング機械について説明する。

3.2.1 定義

チューリング機械 $T = (Q, \Sigma, b, \delta, q_s, Q_f)$ において、任意の異なる遷移規則 $m_1 = [p, \langle s, t \rangle, q]$ 、 $m_2 = [p', \langle s', t' \rangle, q'] \in \delta$ に対して、 $p = p'$ ならば、 $s \neq s'$ を満たすとき T を局所的に前方決定的であるという。また、 T の任意の異なる遷移規則 $m_1 = [p, \langle s, t \rangle, q]$ 、 $m_2 = [p', \langle s', t' \rangle, q'] \in \delta$ に対して $q = q'$ ならば、 $t \neq t'$ を

満たすとき T を局所的に後方決定的であるという。

遷移規則 δ について、本稿では 3 項組を用いるが、他研究の定義には 5 項組を用いている [5]。5 項組で設計を行った場合、ヘッドの移動とテープの記号の書き換えを 1 つの遷移規則で行うことができる為、少ない遷移規則で記述することが出来る。一方 3 項組は、遷移規則の数は 5 項組と比べてヘッドの移動とテープの記号書き換えが別々なため多くなってしまう。しかし、遷移規則に对称性があるため可逆チューリング機械を設計する際に、前方決定的または後方決定的であるということの判断がしやすいため、本稿では 3 項組を用いる。

したがって、可逆チューリング機械 $T' = (Q, \Sigma, b, \delta, q_s, q_f)$ は、局所的に前方決定的かつ後方決定的であり、最終状態からの遷移及び、初期状態への遷移がないものとする。ここで可逆チューリング機械は局所的に前方決定的であるため、最終状態は一つしかないので q_f のみとする。

3.2.2 可逆チューリング機械の例

これまでに定義した可逆チューリング機械に基づいて、簡単な可逆チューリング機械を考えてみる。

例 与えられた 2 進数の 0 と 1 を反転させる可逆チューリング機械 $T_2 = (Q_2, \{0, 1\}, b, \delta_2, q_s, q_f)$ を考える。ただし、 $Q_2 = \{q_s, q_1, q_2, q_3, q_4, q_f\}$ であり、 δ_2 は以下の 3 項組の集合である。

$$\begin{aligned} \delta_2 = \{ & [q_s, \langle b, b \rangle, q_1], [q_1, \rightarrow, q_2], \\ & [q_2, \langle 0, 0 \rangle, q_1], \\ & [q_2, \langle 1, 1 \rangle, q_1], \\ & [q_2, \langle b, b \rangle, q_3], [q_3, \leftarrow, q_4], \\ & [q_4, \langle 0, 1 \rangle, q_3], \\ & [q_4, \langle 1, 0 \rangle, q_3], \\ & [q_4, \langle b, b \rangle, q_f]\} \end{aligned}$$

T_2 は、2 進数表現が書かれたテープが与えられ、ヘッドを 2 進数表現の左隣のます目に置いて初期状態 q_s から動作を開始したとき、書かれている 2 進数表現を反転したものに書き換え、2 進数表現の左隣のます目までヘッドを移動し、最終状態 q_f で停止するチューリング機械である。このとき、 T_2 は 2 進数表現の 1 つ右隣のます目までヘッド移動する。その後、ヘッドを初期の位置に移動させながら、1 を読み込んだとき 0 に書き換える。また、0 を読み込んだとき 1 に書き換える。各状態における T_2 の動作を以下で説明する。

q_s : 2 進数表現 n の左隣の b を読み、 q_1 へ。

q_1 : ヘッドを右に 1 コマ移動し、 q_2 へ。

q_2 : b 以外を読んだとき、 q_1 へ。また、 b を読んだとき、 q_3 へ。

q_3 : ヘッドを左に 1 コマ移動し、 q_4 へ。

q_4 : 0 を読んだとき、1 に書き換える。また、1 を読み込んだ場合 0 に書き換え、 q_3 へ。そして、 b を読んだとき q_f へ。

T_2 において、 q_1 が 1 番目の項として現れている 3 項組は (i) $[q_2, \langle 0, 0 \rangle, q_1]$, (ii) $[q_2, \langle 1, 1 \rangle, q_1]$ または (iii) $[q_2, \langle b, b \rangle, q_3]$ の 3 つである。ある時刻において T_1 が状態 q_1 であり、現在読んでいるます目が 0 ならば (i)、1 ならば (ii) または b ならば (iii) が直後に実行されるということが一意に決まる。これと同様のことは状態 q_4 でも言えるため、 T_2 は局所的に前方決定的である。また、 q_1 が 3 番目の項として現れている 3 項組は (iv) $[q_s, \langle b, b \rangle, q_1]$, (v) $[q_2, \langle 0, 0 \rangle, q_1]$ または (vi) $[q_2, \langle 0, 0 \rangle, q_1]$ の 3 つである。ある時刻において T_1 が状態 q_1 であり、現在読んでいるます目が b ならば (iv)、0 ならば (v) または 1 ならば (vi) が直後に実行されるということが一意に決まる。これと同様のことは状態 q_3 でも言えるため、 T_2 は局所的に後方決定的である。そして T_1

は最終状態からの遷移および初期状態への遷移はない．以上より T_2 は可逆チューリング機械である．

また，この遷移規則による様相を 3.1.2 節の様相の計算ステップの記法を用いて表わす．予め，テープに格納されている 2 進数を「101」とする，

$t = 0$	$(q_s, (\lambda, b, 1))$	$\xRightarrow{T_{1-2}}$	$(q_1, (\lambda, b, 1))$
$t = 1$		$\xRightarrow{T_2}$	$(q_2, (b, 1, 0))$
$t = 2$		$\xRightarrow{T_2}$	$(q_1, (b, 0, 0))$
$t = 3$		$\xRightarrow{T_2}$	$(q_2, (0, 0, 1))$
$t = 4$		$\xRightarrow{T_2}$	$(q_3, (0, 1, 1))$
$t = 5$		$\xRightarrow{T_2}$	$(q_5, (1, 1, b))$
$t = 6$		$\xRightarrow{T_2}$	$(q_7, (1, 1, b))$
$t = 7$		$\xRightarrow{T_2}$	$(q_8, (0, 1, 1))$
$t = 8$		$\xRightarrow{T_2}$	$(q_9, (0, 1, 1))$
$t = 9$		$\xRightarrow{T_2}$	$(q_{10}, (b, 0, 1))$
$t = 10$		$\xRightarrow{T_2}$	$(q_9, (b, 0, 1))$
$t = 11$		$\xRightarrow{T_2}$	$(q_{10}, (\lambda, b, 0))$
$t = 12$		$\xRightarrow{T_2}$	$(q_f, (\lambda, b, 0))$

3.2.3 可逆シミュレーションの例

ここでは非可逆なチューリング機械の遷移規則を工夫することで，可逆チューリング機械に作り変えること（可逆シミュレーション）が出来るということを実際に例を用いて説明する．例には 3.1.3 小節で説明した T_1 を用いる． T_1 の遷移規則 δ_1 は以下の 3 項組の集合であった．

$$\begin{aligned} \delta_1 = \{ & [q_s, \langle b, b \rangle, q_1], [q_1, \rightarrow, q_2], \\ & [q_2, \langle 1, 0 \rangle, q_1], \\ & [q_2, \langle 0, 1 \rangle, q_3], [q_3, \leftarrow, q_4], \\ & [q_2, \langle b, 1 \rangle, q_3], \\ & [q_4, \langle 0, 0 \rangle, q_3], \\ & [q_4, \langle 1, 1 \rangle, q_3], \\ & [q_4, \langle b, b \rangle, q_f] \} \end{aligned}$$

T_1 は次の理由により非可逆なチューリング機械である． q_3 が 3 番目の項として現れている 3 項組は (i) $[q_2, \langle 0, 1 \rangle, q_3]$, (ii) $[q_2, \langle b, 1 \rangle, q_3]$, (iii) $[q_4, \langle 0, 0 \rangle, q_3]$ または (iv) $[q_4, \langle 1, 1 \rangle, q_3]$ の 4 つである．そのうち (i),(ii) または (iv) の 3 つの遷移規則は書き換えた後の記号がどれも 1 である．つまり，ある時刻において T_1 が状態 q_3 にあり，現在読んでいるマス目が 1 ならば，(i),(ii) または (iv) のどれが直前の様相かが一意に決まらない．すなわち，可逆チューリング機械の条件である局所的に後方決定的であるという条件を満たさないため， T_1 は非可逆なチューリング機械である．

この非可逆なチューリング機械 T_1 の遷移規則を可逆になるように設計した可逆チューリング機械 $T_{1-2} = (Q_3, \{b, 0, 1\}, b, \delta_1, q_s, q_f)$ とする．ただし， $Q_3 = \{q_s, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_f\}$ であこのとき，遷

移規則 δ_{1-2} 以下のように表す．

$$\begin{aligned}\delta_{1-2} = \{ & [q_s, \langle b, b \rangle, q_1], [q_1, \rightarrow, q_2], \\ & [q_2, \langle 1, 0 \rangle, q_1], \\ & [q_2, \langle 0, 1 \rangle, q_3], [q_3, \rightarrow, q_5], \\ & [q_5, \langle 0, 0 \rangle, q_7], \\ & [q_5, \langle 1, 1 \rangle, q_7], \\ & [q_2, \langle b, 1 \rangle, q_4], [q_4, \rightarrow, q_6], \\ & [q_6, \langle b, b \rangle, q_7], [q_7, \leftarrow, q_8], \\ & [q_8, \langle 1, 1 \rangle, q_9], [q_9, \leftarrow, q_{10}], \\ & [q_{10}, \langle 0, 0 \rangle, q_9], \\ & [q_{10}, \langle b, b \rangle, q_f]\}\end{aligned}$$

T_1 は 1 を加えた直後の状態から直前に 1 を加えることで 2 進数表現に桁上げが行われたのかを判断することが出来なかった．そこで T_{1-2} では 1 を加える動作の後にヘッドを 2 進数表現の 1 つ左まで移動するのではなく，さらに右に 1 つヘッドを移動し，そのます目に書かれる記号が空白記号であるかそうでないかを確認する動作を追加した．そうすることで 2 進数表現に 1 が加えられたとき，桁上げが行われていたかを判断することが出来るようになる．

3.2.2 小節では単純な可逆チューリング機械について説明したが，この例の様に非可逆なチューリング機械は単純に設計されていても，可逆シミュレーションすることで単純であった遷移規則が複雑になってしまうことがある．

3.3 3 テープ可逆チューリング機械

3.2.3 小節では，1 テープの非可逆チューリング機械を 1 テープの可逆チューリング機械にする可逆シミュレーションを行ったが，本節では非可逆なチューリング機械を 3 本のテープを持つチューリング機械を用いて可逆シミュレーションを行う．3 テープ可逆チューリング機械 T'' を 3.2.1 小節の定義より $T'' = (Q, \{\Sigma_1, \Sigma_2, \Sigma_3\}, b, \delta, q_s, q_f)$ と定める．非可逆なチューリング機械を T として， T'' の 1 本目のテープは T のテープの内容をそのまま表現し，ヘッドの位置も同じとする．2 本目のテープは各計算ステップで T のどの 3 項組で遷移されたのかを記録する．3 本目のテープは 1 本目のテープの記述をコピーし，結果を記録する． $\Sigma_1, \Sigma_2, \Sigma_3$ は 3 本のテープそれぞれで使用される記号の有限集合である．

δ は 3 項組集合であり，各項は $[p, \langle s, t \rangle, q]$ または $[p, d, q]$ の形であった．しかし，1 テープのチューリング機械の遷移規則である s, t, d がそれぞれ 3 テープ分必要になるため，3 テープチューリング機械では $[p, \langle [s_1, s_2, s_3], [t_1, t_2, t_3] \rangle, q]$ または $[p, [d_1, d_2, d_3], q]$ で表す．前者の 3 項組は T'' が状態 p で，3 つのヘッドの記号 s_1, s_2, s_3 を読んだ場合，記号 t_1, t_2, t_3 に書き換え，状態を q にすることを意味する．後者の 3 項組は T'' が状態 q の場合，ヘッドを d_1, d_2, d_3 の方向に動かし，状態 q にすることを意味する．

また，遷移規則は局所的に前方決定的，後方決定的も満たす．すなわち任意の異なる任意の 3 項組 $n_1 = [p, \langle [s_1, s_2, s_3], [t_1, t_2, t_3] \rangle, q], n_2 = [p', \langle [s'_1, s'_2, s'_3], [t'_1, t'_2, t'_3] \rangle, q'] \in \delta$ に対して $p = p'$ ならば $[s_1, s_2, s_3] \neq [s'_1, s'_2, s'_3]$ ， $q = q'$ ならば $[t_1, t_2, t_3] \neq [t'_1, t'_2, t'_3]$ という関係が成り立つ．

3.3.1 可逆化 (Bennett 法)

3 テープ可逆チューリング機械の動作の過程は，大きく以下の 3 つに分割される．

1. 「2 本目のテープに動作履歴を残しながら計算を実行する過程」
2. 「計算結果を 3 本目のテープにコピーする過程」
3. 「動作履歴を可逆的に消去しながら逆の動作を実行する過程」

このようにして動作を実行することで、最終的に 1 本目のテープに入力が残り、3 本目のテープに答えが出力され、不必要な情報は綺麗に消去された状態で停止することが出来る。

3.3.2 Bennett 法による可逆シミュレーションの例

3 テープを用いての可逆シミュレーションの例には 3.1.3 小節で説明した T_1 を用いる。

チューリング機械 T_1 を可逆シミュレーションする 3 テープ可逆チューリング機械 T_3 は、

$$T_3 = (\{q_s, q_1, q_2, q_3, q_4, q_f, c_b, c_1, c_2, c_3, c_4, c_5, p_s, p_1, p_2, p_3, p_4, p_f\}, \{\{b, 0, 1\}, \{0, 1, 2, 3, 4, 5, 6, 7\}, \{b, 0, 1\}\}, b, \delta_3, q_s, \{p_s\})$$

遷移規則 δ_3 は、

$$\begin{aligned} \delta_3 = \{ & [q_s, \langle [b, b, b], [b, 1, b] \rangle, q_1], [q_1, [\quad, \quad, \quad], q_2], \\ & [q_2, \langle [1, b, b], [0, 2, b] \rangle, q_1], [q_3, [\quad, \quad, \quad], q_4], \\ & [q_2, \langle [0, b, b], [1, 3, b] \rangle, q_3], \\ & [q_2, \langle [b, b, b], [1, 4, b] \rangle, q_3], \\ & [q_4, \langle [0, b, b], [0, 5, b] \rangle, q_3], \\ & [q_4, \langle [1, b, b], [1, 6, b] \rangle, q_3], \\ & [q_4, \langle [b, b, b], [b, 7, b] \rangle, q_f], [q_f, [\quad, \quad, \quad], c_0], \\ & [c_0, \langle [b, b, b], [b, b, b] \rangle, c_1], [c_1, [\quad, \quad, \quad], c_2], \\ & [c_2, \langle [1, b, b], [1, b, 1] \rangle, c_1], \\ & [c_2, \langle [0, b, b], [0, b, 0] \rangle, c_1], \\ & [c_2, \langle [b, b, b], [b, b, b] \rangle, c_3], [c_3, [\quad, \quad, \quad], c_4], \\ & [c_4, \langle [1, b, 1], [1, b, 1] \rangle, c_3], \\ & [c_4, \langle [0, b, 0], [0, b, 0] \rangle, c_3], \\ & [c_4, \langle [b, b, b], [b, b, b] \rangle, c_5], [c_5, [\quad, \quad, \quad], p_f], \\ & [p_f, \langle [b, 7, b], [b, b, b] \rangle, p_4], \\ & [p_3, \langle [1, 6, b], [1, b, b] \rangle, p_4], \\ & [p_3, \langle [0, 5, b], [0, b, b] \rangle, p_4], \\ & [p_3, \langle [1, 4, b], [b, b, b] \rangle, p_2], \\ & [p_3, \langle [1, 3, b], [0, b, b] \rangle, p_2], \\ & [p_1, \langle [0, 2, b], [1, b, b] \rangle, p_2], [p_4, [\quad, \quad, \quad], p_3], \\ & [p_1, \langle [b, 1, b], [b, b, b] \rangle, p_s], [p_2, [\quad, \quad, \quad], p_1] \} \end{aligned}$$

上記の 3 テープ可逆チューリング機械の動作を順に解説する。まず 1 つ目の過程であるが、 δ_3 の最初の 9 個の 3 項組は δ_1 の 9 個の 3 項組に対応している。これらは 1 本目のテープ上で T_1 と同じ遷移するだけでなく、2 本目のテープに δ_1 の 1 から 7 のどの 3 項組が使われかを示す 1 から 7 の数字を書き込むことで、可逆チューリング機械の条件を満たしながら計算を実行することが出来る。2 つ目の過程では、 T_3 は、 T_1 の最終状態である q_f に遷移したとき、 c_s に移し、その後 $c_i (i = 0, 1, \dots, 5)$ を使って 1 本目のテープに記された記号を 3 本目のテープに記録する。記録後に不必要情報を消去するため状態を p_f に遷移する。3 つ目の過程では、 δ_3 の最後の 9 個の 3 項組は最初の 9 個の 3 項組の逆から、つまり q_f から q_s までの動作を行う。そのため p と q はそれぞれ対応していて、最終的には状態 p_s で停止する。

また、この遷移規則による様相を 3.1.2 節の様相の計算ステップの記法を用いて表わす。

$t = 0$	$(q_s, ((\lambda, b, 1), (\lambda, b, \lambda), (\lambda, b, \lambda)))$	\Rightarrow_{T_3}	$(q_1, ((\lambda, b, 1), (\lambda, 1, \lambda), (\lambda, b, \lambda)))$
$t = 1$		\Rightarrow_{T_3}	$(q_2, ((b, 1, 0), (1, b, \lambda), (\lambda, b, \lambda)))$
$t = 2$		\Rightarrow_{T_3}	$(q_1, ((b, 0, 0), (1, 2, \lambda), (\lambda, b, \lambda)))$
$t = 3$		\Rightarrow_{T_3}	$(q_2, ((0, 0, 1), (2, b, \lambda), (\lambda, b, \lambda)))$
$t = 4$		\Rightarrow_{T_3}	$(q_3, ((0, 1, 1), (2, 3, \lambda), (\lambda, b, \lambda)))$
$t = 5$		\Rightarrow_{T_3}	$(q_4, ((b, 0, 1), (3, b, \lambda), (\lambda, b, \lambda)))$
$t = 6$		\Rightarrow_{T_3}	$(q_3, ((b, 0, 1), (3, 5, \lambda), (\lambda, b, \lambda)))$
$t = 7$		\Rightarrow_{T_3}	$(q_4, ((\lambda, b, 0), (5, b, \lambda), (\lambda, b, \lambda)))$
$t = 8$		\Rightarrow_{T_3}	$(q_f, ((\lambda, b, 0), (5, 7, \lambda), (\lambda, b, \lambda)))$
$t = 9$		\Rightarrow_{T_3}	$(c_0, ((\lambda, b, 0), (7, b, \lambda), (\lambda, b, \lambda)))$
$t = 10$		\Rightarrow_{T_3}	$(c_1, ((\lambda, b, 0), (7, b, \lambda), (\lambda, b, \lambda)))$
$t = 11$		\Rightarrow_{T_3}	$(c_2, ((b, 0, 1), (7, b, \lambda), (b, b, \lambda)))$
$t = 12$		\Rightarrow_{T_3}	$(c_1, ((b, 0, 1), (7, b, \lambda), (b, 0, \lambda)))$
$t = 13$		\Rightarrow_{T_3}	$(c_2, ((0, 0, 1), (7, b, \lambda), (0, b, \lambda)))$
$t = 14$		\Rightarrow_{T_3}	$(c_1, ((0, 1, 1), (7, b, \lambda), (0, 1, \lambda)))$
$t = 15$		\Rightarrow_{T_3}	$(c_2, ((1, 1, b), (7, b, \lambda), (1, b, \lambda)))$
$t = 16$		\Rightarrow_{T_3}	$(c_1, ((1, 1, b), (7, b, \lambda), (1, 1, \lambda)))$
$t = 17$		\Rightarrow_{T_3}	$(c_2, ((1, b, \lambda), (7, b, \lambda), (1, b, \lambda)))$
$t = 18$		\Rightarrow_{T_3}	$(c_3, ((1, b, \lambda), (7, b, \lambda), (1, b, \lambda)))$
$t = 19$		\Rightarrow_{T_3}	$(c_4, ((1, 1, b), (7, b, \lambda), (1, 1, b)))$
$t = 20$		\Rightarrow_{T_3}	$(c_3, ((1, 1, b), (7, b, \lambda), (1, 1, b)))$
$t = 21$		\Rightarrow_{T_3}	$(c_4, ((0, 1, 1), (7, b, \lambda), (0, 1, 1)))$
$t = 21$		\Rightarrow_{T_3}	$(c_3, ((0, 1, 1), (7, b, \lambda), (0, 1, 1)))$
$t = 22$		\Rightarrow_{T_3}	$(c_4, ((b, 0, 1), (7, b, \lambda), (b, 0, 1)))$
$t = 23$		\Rightarrow_{T_3}	$(c_3, ((b, 0, 1), (7, b, \lambda), (b, 0, 1)))$
$t = 24$		\Rightarrow_{T_3}	$(c_4, ((\lambda, b, 0), (7, b, \lambda), (\lambda, b, 0)))$
$t = 25$		\Rightarrow_{T_3}	$(c_5, ((\lambda, b, 0), (7, b, \lambda), (\lambda, b, 0)))$
$t = 26$		\Rightarrow_{T_3}	$(p_f, ((\lambda, b, 0), (5, 7, b), (\lambda, b, 0)))$
$t = 27$		\Rightarrow_{T_3}	$(p_4, ((\lambda, b, 0), (5, b, \lambda), (\lambda, b, 0)))$
$t = 28$		\Rightarrow_{T_3}	$(p_3, ((b, 0, 1), (3, 5, b), (\lambda, b, 0)))$
$t = 29$		\Rightarrow_{T_3}	$(p_4, ((b, 0, 1), (3, b, \lambda), (\lambda, b, 0)))$
$t = 30$		\Rightarrow_{T_3}	$(p_3, ((0, 1, 1), (2, 3, b), (\lambda, b, 0)))$
$t = 31$		\Rightarrow_{T_3}	$(p_2, ((0, 0, 1), (2, b, \lambda), (\lambda, b, 0)))$
$t = 32$		\Rightarrow_{T_3}	$(p_1, ((b, 0, 0), (1, 2, b), (\lambda, b, 0)))$
$t = 33$		\Rightarrow_{T_3}	$(p_2, ((b, 1, 0), (1, b, \lambda), (\lambda, b, 0)))$
$t = 34$		\Rightarrow_{T_3}	$(p_1, ((\lambda, b, 1), (\lambda, 1, \lambda), (\lambda, b, 0)))$
$t = 35$		\Rightarrow_{T_3}	$(p_s, ((\lambda, b, 1), (\lambda, b, \lambda), (\lambda, b, 0)))$

第 4 章

可逆チューリング機械から R-WHILE への変換

この章では命令型可逆プログラミング言語 R-WHILE (以下, 単に R-WHILE と呼ぶことにする) を万能可逆チューリング機械に変換する規則を与えることで, R-WHILE が万能可逆チューリング機械を実装可能であることを示す。

4.1 R-WHILE について

ここでは R-WHILE[7] について説明する。

R-WHILE は, Jones の言語 WHILE を可逆化したものである。R-WHILE は非可逆なプログラムを記述することができない。そのため単射関数しか表すことはできない。この言語の特徴は木構造のデータを表現できることである。それにより単純な方法で身近なデータ構造をモデリングが可能である, このことは, 既存の可逆命令言語 Janus では難しいことである。

4.1.1 プログラム例: リスト逆転

与えられたリストを逆転させて表示するプログラム例 reverse を用いていくつかの言語の機能について説明する。

```
read X; (*リスト X を逆転させるプログラム reverse*)
  from (=? Y nil)
  loop (Z.X) <= X;
    Y<= (Z.Y)
  until(=? X nil);
write Y
```

プログラムの入力の変数 X を読み込む。そして, 出力は変数 Y を書き出す。すべての変数の初期値は nil である。可逆ループを行う命令, loop...until は Y が nil であると主張されたとき, 繰り返しが行われ, X が nil のときに繰り返しを終了する。loop 内の命令はテストとアサーションが偽である限り繰り返す。loop 内の最初の命令 ((Z.X) <= X) では X の値を先頭とそれ以降の部分に分解をしている。2 番目の命令 (Y <= (Z.X)) では, Z と Y をペアにした値を構成している。可逆置換は右辺の元の値を左辺の変数に束縛する前に, 右辺の値を nil にセットする (たとえば Y <= (Z.Y) が実行された後 Z は nil である)。出力の変数である Y を除くすべての変数は最終的に nil でなくてはならない。プログラム reverse の適用は $\llbracket \text{reverse} \rrbracket^{\text{R-WHILE}}$ ('a.('b.('c.nil)))=('c.('b('a.nil))) の様に示す。ここで, リストの要素 'a, 'b または 'c はアトムである。エントリーアサーションと可逆置換により, プログラムは可逆である。

$E, F ::= X \mid d \mid \text{cons } E F \mid \text{hd } E \mid \text{tl } E \mid =? E F$	式
$Q, R ::= X \mid d \mid \text{cons } Q R$	パターン
$C, D ::= X \hat{=} E$	命令
$\mid Q \leq R$	
$\mid C; D$	
$\mid \text{if } E \text{ then } C \text{ else } D \text{ fi } F$	
$\mid \text{from } E \text{ do } C \text{ loop } D \text{ until } F$	
$P ::= \text{read } X; C; \text{write } Y$	R-WHILE のプログラム

図 4.1: 言語 R-WHILE の構文規則

4.1.2 構文

R-WHILE の構文規則は単純である (図 4.1). プログラム P はただ一つの入口と出口の点 ($\text{read}, \text{write}$) をもち, 命令 C がプログラムの本体である.

プログラムのデータ領域 \mathbb{D} はアトム nil とすべての組 (d_1, d_2) を含む ($d_1, d_2 \in \mathbb{D}$) 最小の集合である. Var は変数名の無限集合である. 本稿では $d, e, f, \dots \in \mathbb{D}$ とする. また, $X, Y, Z, \dots \in Vars$ とする.

式は変数 X , 定数 d , または複数の演算子からなる (先頭とそれ以降を表す hd と tail , 組を表す cons また等号を表す $=?$) からなる. パターンは式の部分集合であり, 変数 X , 定数 d またはパターンのペアを表す $\text{cons } Q R$ からなる. 本稿では以後ペアを表す $\text{cons } E F$ または $\text{cons } Q R$ をそれぞれ $(E.F)$ または $(Q.R)$ と表記する. パターンは線形的でなくてはならない. すなわち, 反復変数は含まれない. また, この言語は局所変数をもたない.

可逆代入 $X \hat{=} E$ において, 左辺の変数 X に右辺の式 E があらわれてはならない. 可逆置換 $Q \leq R$ は Q の変数を R の変数を使って更新する. 可逆代入と比べ両辺に表されるのはパターンの Q と R である.

R-WHILE の 2 つの構造化された制御フロー演算子は条件文の $\text{if } E \text{ then } C \text{ else } D \text{ fi}$ と繰り返し文の $\text{from } E \text{ do } C \text{ loop } D \text{ until } F$ である. 繰り返し文はテスト E をもち, 条件文はアサーション F をもつ.

可逆条件文 $\text{if } E \text{ then } C \text{ else } D \text{ fi } F$ の制御フローの分岐はテスト E に依存する. もし真であれば命令 C が実行され, アサーション E は真でなくてはならない. また, もし偽であった場合命令 D が実行され, アサーション E は偽でなくてはならない. E と F の返す値が対応していない場合, 条件文は定義されない. 可逆ループ $\text{from } E \text{ do } C \text{ loop } D \text{ until } F$ は図??の様に描くことができる. ループを行うとき, アサーション E は真でなくてはならない (図??の t). そして, 命令 C が実行される. 実行後のテスト F が真であれば繰り返しは続行される. もし F が偽であった場合, 命令 D が実行される. また, アサーション E は偽でなくてはならない (図??の f).

私たちは $(d_1.(d_2.(\dots (d_n.\text{nil}) \dots)))$ をリスト $(d_1 d_2 \dots d_n)$ と書き, $(d_1.(d_2.(\dots (d_{n-1}.d_n) \dots)))$ を不適切なリスト $(d_1 \dots d_{n-1} d_n)$ と書く.

リストと似た表記として, 私たちは $(Q_1.(Q_2.(\dots (Q_n.\text{nil}) \dots)))$ をリスト $(Q_1 Q_2 \dots Q_n)$ と書き, $(Q_1.(Q_2.(\dots (Q_{n-1}.Q_n) \dots)))$ を不適切なリスト $(Q_1 \dots Q_{n-1} Q_n)$ と書く.

式 $(\text{list } E_1 \dots E_n)$ は式 $(\text{cons } E_1 \dots (\text{cons } E_n \text{ nil}) \dots)$ の省略法として用いられる. また, 式 $(\text{cons} * E_1 \dots E_{n-1} E_n)$ は式 $(\text{cons } E_1 \dots (\text{cons } E_{n-1} E_n) \dots)$ として用いられる.

$$\begin{aligned}
\mathcal{E}[\mathbf{d}]\sigma &= d & \mathcal{E}[\mathbf{cons}\ E\ F]\sigma &= (\mathcal{E}[\mathbf{E}]\sigma, \mathcal{E}[\mathbf{F}]\sigma) \\
\mathcal{E}[\mathbf{X}]\sigma &= \sigma(\mathbf{X}) & \mathcal{E}[\mathbf{=?}\ E\ F]\sigma &= \begin{cases} \mathbf{true} & \text{if } \mathcal{E}[\mathbf{E}]\sigma = \mathcal{E}[\mathbf{F}]\sigma \\ \mathbf{false} & \text{otherwise} \end{cases} \\
\mathcal{E}[\mathbf{hd}\ E]\sigma &= e \text{ if } \mathcal{E}[\mathbf{E}]\sigma = (e.f) \\
\mathcal{E}[\mathbf{tl}\ E]\sigma &= f \text{ if } \mathcal{E}[\mathbf{E}]\sigma = (e.f) \\
\\
\mathcal{Q}[\mathbf{d}]\sigma &= (d, \sigma) \\
\mathcal{Q}[\mathbf{X}](\sigma \uplus \{\mathbf{X} \mapsto d\}) &= (d, \sigma \uplus \{\mathbf{X} \mapsto \mathbf{nil}\}) \\
\mathcal{Q}[\mathbf{cons}\ Q\ R]\sigma &= ((d_1.d_2), \sigma_2) \text{ where } (d_1, \sigma_1) = \mathcal{Q}[Q]\sigma \wedge (d_2, \sigma_2) = \mathcal{Q}[R]\sigma_1 \\
\\
\mathcal{C}[\mathbf{X} \leftarrow \mathbf{E}](\sigma \uplus \{\mathbf{X} \mapsto d\}) &= \sigma \uplus \{\mathbf{X} \mapsto d \odot \mathcal{E}[\mathbf{E}]\sigma\} \\
\mathcal{C}[\mathbf{Q} \leq \mathbf{R}] &= \mathcal{Q}[\mathbf{Q}]^{-1}(\mathcal{Q}[\mathbf{R}]\sigma) \\
\mathcal{C}[\mathbf{C}; \mathbf{D}]\sigma &= \mathcal{C}[\mathbf{D}](\mathcal{C}[\mathbf{C}]\sigma) \\
\mathcal{C}[\mathbf{if}\ E\ \mathbf{then}\ C\ \mathbf{else}\ D\ \mathbf{fi}\ F] &= \begin{cases} \sigma' & \text{if } \mathcal{E}[\mathbf{E}]\sigma = \mathbf{true} \wedge \sigma' = \mathcal{C}[\mathbf{C}]\sigma \wedge \mathcal{E}[\mathbf{F}]\sigma' = \mathbf{true} \\ \sigma' & \text{if } \mathcal{E}[\mathbf{E}]\sigma = \mathbf{false} \wedge \sigma' = \mathcal{C}[\mathbf{D}]\sigma \wedge \mathcal{E}[\mathbf{F}]\sigma' = \mathbf{false} \end{cases} \\
\mathcal{C}[\mathbf{from}\ E\ \mathbf{do}\ C\ \mathbf{loop}\ D\ \mathbf{until}\ F] &= \sigma' \text{ if } \mathcal{E}[\mathbf{E}] = \mathbf{true} \wedge \sigma' = \mathbf{fix}(F)(\sigma) \\
&\quad \text{where } F(\varphi) = \begin{aligned} &\{(\sigma, \sigma_1) \mid \sigma_1 = \mathcal{C}[\mathbf{C}]\sigma \wedge \mathcal{E}[\mathbf{F}]\sigma_1 = \mathbf{true}\} \cup \\ &\{(\sigma, \sigma_3) \mid \sigma_1 = \mathcal{C}[\mathbf{C}]\sigma \wedge \mathcal{E}[\mathbf{F}]\sigma_1 = \mathbf{false} \wedge \\ &\quad \sigma_2 = \mathcal{C}[\mathbf{D}]\sigma_1 \wedge \mathcal{E}[\mathbf{E}]\sigma_2 = \mathbf{false} \wedge \\ &\quad \sigma_3 = \varphi(\sigma_2)\} \end{aligned} \\
\\
\mathcal{P}[\mathbf{P}]D = D' &\text{ if } \mathbf{P} \text{ is read } \mathbf{X}; \mathbf{C}; \mathbf{write}\ \mathbf{Y} \wedge \mathcal{C}[\mathbf{C}](\sigma_X^P(D)) = \sigma_Y^P(D')
\end{aligned}$$

図 4.2: R-WHILE の表示的意味

4.1.3 意味論

ここでは R-WHILE のための表示的意味論を定義する．まず，いくつかの表記法を始め，図 4.2 の 4 つの意味関数について説明する．

プログラム P の中にある変数の組と式 E はそれぞれ $Var(P)$, $Var(E)$ と示される．ブール値は明確な二つの要素 ($\mathbf{false}, \mathbf{true} \in \mathbb{D}$) によって示される．未定義の値は \perp によって示される．そして，組 X または $\{\perp\}$ は X_\perp によって示される．私たちは値の範囲として \mathbb{D}_\perp を用いる．

プログラム P のための状態 σ は $Var(P)$ から \mathbb{D}_\perp への関数である． P のためのすべての状態の組は $Stores^P$ によって示される．状態 $\sigma \setminus X$ は変数 X を \perp に写すことを除けば σ と同一である．状態 $\sigma \uplus \sigma'$ は状態 σ と σ' の束縛するものの非交和であり，それらは共通の変数をもたない．未定義の状態すなわち，どのような変数の対しても \perp をかえすものは \perp とも書かれる．

状態 σ_X^P は変数 X を D に束縛する．また， $Var(P) = \{X, Y_1, \dots, Y_n\}$ 内の他のすべての変数を \mathbf{nil} に束縛する．すなわち，

$$\sigma_X^P(D) = [X \mapsto D, Y_1 \mapsto \mathbf{nil}, \dots, Y_n \mapsto \mathbf{nil}]$$

となる．

意味関数は以下の様に式 (Expressions), パターン (Patterns), 命令 (Commands) またはプログラム (Programs) の四種類によって定義される (図 4.2)．このとき， A から B への単射の写像を $A \hookrightarrow B$ によって示す．

$$\mathcal{E} : \text{Expressions} \rightarrow (\text{Stores}^P \hookrightarrow \mathbb{D}_\perp)$$

$$\mathcal{Q} : \text{Patterns} \rightarrow (\text{Stores}^P \hookrightarrow \mathbb{D}_\perp \times \text{Stores}_\perp^P)$$

$$\mathcal{C} : \text{commands} \rightarrow (\text{Stores}^P \hookrightarrow \text{Stores}_\perp^P)$$

$$\mathcal{P} : \text{Programs} \rightarrow (\mathbb{D} \hookrightarrow \mathbb{D}_\perp)$$

プログラム p は部分関数 $\mathcal{P}[p]$ を示す．本稿では p が R-WHILE のプログラムであることを明示するために $\llbracket p \rrbracket^{\text{R-WHILE}}$

式評価 \mathcal{E} は状態 σ の式 E を \mathbb{D} の値へ評価する．変数 X ，定数 d ，または複数の演算子からなる（先頭とそれ以降を表す hd と $tail$ ，組を表す $cons$ また等号を表す $=?$ ）の評価は予想されるとおりである．また \mathcal{E} は \perp へ評価することができる．もし E の値が nil だった場合 $hd\ E$ と $tl\ E$ の値は定義されないからである．

パターン評価 Q は状態 σ のパターン Q を値と状態の組へ評価する． Q において使われるすべての変数は結果の状態で nil に縛られる．これは，結果の状態を構築するすべての値を状態から移動する．いくつかの σ のために $Q[Q]\ \sigma = (\mathcal{E}[Q], \sigma')$ とする．任意の Q において，表示 $Q[Q]$ は単射関数である．そのため，逆関数 $Q[Q]^{-1}$ はただ一つである．逆パターン評価 $Q[Q]^{-1}(d, \sigma')$ は値と状態の組から Q のすべての変数の新しい状態をとる．またこれは，定数 d の対応する部分に縛られおり，与えられるそれらの変数は状態 σ において nil に縛られる．もし Q が Q と σ について定義されている場合， $Q[Q]^{-1}(Q[Q]\sigma) = \sigma$ が成り立つ．パターン評価とその逆は可逆置換の意味を定義するために用いられる．

ここからは命令評価 \mathcal{C} について述べる．非可逆な言語における代入は破壊的である．それらは左辺の変数の値を上書きする．そして，代入を行った後，元の値を再び構築することはできない．そのため，可逆プログラミング言語では言語では基本的に用いることができない．そのかわりに，可逆代入 $X \hat{=} E$ を用いる．これは，もし X の値が nil であるとき， X に E の値にし， X の値が E の値と等しいとき， X の値を nil にするというものである．前者の場合 E は複製される．また，後者の場合 X と E の等しさが主張される．以下で，更新演算子 \odot を定義することによって意味論を正式なものにする． \odot は

$$d \odot e = \begin{cases} e & \text{if } d = nil \\ nil & \text{if } d = e \neq nil \end{cases}$$

と定義する．こうすることで，可逆代入を以下の様に定義することができ，

$$\mathcal{C}[X \hat{=} E](\sigma \uplus [X \mapsto d']) = \sigma \uplus [X \mapsto d']$$

となる．このとき， $d' = d \odot \mathcal{E}[E]\sigma$ である． E が評価される σ において，左辺の X に対する束縛を含まない．これは直和 $\sigma \uplus [X \mapsto d]$ であることから導かれる．それゆえ，もし E に X が現れた場合，代入は定義されない．そうでない場合，無効な代入 $X \hat{=} X$ は X がどのような値であっても X に nil をセットする．そして， $\mathcal{C}[X \hat{=} X]$ は単射ではなくなってしまう．

可逆置換 $Q \Leftarrow R$ は Q と R の両辺を置き換えることによって変数を更新する． $\mathcal{C}[Q \Leftarrow R]\ \sigma$ は最初的评价 $Q[R]\ \sigma$ によって定義される．このとき， $Q[R]\ \sigma$ は (d, σ') である．また， $Var(R)$ のすべての変数が nil である場合を除き， d は $\mathcal{E}[R]\ \sigma$ であり，状態 σ' は σ である．そしてこれに逆評価の $Q[Q]^{-1}(d, \sigma')$ が実行され，可逆置換の最終状態である σ'' を得る．このとき $Var(Q)$ のすべての変数は状態 σ において nil にならなくてはならない．逆評価の $Q[Q]^{-1}(d, \sigma')$ は $d = \mathcal{E}[Q]\sigma''$ のような d の一部に束縛される．

可逆条件文 $\text{if } E \text{ then } C \text{ else } D \text{ fi } F$ において，制御フローの分岐はテスト E に依存する．テスト E が $true$ であった場合，命令 C が実行される．そして，アサーション F は必ず $true$ でなくてはならない．テスト E が $false$ であった場合，命令 D が実行される．そして，アサーション F は $false$ でなくてはならない．もしアサーション F の値とテストの E の値が対応していない場合，条件分岐は定義されない．アサーション F は条件分岐の実行が後方決定的であることを主張する．

可逆的繰り返し $\text{from } E \text{ do } C \text{ loop } D \text{ until } F$ はフローチャートの様に図式化することができる（図 4.3）．意味論は以下のとおりである．繰り返しの入口において，アサーション E は $true$ でなくてはならない（図 4.3 の t の部分）．その後，命令 F は実行される．そのとき，テスト F が $true$ である場合，繰り返しは終了する．もし F が $false$ であった場合，命令 D が実行される．そして，アサーション E は $false$ でなくてはならない（図 4.3 の f の部分）．もし E の値が要求される値とことなる場合，繰り返しは定義されない．

4.1.4 プログラム逆変換器

プログラミング逆変換器は従来の言語では一般的に利用することができない．しかし可逆プログラミング言語である R-WHILE にはプログラム逆変換器 \mathcal{I} が定義されている（図 4.4）．それによって反転されたプログラ

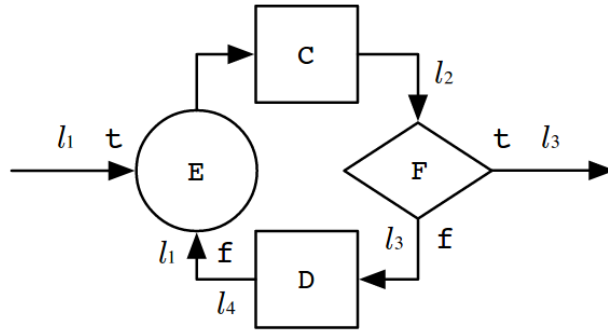


図 4.3: while loop のフローチャート

$\mathcal{I}[\text{X} \hat{=} \text{E}]$	$= \text{X} \hat{=} \text{E}$
$\mathcal{I}[\text{Q} \leq \text{R}]$	$= \text{Q} \leq \text{R}$
$\mathcal{I}[\text{C}; \text{D}]$	$= \mathcal{I}[\text{D}]; \mathcal{I}[\text{C}]$
$\mathcal{I}[\text{if E then C else D fi F}]$	$= \text{if F then } \mathcal{I}[\text{C}] \text{ else } \mathcal{I}[\text{D}] \text{ fi E}$
$\mathcal{I}[\text{from E do C loop D until F}]$	$= \text{from F do } \mathcal{I}[\text{C}] \text{ loop } \mathcal{I}[\text{D}] \text{ until E}$
$\mathcal{I}[\text{read X; C; write Y}]$	$= \text{read Y; } \mathcal{I}[\text{C}]; \text{ write X}$

図 4.4: R-WHILE の逆変換器 \mathcal{I}

ムを再帰的降下によって得ることができる．可逆的代入は自己逆の為変化はない．可逆的置き換えは両辺を入れ替えるだけである．連続して命令を実行する命令の逆はそれぞれの逆命令の順序を逆にされたものである．テストとアサーションは制御フロー演算子ではそれぞれの役割を交代する．そして，プログラムは入力と出力の変数と入れ替え，プログラムの本体を逆変換することで逆変換されたプログラムを求めることができる．

例えば，図 4.5はリストを反転させる R-WHILE のプログラム `reverse` と逆変換器 \mathcal{I} によって求められた `reverse` のプログラム $\mathcal{I}[\text{reverse}]$ である．逆変換されたプログラムは元のプログラムと同一の動作をする． $\mathcal{I}[\text{reverse}]$ は X と Y の位置が入れ替わっているを除き，`reverse` と同じプログラムの構造をしていることがわかる．

プログラム P は二回逆変換を行うことで元のプログラムが求められる ($\text{P} = \mathcal{I}[\mathcal{I}[\text{P}]]$) ．

4.1.5 糖類構文としての書き換え規則

プログラムは糖類構文としての可逆版の書き換え規則をもちいることで読みやすくなる．糖類構文によって，複数の変数 $[\text{X}_1, \dots, \text{X}_n]$ は複数の規則 $\text{Rule}_1 \dots \text{Rule}_n$ によって書き換えられる．規則 Rule_j は

1. $[\text{Q}_1, \dots, \text{Q}_n] \Rightarrow [\text{E}_1, \dots, \text{E}_n]$ または，
2. $[\text{Q}_1, \dots, \text{Q}_n] \Rightarrow \text{C};$

のいずれかである．それぞれの意味は次のようである．もし $\text{X}_1, \dots, \text{X}_n$ の値が連続して試される規則 Rule_j のパターン $\text{Q}_1, \dots, \text{Q}_n$ に一致した場合， Q_i は縛られる．次に Rule_j の形式に応じて， $\text{X}_1, \dots, \text{X}_n$ が $\text{E}_1, \dots, \text{E}_n$ に縛れるか，命令 C が実行され， X_i も更新される．変数が同じ値と一致する場合，繰り返し変数を含むパターンが一致する．

```

read X;
from (= ? Y nil)
loop (Z.X) <= X;
    Y <= (Z.Y)
until(=? X nil);
write Y

```

(a) プログラム reverse

```

read Y;
from (= ? X nil)
loop (Z.Y) <= Y;
    X <= (Z.X)
until(=? Y nil);
write X

```

(b) プログラム $\mathcal{I}[\text{reverse}]$

図 4.5: 逆変換器 \mathcal{I} の例

4.2 可逆チューリング機械から R-WHILE への変換

ここでは、可逆チューリング機械を R-WHILE のプログラムで模倣するための、変換の規則を与える。

4.2.1 可逆チューリング機械を模倣する R-WHILE プログラム

図 4.6a にチューリング機械プログラムからの変換で得られた R-WHILE プログラムを示す。なお記述には変換規則を用いた。

main プログラム (図 4.6a) は入力としてチューリング機械のテープ上に書かれている記号列 R を読み込む。その後、計算を実行し、書き換えられた記号列 R' を出力するというものである。main プログラム本体の Q はチューリング機械の内部状態を表している。また、 T はチューリング機械のテープの状態を表している。そのため、可逆的代入 $Q \leftarrow \bar{q}_s$ によって、内部状態を表す Q は初期状態になる。また可逆的置換 $T \leftarrow (nil \ \bar{b} \ R)$ によって、ヘッドがテープにかかっている記号列の一つ左を指している状態を表している。そして、組 (Q, T) はチューリング機械の様相を表している。プログラム内のループでは、チューリング機械の内部状態が初期状態から最終状態に遷移するまでマクロ STEP が繰り返し実行される。繰り返しを終了した後、命令によって T と Q の値は nil となる。

図 4.6b で定義されるマクロ STEP(Q, T) では、様相 (Q, T) を書き換え規則により書き換える。 $\mathcal{T}[t]^*$ は遷移規則から R-WHILE の書き換え規則への変換器 \mathcal{T} (図 4.7) によって生成された書き換え規則の列である。変換器 \mathcal{T} によって

それぞれの書き換え規則は図 4.7 の変換器 \mathcal{T} によって遷移規則 $t \in \delta$ から生成される。 \bar{q} は、状態 q に対応する R-WHILE のアトムである。可逆チューリング機械の遷移規則列を変換した場合、異なる書き換え規則は矢印 \Rightarrow の左側のパターンと右側で返却される値がそれぞれ重なることはない。

マクロ MOVE \bar{L} (図 4.6c) はヘッドを一つ左に動かすためのマクロである。チューリング機械のテープ (l, s, r) はスタック L と R を用いて $(L \ S \ R)$ として表す。マクロ MOVE \bar{L} はマクロ PUSH と POP を実行し変化したテープの状態を T に置き換える。ヘッドを一つ右に動かすためのマクロ MOVE \bar{R} はマクロ MOVE \bar{L} を逆変換することで得ることができる。マクロ PUSH は、アトム S をスタック STK にプッシュするためのマクロである。 S が空白記号の場合、 S, STK ともに nil をかえす。マクロ POP はマクロ PUSH を逆変換することで得ることができる。ただし、スタック STK が nil だった場合、マクロ POP は空白記号をポップする。

4.3 模倣できていることの証明

この章では、4.2 章で与えた規則が実際に可逆チューリング機械を模倣することができているかを示す。

可逆チューリング機械，可逆チューリング機械の計算ステップ，様相，意味，プログラミング言語 R-WHILE または R-WHILE の書き換えの規則の定義はすでに本稿のなかで定義されたものを用いる．

R-WHILE の \mathbb{D} を可逆チューリング機械の Σ^* へと変換する変換器を R2T とし，以下の様に定義する．

$$R2T : \mathbb{D} \rightarrow \Sigma^*$$

$$R2T[\text{nil}] = 0$$

$$R2T[(d_1, d_2)] = 1 \ R2T[d_1] \cdot R2T[d_2]$$

また，R2T の逆である Σ^* を \mathbb{D} へ変換する変換機を T2R とする．

定理 1 任意の可逆チューリング機械 T，任意の x に対して

$$[T] x = R2T[[T[T]](T2R[x])]$$

である．定理 1 を証明するにあたって補題 1 と補題 2 を与える．ここでは変換器 \mathcal{T} によって変換されたものを $\bar{\cdot}$ をつかって記述する．

補題 1 任意の様相 $\sigma = (q, (l.s.r))$ ， $\sigma' = (q'(l', s', r'))$ に対して，

$$\sigma \xRightarrow{T} \sigma'$$

のとき，

$$\bar{\sigma} = (\bar{q}, (\bar{l}, \bar{s}, \bar{r})) = \{Q \mapsto (\bar{q}), T \mapsto (\bar{l}, \bar{s}, \bar{r})\}$$

として，

$$\begin{aligned} \mathcal{C}[\text{STEP}(Q, T)] \{Q \mapsto (\bar{q}), T \mapsto (\bar{l}, \bar{s}, \bar{r})\} = \\ \bar{\sigma'} = (\bar{q'}, (\bar{l'}, \bar{s'}, \bar{r'})) = \{Q \mapsto (\bar{q'}), T \mapsto (\bar{l'}, \bar{s'}, \bar{r'})\} \end{aligned}$$

が成り立つ．このとき， $\mathcal{C}[\cdot]$ は対象がプログラムであることを示している．

[証明] 補題 1 を証明する為に，可逆チューリング機械における 4 パターンの遷移規則において，補題 1 が成り立つことを示す．R-WHILE で表記するにあたって， $\bar{l} = L$ ， $\bar{s} = S$ または $\bar{r} = R$ とする．

(i) 遷移規則が $(q, \langle s, s' \rangle, q')$ の場合

可逆チューリング機械の様相は

$$(q, (l, s, r)) \xRightarrow{T} (q', (l, s', r))$$

のようになる．一方 R-WHILE では，

$$\mathcal{T} [[q, \langle s, s' \rangle, q']] = [\bar{q}, (L, S, R)] \Rightarrow [\bar{q'}, (L, S', R)]$$

によって

$$[\bar{q'}, (L, S', R)]$$

となり，

$$[\bar{q'}, (L, S, R)] = (q', (l, s', r))$$

である．

(ii) 遷移規則が (q, \leftarrow, q') の場合

可逆チューリング機械の様相は

$$(q, (ls', s, r)) \xRightarrow{T} (q', (l, s', sr))$$

のようになる．一方 R-WHILE では，

$$\mathcal{T} [[q, \leftarrow, q']] = [\bar{q}, T] \Rightarrow \{\text{MOVE}(T); Q \wedge \bar{q}; Q \wedge \bar{q'}\}$$

によって

$$[\bar{q'}, (L, S', SR)]$$

となり ,

$$[\bar{q}', (L, S', SR)] = (q', (l, s', sr))$$

である .

(iii) 遷移規則が (q, \rightarrow, q') の場合
可逆チューリング機械の様相は

$$(q, (l, s, s'r)) \xRightarrow{T} (q', (ls, s', r))$$

のようになる . 一方 R-WHILE では ,

$$\mathcal{T} [[q, \rightarrow, q']] = [\bar{q}, T] \Rightarrow \{\text{MOVER}(T); Q \hat{=} \bar{q}; Q \hat{=} \bar{q}'\}$$

によって

$$[\bar{q}', (LS, S', R)]$$

となり ,

$$[\bar{q}', (LS, S', R)] = (q', (ls, s', r))$$

である .

(iv) 遷移規則が (q, \downarrow, q') の場合
可逆チューリング機械の様相は

$$(q, (l, s, r)) \xRightarrow{T} (q', (l, s, r))$$

のようになる . 一方 R-WHILE では ,

$$\mathcal{T} [[q, \downarrow, q']] = [\bar{q}, T] \Rightarrow [\bar{q}', T]$$

によって

$$[\bar{q}', (L, S, R)]$$

となり ,

$$[\bar{q}', (L, S, R)] = (q', (l, s, r))$$

である .

(i) , (ii) , (iii) または (iv) により , 補題 1 が成り立つのは明らかである .

補題 2 任意の様相 $\sigma = (q, (l.s.r))$, $\sigma' = (q'(l', s', r'))$ または自然数 n に対して ,

$$\sigma \xRightarrow{T}^n \sigma'$$

であるならば ,

$$(\mathcal{C}[\text{STEP}(Q.T)])^n$$

が成り立つ .

[証明] 可逆チューリング機械の計算ステップが n 回実行される場合 , R-WHILE では可逆ループによる繰り返しによって , マクロ STEP(Q,T) を n 回実行する .

(i) $n = 1$ の場合

補題 1 より補題 2 が成り立つのは明らかである .

(ii) $n = k$ の場合

補題 2 が成り立つとすると

$$\sigma \xRightarrow{T}^k \sigma' \iff (\mathcal{C}[\text{STEP}(Q.T)])^k$$

である . $n = k + 1$ のとき

<pre> read R; Q ^ = $\overline{q_s}$; T <= (nil \overline{b} R); from (=? Q $\overline{q_s}$) loop STEP(Q,T) until (=? Q $\overline{q_f}$); (nil \overline{b} R') <= T; Q ^ = $\overline{q_f}$; write R' </pre>	<pre> macro STEP(Q,T) \equiv rewrite [Q,T] by $\mathcal{T}[[t]]^*$ </pre>	<pre> macro MOVEL(T) \equiv (L S R) <= T; PUSH(S,R); POP(S,L); T <= (L S R) </pre>	<pre> macro PUSH(S,STK) \equiv rewrite [S,STK] by $[\overline{b},\text{nil}] \Rightarrow [\text{nil},\text{nil}]$ $[S,\text{STK}] \Rightarrow [\text{nil},(S.\text{STK})]$ </pre>
(a) main プログラム	(b) マクロ STEP	(c) マクロ MOVEL	(d) マクロ PUSH

図 4.6: RTM を模倣する R-WHILE プログラム

$$\begin{aligned}
\mathcal{T}[[q_1, \langle s_1, s_2 \rangle, q_2]] &= [\overline{q_1}, (L \overline{s_1} R)] \Rightarrow [\overline{q_2}, (L \overline{s_2} R)] \\
\mathcal{T}[[q_1, \leftarrow, q_2]] &= [\overline{q_1}, T] \Rightarrow \{\text{MOVEL}(T); Q \wedge = \overline{q_1}; Q \wedge = \overline{q_2}\} \\
\mathcal{T}[[q_1, \rightarrow, q_2]] &= [\overline{q_1}, T] \Rightarrow \{\text{MOVER}(T); Q \wedge = \overline{q_1}; Q \wedge = \overline{q_2}\} \\
\mathcal{T}[[q_1, \downarrow, q_2]] &= [\overline{q_1}, T] \Rightarrow [\overline{q_2}, T]
\end{aligned}$$

図 4.7: 遷移規則から R-WHILE の書き換え規則への変換器 \mathcal{T}

$$(\mathcal{C}[[\text{STEP}(Q.T)]]^{k+1})$$

であり ,

$$(\mathcal{C}[[\text{STEP}(Q.T)]]^{k+1}) = (\mathcal{C}[[\text{STEP}(Q.T)]]^1)(\mathcal{C}[[\text{STEP}(Q.T)]]^k)$$

であるため , $n = k + 1$ のときも補題 2 は成立する .

(i) , (ii) よりすべての自然数 n について補題 2 は成立する .

補題 1 , 補題 2 より定理 1 は成り立つ .

第 5 章

結論

5.1 結果

5.2 考察

謝辞

本研究を進めるにあたり，研究の進め方や論文の査読をはじめとした，本研究全体を通しての，横山准教授のご指導に感謝致します。青山幹雄教授と宮澤元准教授には，中間発表会において研究の方向性や考え方に関するコメントを頂きましたことに心より感謝致します。最後に，研究を共に行い，予稿，本稿，要旨の査読と助言をして頂きました横山研究室の皆様には感謝致します。

付録

参考文献

- [1] Stephen C. Kleene: The Church-Turing Thesis, Stanford Encyclopedia of Philosophy (online), available from <<https://plato.stanford.edu/entries/church-turing/>> (accessed 2017-09-27).
- [2] Ryo Aoki, Shintaro Shibata, Tetsuo Yokoyama.: A universal reversible Turing machine program in reversible programming language R-WHILE (2016).
- [3] 丸岡 章: “計算理論とオートマトン言語理論 コンピュータの原理を明かす “, pp.4–5, 142–169, サイエンス社 (2005).
- [4] 米田 政明, 広瀬 貞樹, 大里 延康, 大川 知: “オートマトン・言語理論の基礎 “, pp.60, 85–103, 近代科学社 (2003).
- [5] 森田 憲一: “可逆計算 ナチュラルコンピューティングシリーズ Vo5“, pp.13–22, 近代科学社 (2012).
- [6] Axelsen, H. B. and Glück, R.: What Do Reversible Programs Compute?, *Foundations of Software Science and Computational Structures. Proceedings* (Hofmann, M. , ed.), LNCS, Vol.6604, Springer-Verlag, pp. 42–56 (2011)
- [7] Glück, R. and Yokoyama, T.: A Linear-Time Self-Interpreter of a Reversible Imperative Language.
- [8] Yokoyama, T. , Axelsen, H. B. and Glück, R.: Towards a Reversible Functional Language, *Reversible Computation. Proceedings* (De Vos, A. and Wille, R. , eds.), LNCS, Vol. 7165, Springer-Verlag, pp. 14–29 (2012)
- [9] Jones, N. D.: Computability and Complexity: *From a Programming Perspective*, MIT Press (1997). Revised version, available from <http://www.diku.dk/~neil/Comp2book.html>.