Involutory Turing Machines

Keisuke Nakano^(✉)

Research Institute of Electrical Communication, Tohoku University, Sendai, Japan
k.nakano@acm.org

Abstract. An involutory function, also called involution, is a function f that is its own inverse, i.e., $f(f(x)) = x$ holds whenever $f(x)$ is defined. This paper presents a computational model of involution as a variant of Turing machines, called an *involutory Turing machine*. The computational model is shown to be complete in the sense that not only does an involutory Turing machine always compute an involution but also every involutory computable function can be computed by an involutory Turing machine. As any involution is injective (hence reversible), any involutory Turing machine forms a standard reversible Turing machine that is backward deterministic. Furthermore, the existence of a universal involutory Turing machine is shown under an appropriate redefinition of universality given by Axelsen and Glück for reversible Turing machines. This work is motivated by characterizing bidirectional transformation languages.

Keywords: Involution · Reversible Turing machine · Universal Turing machine · Bidirectional transformation language · Time-symmetric machine

1 Introduction

An involutory function, also called involution, is a function that is its own inverse, i.e., $f(f(x)) = x$ holds whenever $f(x)$ is defined. In mathematics, because of their symmetric behavior, involutions have been used for solving functional equations and proving theorems, e.g., Zagier’s one-sentence proof for Fermat’s theorem on sums of two squares [12]. Even in computer science, involutions appear in cryptographic systems such as one-time pad and RC4.

This paper presents a computational model for involution as a variant of Turing machines with *function semantics*, where input and output words are specified by tapes of initial and final configurations, respectively. The idea to have such a model for involution is to impose a restriction on a standard Turing machine so that the reversed run of every valid run is valid. The restriction can be simply described by associating one transition rule with another according to a certain involution over states. We call it an *involutory Turing machine*. It is easy to find that an involutory Turing machine always computes an involution under the restriction.

The present paper takes a further step. The involutory Turing machine is shown to be a ‘complete’ computational model for involution: any involutory computable function can be defined by an involutory Turing machine. That is, for a given non-involutory Turing machine that computes an involution, there exists an equivalent involutory Turing machine.

This work is inspired by Axelsen and Glück’s work [1,2] where the expressiveness of *reversible Turing machines* is discussed. A reversible Turing machine is defined as a backward-deterministic Turing machine and hence computes only an injective function. They have shown that any injective computable function can be defined by a reversible Turing machine as we will show for involutory Turing machines. As an involutory function is a special kind of injective function, an involutory Turing machine can be regarded as a special reversible Turing machine.

Furthermore, this paper addresses the universality of involutory Turing machines as done by Axelsen and Glück [2] for reversible Turing machines. A standard Turing machine is said to be universal if it can simulate any Turing machine on arbitrary input, and it is known that there is a universal Turing machine. As for involutory Turing machines, there is no universal machine under the same definition of universality because the simulating function is not involutory. Therefore, we adopt an alternative definition of universality which has been introduced by Axelsen and Glück for reversible Turing machines. In their definition, the universal machine is allowed to preserve a given machine description as part of the output. We will show the existence of a universal involutory Turing machine under this redefinition.

In summary, the main contributions of this paper are as follows.

- An involutory Turing machine is proposed as a multi-tape Turing machine with restrictive transition rules and tape permutation. An involutory Turing machine always computes an involution.
- An involutory Turing machine is shown to be complete, i.e., every computable involution is defined by an involutory Turing machine.
- It is shown that for every k -tape involutory Turing machine, there exists a 2-tape involutory Turing machine that computes an isomorphic function.
- It is shown that there exists a universal involutory Turing machine in terms of an appropriate definition of universality.

In addition to the above, our design choice, limitations, and applications of involutory Turing machines will be discussed in Sect. 6. In particular, an application to bidirectional transformation will shed a light on a practical aspect of our computational model for involution.

The restriction imposed on Turing machines to be involutory coincides with *time symmetry* introduced by Gajardo et al. [5] for cellular automata to describe a corresponding physical notion. One might call our model a *time-symmetric Turing machine* in this sense. More detail is discussed as one of the related work in Sect. 7.

2 Preliminaries

The set of non-negative integers is denoted by $\mathbb{N}$. For $n \in \mathbb{N}$, the set $\{1, \dots, n\}$ is denoted by $[n]$, in particular, $[0] = \emptyset$. The set of all words over an alphabet (that is a finite set of symbols) Σ is denoted by Σ^* . For convenience, we assume that a nested tuple of words can be regarded as a flattened one, e.g., $((w_1, w_2), w_3)$ and $(w_1, (w_2, w_3))$ may be identified with (w_1, w_2, w_3) for $w_1, w_2, w_3 \in \Sigma^*$.

For a (binary) relation $R \subseteq A \times B$, $a R b$ stands for $(a, b) \in R$. The identity relation $Id_A \subseteq A \times A$ is $\{(a, a) \mid a \in A\}$. The composition of two relations $R \subseteq A \times B$ and $S \subseteq B \times C$, denoted by $S \circ R$, is given as $\{(a, c) \mid \exists b \in B, a R b \wedge b S c\}$. For a relation $R \subseteq A \times B$ over two sets A and B , the *inverse relation* $R^{-1} \subseteq B \times A$ is defined by $\{(b, a) \mid a R b\}$. A relation $R \subseteq A \times A$ is said to be *symmetric* if $R^{-1} = R$. A relation $R \subseteq A \times B$ is said to be *functional* if $a R b_1$ and $a R b_2$ imply $b_1 = b_2$ for any $a \in A$ and $b_1, b_2 \in B$. A functional relation $R \subseteq A \times B$, written by $R : A \rightarrow B$, is simply called a (partial) *function* and $R(a)$ with $a \in A$ stands for $b \in B$ such that $a R b$ if exists. A function $R : A \rightarrow B$ is said to be *total* if $R(a) \in B$ is defined for any $a \in A$. A function $R : A \rightarrow B$ is said to be *injective* if R^{-1} is functional. For any injective function $R : A \rightarrow B$ it is easy to see that $R^{-1} \circ R \subseteq Id_A$ and $R \circ R^{-1} \subseteq Id_B$ hold. A function $R : A \rightarrow B$ is said to be *bijective* if both R and R^{-1} are total and injective. A function $R : A \rightarrow A$ is called an *involution* if R is symmetric. An involutory function is also called *involution*.

A *permutation* on $[k]$ is a bijective function over $[k]$ for a fixed integer $k \in \mathbb{N}$. A permutation can be expressed as the product of disjoint cycles, e.g., $(1\ 5\ 4)(3\ 7)$ denotes a permutation π such that $\pi(1) = 5$, $\pi(5) = 4$, $\pi(4) = 1$, $\pi(3) = 7$, $\pi(7) = 3$, and $\pi(i) = i$ for any other i . The inversion of a permutation is obtained by reversing every cycle, e.g., $((1\ 5\ 4)(3\ 7))^{-1} = (4\ 5\ 1)(7\ 3)$.

3 The Turing Machine and Its Known Variants

The Turing machine is one of the best-known computational models which can implement any computable function. A Turing machine manipulates symbols on a doubly-infinite tape of cells according to an internal state and a fixed transition relation. We use a triplet format [2] to represent the transition relation without loss of generality. Although it is well known that they are equivalent to single-tape Turing machines in power [10], we consider multi-tape Turing machines to make it easy to investigate various properties of involutory Turing machines. Moreover, our model of multi-tape Turing machines has a single instruction for permuting the order of tapes. This feature does not change the expressive power of Turing machines. As its byproduct, we can limit any other instruction only to the first tape. To apply an instruction to the other tape, we permute tapes so as for the tape to be the first before the instruction (and permute them back if needed).

Definition 3.1 (*k*-tape Turing machine). A *k*-tape Turing machine T is a tuple $(Q, \Sigma, q_{\text{ini}}, q_{\text{fin}}, \Delta)$ where Q is a finite set of *states*, Σ is a *tape alphabet* not containing the special blank symbol $\sqcup$, $q_{\text{ini}} \in Q$ is the *initial state*, $q_{\text{fin}} \in Q$ is the *final state*, and $\Delta = \Delta^{\text{rw}} \uplus \Delta^{\leftarrow} \uplus \Delta^{\uparrow}$ is a ternary relation defining a set of *transition rules* where:

$$\begin{aligned} \Delta^{\text{rw}} &\subseteq (Q \setminus \{q_{\text{fin}}\}) \times (\Sigma_{\sqcup} \times \Sigma_{\sqcup}) \times (Q \setminus \{q_{\text{ini}}\}) && \text{(SYMBOL RULES)} \\ \Delta^{\leftarrow} &\subseteq (Q \setminus \{q_{\text{fin}}\}) \times \{\leftarrow, \diamond, \rightarrow\} \times (Q \setminus \{q_{\text{ini}}\}) && \text{(MOVE RULES)} \\ \Delta^{\uparrow} &\subseteq (Q \setminus \{q_{\text{fin}}\}) \times \Pi_k \times (Q \setminus \{q_{\text{ini}}\}) && \text{(PERMUTATION RULES)} \end{aligned}$$

where $\Sigma_{\sqcup}$ stands for $\Sigma \uplus \{\sqcup\}$ and Π_k is the set of all permutations over $[k]$. For $q, q' \in Q$, a *symbol rule* in Δ^{rw} has the form $(q, s \rightarrow s', q')$ with $s, s' \in \Sigma_{\sqcup}$; a *move rule* in $\Delta^{\leftarrow}$ has the form (q, d, q') with $d \in \{\leftarrow, \diamond, \rightarrow\}$; a *permutation rule* in $\Delta^{\uparrow}$ has the form $(q, \uparrow \pi, q')$ with permutation $\pi \in \Pi_k$. A permutation rule is said to be *involutory* if π is involutory.

As presented in [2], symbol rules and move rules are independently given for the convenience of further discussion. Although these two kinds of actions are caused by a single rule in standard Turing machines [10], the separation of rules does not change the expressiveness of functions. It is easy to simulate a transition rule in a standard Turing machine by two transition rules and extra states in the present model. Moreover, the present model introduces permutation rules $(q, \uparrow \pi, q')$, which permute k tapes without moving their heads in the order given by a permutation π . Again, they do not change the expressiveness because the operation can be simulated by a standard Turing machine with maintaining the left and right ends at the used cells for every tape to copy them to each other.

The configuration of a *k*-tape Turing machine is specified by the current internal state and k tapes with their tape head. The status of a tape with its head is represented by $\langle l, s, r \rangle \in \Sigma_{\sqcup}^{\omega} \times \Sigma_{\sqcup} \times \Sigma_{\sqcup}^{\omega}$ where s is the symbol at its head position and l and r are the left and right tapes of the head. Note that $\Sigma_{\sqcup}^{\omega}$ is the set of infinite words going infinitely to the right. Accordingly, l is ‘mirrored’ where its first symbol is the immediate left one of the head.

Definition 3.2 (Configuration). The *configuration* of a *k*-tape Turing machine $T = (Q, \Sigma, q_{\text{ini}}, q_{\text{fin}}, \Delta)$ is a tuple $(q, (\langle l_1, s_1, r_1 \rangle, \dots, \langle l_k, s_k, r_k \rangle))$ where $q \in Q$ is an *internal state*, $l_i, r_i \in \Sigma_{\sqcup}^{\omega}$ for each $i \in [k]$ are the left and right of the i -th tape head and include only finite non-blank symbols, and $s_i \in \Sigma_{\sqcup}$ for each $i \in [k]$ is the symbol at the i -th tape head. The set of all configurations of T is written by $\mathcal{C}_T$.

Definition 3.3 (Computation step). Let $T = (Q, \Sigma, q_{\text{ini}}, q_{\text{fin}}, \Delta)$ be a *k*-tape Turing machine. Then a single *computation step* is defined as a relation $\vdash_T$ over

$\mathcal{C}_T$ such that

$$\begin{array}{ll}
(q, (\langle l, s, r \rangle, \dots)) \vdash_T (q', (\langle l, s', r \rangle, \dots)) & \text{when } (q, s \Rightarrow s', q') \in \Delta \\
(q, (\langle s'l, s, r \rangle, \dots)) \vdash_T (q', (\langle l, s', sr \rangle, \dots)) & \text{when } (q, \leftarrow, q') \in \Delta \\
(q, (\langle l, s, r \rangle, \dots)) \vdash_T (q', (\langle l, s, r \rangle, \dots)) & \text{when } (q, \diamond, q') \in \Delta \\
(q, (\langle l, s, s'r \rangle, \dots)) \vdash_T (q', (\langle sl, s', r \rangle, \dots)) & \text{when } (q, \rightarrow, q') \in \Delta \\
(q, (t_1, \dots, t_k)) \vdash_T (q', (t_{\pi(1)}, \dots, t_{\pi(k)})) & \text{when } (q, \uparrow \pi, q') \in \Delta.
\end{array}$$

The reflexive transitive closure of $\vdash_T$ is denoted by $\vdash_T^*$.

The semantics of a k -tape Turing machine T is given by a relation over k words based on $\vdash_T^*$ as below. We follow the style of Axelsen and Glück called a *function semantics* where an input and an output word are in the tape at the initial and the final configuration of a run, respectively, rather than the usual style with input and output tapes. This view makes it easier to capture the functional behavior of Turing machines. In the rest of the paper, a finite word $w \in \Sigma^*$ is used to represent an infinite word $w \sqcup^\omega \in \Sigma_\sqcup^\omega$; thereby ε denotes $\sqcup^\omega$.

Definition 3.4 (Function semantics of Turing machines). Let $T = (Q, \Sigma, q_{\text{ini}}, q_{\text{fin}}, \Delta)$ be a k -tape Turing machine. The *semantics* of T , denoted by $\llbracket T \rrbracket$, is given by the relation

$$\begin{aligned}
\llbracket T \rrbracket = \{ & ((w_1, \dots, w_k), (w'_1, \dots, w'_k)) \in (\Sigma^*)^k \times (\Sigma^*)^k \\
& \mid (q_{\text{ini}}, (\langle \varepsilon, \sqcup, w_1 \rangle, \dots, \langle \varepsilon, \sqcup, w_k \rangle)) \vdash_T^* (q_{\text{fin}}, (\langle \varepsilon, \sqcup, w'_1 \rangle, \dots, \langle \varepsilon, \sqcup, w'_k \rangle)) \}.
\end{aligned}$$

Recall that we may write $\llbracket T \rrbracket(w_1, \dots, w_k) = (w'_1, \dots, w'_k)$ if $\llbracket T \rrbracket$ is functional.

In the rest of the paper, for every Turing machine T it is assumed that for any sequence $(q_{\text{ini}}, (\langle l_1, s_1, r_1 \rangle, \dots, \langle l_k, s_k, r_k \rangle)) \vdash_T^* (q_{\text{fin}}, (\langle l'_1, s'_1, r'_1 \rangle, \dots, \langle l'_k, s'_k, r'_k \rangle))$ of computation steps, we have $(l_1, s_1) = \dots = (l_k, s_k) = (\varepsilon, \sqcup)$ if and only if $(l'_1, s'_1) = \dots = (l'_k, s'_k) = (\varepsilon, \sqcup)$. We call it the *tidiness* assumption.

Here are two examples of Turing machines. It is easy to see that they naturally conform to the tidiness assumption. Moreover, both examples are reversible as seen later.

Example 3.5. Let π be a permutation on $[k]$. The k -tape Turing machine $\text{Perm}(\pi) = (\{q_{\text{ini}}, q_{\text{fin}}\}, \Sigma, q_{\text{ini}}, q_{\text{fin}}, \{(q_{\text{ini}}, \uparrow \pi, q_{\text{fin}})\})$ computes a function which permutes its arguments in accordance with π .

Example 3.6. The 1-tape Turing machine $T_{\text{bnot}} = (\{q_{\text{ini}}, q_{\text{bnot}}, q_{\text{done}}, q_{\text{back}}, q_{\text{fin}}\}, \{0, 1\}, q_{\text{ini}}, q_{\text{fin}}, \Delta)$ with

$$\begin{aligned}
\Delta = \{ & (q_{\text{ini}}, \sqcup \Rightarrow \sqcup, q_{\text{next}}), \\
& (q_{\text{next}}, \rightarrow, q_{\text{bnot}}), (q_{\text{bnot}}, 0 \Rightarrow 1, q_{\text{next}}), (q_{\text{bnot}}, 1 \Rightarrow 0, q_{\text{next}}), (q_{\text{bnot}}, \sqcup \Rightarrow \sqcup, q_{\text{back}}), \\
& (q_{\text{back}}, \leftarrow, q_{\text{done}}), (q_{\text{done}}, 0 \Rightarrow 0, q_{\text{back}}), (q_{\text{done}}, 1 \Rightarrow 1, q_{\text{back}}), (q_{\text{done}}, \sqcup \Rightarrow \sqcup, q_{\text{fin}}) \}
\end{aligned}$$

computes a bitwise negation.

Definition 3.4 implies that the semantics of a Turing machine returns a tuple that consists of the same number of words as a given input. However, when the function either accepts (returns) only empty word for some specific arguments, we may regard it as a function whose input (output) tuple consists of fewer words. The next example $Dup(1)$ illustrates the case where a 2-tape Turing machine computes a function that takes a single word and returns a pair of words.

Example 3.7. A 2-tape Turing machine $Dup(1) = (Q, \Sigma, q_{ini}, q_{fin}, \Delta)$ with

$$\begin{aligned} Q &= \{q_{ini}, q_{skip}, q_{check}, q_{back}, q_{get}, q_{put}, q_{done}, q_{fin}\} \cup \bigcup_{a \in \Sigma} \{q_{g(a)}, q_{p(a)}\} \\ \Delta &= \{(q_{ini}, \sqcup \Rightarrow \sqcup, q_{skip}), (q_{skip}, \rightarrow, q_{check}), (q_{check}, \sqcup \Rightarrow \sqcup, q_{back}), \\ &\quad (q_{back}, \leftarrow, q_{get}), (q_{put}, \leftarrow, q_{done}), (q_{done}, \uparrow(1\ 2), q_{back}), (q_{get}, \sqcup \Rightarrow \sqcup, q_{fin}), \} \cup \\ &\quad \bigcup_{a \in \Sigma} \{(q_{check}, a \Rightarrow a, q_{skip}), (q_{get}, a \Rightarrow a, q_{g(a)}), \\ &\quad (q_{g(a)}, \uparrow(1\ 2), q_{p(a)}), (q_{p(a)}, \sqcup \Rightarrow a, q_{put})\} \end{aligned}$$

computes a duplicate function, i.e., $\llbracket Dup(1) \rrbracket(w) = (w, w)$ for any $w \in \Sigma^*$. Similarly, we can define a $2k$ -tape Turing machine such that $\llbracket Dup(k) \rrbracket(w_1, \dots, w_k) = (w_1, \dots, w_k, w_1, \dots, w_k)$ for any $w_1, \dots, w_k \in \Sigma^*$.

Definition 3.8 (Forward/backward determinism). Let $T = (Q, \Sigma, q_{ini}, q_{fin}, \Delta)$ be a k -tape Turing machine. Then T is *forward deterministic* if, for any distinct pair $(q, a_1, q_1), (q, a_2, q_2) \in \Delta$ of transition rules, we have $a_1 = s_1 \Rightarrow s'_1$ and $a_2 = s_2 \Rightarrow s'_2$ with some $s_1, s'_1, s_2, s'_2 \in \Sigma_{\sqcup}$ and $s_1 \neq s_2$. The Turing machine T is *backward deterministic* if, for any distinct pair $(q_1, a_1, q), (q_2, a_2, q) \in \Delta$ of transition rules, we have $a_1 = s_1 \Rightarrow s'_1$ and $a_2 = s_2 \Rightarrow s'_2$ with some $s_1, s'_1, s_2, s'_2 \in \Sigma_{\sqcup}$ and $s'_1 \neq s'_2$.

The forward (backward) deterministic Turing machine has no pair of move rules which have the same source (target) state. With regard to a configuration step $C_1 \vdash_T C_2$, C_1 uniquely determines C_2 if T is forward deterministic while C_2 uniquely determines C_1 if T is backward deterministic. The definition of forward and backward determinism is exactly the same as *local* forward and backward determinism in [2].

Definition 3.9 (Reversible Turing machine). A k -tape Turing machine T is *reversible* if T is forward and backward deterministic.

Given an involutory function φ over a set Q of states such $\varphi(q_{ini}) = q_{fin}$ (hence, $\varphi(q_{fin}) = q_{ini}$), let us define a function $\tilde{\varphi}$ which ‘flips’ transition rules in terms of φ . This function plays important roles to recognize properties of reversible and involutory Turing machines. The function $\tilde{\varphi}$ is defined as a bijective function over transition rules by

$$\tilde{\varphi}((q, (a_1, a_2, \dots, a_k), q')) = (\varphi(q'), (a_1^{-1}, a_2^{-1}, \dots, a_k^{-1}), \varphi(q))$$

where $(s \dot{\rightarrow} s')^{-1} = s' \dot{\rightarrow} s$, $(\leftarrow)^{-1} = \rightarrow$, $(\diamond)^{-1} = \diamond$, $(\rightarrow)^{-1} = \leftarrow$, and $(\uparrow \pi)^{-1} = \uparrow(\pi^{-1})$. The function $\tilde{\varphi}$ is naturally extended for a set of transition rules and a Turing machine. For a set Δ of transition rules, $\tilde{\varphi}(\Delta)$ represents $\{\tilde{\varphi}(r) \mid r \in \Delta\}$. For a k -tape Turing machine $T = (Q, \Sigma, q_{\text{ini}}, q_{\text{fin}}, \Delta)$, $\tilde{\varphi}(T)$ represents $(Q, \Sigma, q_{\text{fin}}, q_{\text{ini}}, \tilde{\varphi}(\Delta))$. For a k -tape Turing machine T , $\tilde{\varphi}(T)$ is forward (backward) deterministic if T is backward (forward) deterministic. It is easy to see that the function $\tilde{\varphi}$ is involutory for any involutory function φ , that is, $\tilde{\varphi}(\tilde{\varphi}(x)) = x$ for any transition rule, any set of transition rules, and any Turing machine x as long as $\tilde{\varphi}(x)$ is defined.

Let the function $\tilde{\varphi}$ be extended for a configuration of a Turing machine as well so that $\tilde{\varphi}((q, (t_1, \dots, t_k))) = (\varphi(q), (t_1, \dots, t_k))$. Then the following proposition holds straightforwardly.

Proposition 3.10. Let T be a k -tape Turing machine and φ be an involution over a set of states in T . Then, for any a computation step $C_1 \vdash_T C_2$, we have $\tilde{\varphi}(C_2) \vdash_{\tilde{\varphi}(T)} \tilde{\varphi}(C_1)$.

The simplest function ι as such φ is given as $\iota(q_{\text{ini}}) = q_{\text{fin}}$, $\iota(q_{\text{fin}}) = q_{\text{ini}}$, and $\iota(q) = q$ for $q \in Q \setminus \{q_{\text{ini}}, q_{\text{fin}}\}$. Let us write T^{-1} for $\tilde{\iota}(T)$. Then, for a given reversible Turing machine T , T^{-1} gives an inversion of T as shown by Bennett [3] and reformulated by Axelsen and Glück [2].

Theorem 3.11 (Semantics of reversible Turing machines). Given a k -tape reversible Turing machine T , $\llbracket T^{-1} \rrbracket = \llbracket T \rrbracket^{-1}$ holds.

Proof. For simplicity of the proof, only the case of $k = 1$ is shown. The proof can be easily generalized to the other cases. Let $T = (Q, \Sigma, q_{\text{ini}}, q_{\text{fin}}, \Delta)$ be a 1-tape reversible Turing machine. The equation $\llbracket T^{-1} \rrbracket(w) = v$ holds if and only if $(q_{\text{fin}}, \langle \varepsilon, \sqcup, w \rangle) \vdash_{T^{-1}}^* (q_{\text{ini}}, \langle \varepsilon, \sqcup, v \rangle)$ by definition. From Proposition 3.10, $(q_{\text{ini}}, \langle \varepsilon, \sqcup, v \rangle) \vdash_T^* (q_{\text{fin}}, \langle \varepsilon, \sqcup, w \rangle)$ holds, which implies $\llbracket T \rrbracket(v) = w$, i.e., $\llbracket T \rrbracket^{-1}(w) = v$. $\square$

A reversible Turing machine is a complete model in the sense that every injective computable function can be defined by a reversible Turing machine, which has been proved by Axelsen and Glück. Even though our computational model slightly differs from their one in that it has tape permutation rules, their proof of the statement works in our setting because our k -tape Turing machine can simulate theirs with k tapes and vice versa.

Theorem 3.12 (Expressiveness of reversible Turing machines [2]). The reversible Turing machines can compute exactly all injective computable functions. That is, given a k -tape Turing machine T such that $\llbracket T \rrbracket$ is injective, there is a k -tape reversible Turing machine T' such that $\llbracket T' \rrbracket = \llbracket T \rrbracket$.

The Turing machines $\text{Perm}(\pi)$ given in Example 3.5, T_{bnot} given in Example 3.6, and $\text{Dup}(k)$ given in Example 3.7 are all reversible. In particular, the inverse $\text{Dup}(k)^{-1}$ computes a partial function that checks equivalence between the first k words and the last k words and returns the k words if the check succeeds.

Definition 3.13 (Concatenation of Turing machines). Let $T_1 = (Q_1, \Sigma, q_{\text{ini},1}, q_{\text{fin},1}, \Delta_1)$ and $T_2 = (Q_2, \Sigma, q_{\text{ini},2}, q_{\text{fin},2}, \Delta_2)$ be k -tape Turing machines where $Q_1 \cap Q_2 = \emptyset$ without loss of generality. The *concatenation* of T_1 and T_2 , denoted by $T_2 \circ T_1$, is a k -tape Turing machine $(Q_1 \uplus Q_2, \Sigma, q_{\text{ini},1}, q_{\text{fin},2}, \Delta_1 \uplus \Delta_2 \uplus \{(q_{\text{fin},1}, \diamond, q_{\text{ini},2})\})$.

In the case where Q_1 and Q_2 are not disjoint, every state in either should be renamed before the concatenation. The reversibility of Turing machines is closed under concatenation as stated below. The proof is straightforward.

Proposition 3.14 (Concatenation of reversible Turing machines). If T_1 and T_2 are reversible Turing machines, so is $T_2 \circ T_1$.

The semantics of concatenation of two reversible Turing machines is equivalent to the function composition of their semantics as shown in the following theorem. Note that Turing machines to be concatenated are assumed tidy. The Turing machine obtained by the concatenation is also tidy.

Theorem 3.15 (Semantics of concatenation of reversible Turing machines). For two k -tape reversible Turing machines T_1 and T_2 , we have $\llbracket T_2 \circ T_1 \rrbracket = \llbracket T_2 \rrbracket \circ \llbracket T_1 \rrbracket$.

Proof. For simplicity of the proof, only the case of $k = 1$ is shown. The proof can be easily generalized to the other cases. Let $T_1 = (Q_1, \Sigma, q_{\text{ini},1}, q_{\text{fin},1}, \Delta_1)$ and $T_2 = (Q_2, \Sigma, q_{\text{ini},2}, q_{\text{fin},2}, \Delta_2)$ be k -tape Turing machines. When $\llbracket T \rrbracket(w_1) = w_2$, we show that there exists w such that $\llbracket T_1 \rrbracket(w_1) = w$ and $\llbracket T_2 \rrbracket(w) = w_2$. By the construction of $T = T_2 \circ T_1$, there exists a sequence of computation steps $(q_{\text{ini},1}, \langle \varepsilon, \sqcup, w \rangle) \vdash_T^* (q_{\text{fin},2}, \langle \varepsilon, \sqcup, v \rangle)$. Because of the construction of transition rules of T , the sequence contains exactly one computation step induced by the rule $(q_{\text{fin},1}, \diamond, q_{\text{ini},2})$ which bridges Q_1 and Q_2 . Hence, the sequence has the form $(q_{\text{ini},1}, \langle \varepsilon, \sqcup, w \rangle) \vdash_T^* (q_{\text{fin},1}, \langle l, s, w' \rangle) \vdash_T (q_{\text{ini},2}, \langle l, s, w' \rangle) \vdash_T^* (q_{\text{fin},2}, \langle \varepsilon, \sqcup, v \rangle)$. The tidiness assumption and determinism (coming from reversibility) of T_1 and T_2 result in $l = \varepsilon$, $s = \sqcup$, $\llbracket T_1 \rrbracket(w_1) = w'$, and $\llbracket T_2 \rrbracket(w') = w_2$. $\square$

The k -tape Turing machine T can be seen as the m -tape Turing machine when $k \leq m$ by leaving all $(k + 1)$ -th through m -th tapes and their heads unchanged. We write $\text{Ext}_{k \rightarrow m}(T)$ for the extended Turing machine.

Proposition 3.16 (Extended Turing machines). Let T be a k -tape Turing machine with $k \leq m$. Then we have $\llbracket \text{Ext}_{k \rightarrow m}(T) \rrbracket(w_1, \dots, w_k, w_{k+1}, \dots, w_m) = (\llbracket T \rrbracket(w_1, \dots, w_k), w_{k+1}, \dots, w_m)$ for any words $w_1, \dots, w_k, w_{k+1}, \dots, w_m$ whenever the right-hand side is defined. If T is reversible, so is $\text{Ext}_{k \rightarrow m}(T)$. Furthermore, $(\text{Ext}_{k \rightarrow m}(T))^{-1} = \text{Ext}_{k \rightarrow m}(T^{-1})$ holds.

4 Involutory Turing Machine

Involutory Turing machines are introduced and investigated in this section. As reversible Turing machines exactly characterize all injective computable functions, involutory Turing machines exactly characterize all involutory computable

functions. As every involutory function is injective, an involutory Turing machine is defined as a special kind of reversible Turing machine. The tape reduction on the involutory Turing machines is also addressed in this section.

Definition 4.1 (Involutory Turing machine). Let $T = (Q, \Sigma, q_{\text{ini}}, q_{\text{fin}}, \Delta)$ be a k -tape Turing machine and φ be an involutory function over Q such that $\varphi(q_{\text{ini}}) = q_{\text{fin}}$ (hence $\varphi(q_{\text{fin}}) = q_{\text{ini}}$). Then T is *involutory* if T is reversible, $\tilde{\varphi}(T) = T$ holds, and every permutation rule is involutory. The function φ is called a *state involution* of T .

Theorem 4.2 (Semantics of involutory Turing machine). If T is an involutory Turing machine T , then $\llbracket T \rrbracket$ is involutory.

Proof. Let $T = (Q, \Sigma, q_{\text{ini}}, q_{\text{fin}}, \Delta)$ be an involutory Turing machine with state involution φ . We show that $\llbracket T \rrbracket(w) = v$ implies $\llbracket T \rrbracket(v) = w$. Assume $\llbracket T \rrbracket(w) = v$. By definition, we have $(q_{\text{ini}}, \langle \varepsilon, \sqcup, w \rangle) \vdash_T^* (q_{\text{fin}}, \langle \varepsilon, \sqcup, v \rangle)$. Since T is involutory, that is, $\tilde{\varphi}(T) = T$, we have $(q_{\text{ini}}, \langle \varepsilon, \sqcup, v \rangle) \vdash_T^* (q_{\text{fin}}, \langle \varepsilon, \sqcup, w \rangle)$ by Proposition 3.10. $\square$

Example 4.3. The Turing machine $\text{Perm}(\pi)$ in Example 3.5 is involutory whenever π is involutory.

The Turing machine T_{not} in Example 3.6 is *not* involutory, although its semantics is involutory. We will see later that there exists an involutory Turing machine equivalent to a given Turing machine whenever its semantics is involutory.

The class of involutory Turing machines have one of the typical and trivial properties of involution, that is closed under conjugation, i.e., for any injective function g , $g^{-1} \circ f \circ g$ is involutory whenever so is f . In terms of Turing machines, the property is described by the following statement.

Lemma 4.4 (Closed under conjugation). Let T be a k -tape involutory Turing machine. For any k -tape reversible Turing machine T_r , the k -tape reversible Turing machine $T_r^{-1} \circ T \circ T_r$ is involutory.

Proof. Let $T = (Q, \Sigma, q_{\text{ini}}, q_{\text{fin}}, \Delta)$ be a k -tape involutory Turing machine with state involution φ , and $T_r = (Q_r, \Sigma_r, q_{\text{ini},r}, q_{\text{fin},r}, \Delta_r)$ be a k -tape reversible Turing machine. We assume that every state q_r of T_r^{-1} is renamed to $\bar{q}_r$ and let $\bar{Q}_r = \{\bar{q}_r \mid q_r \in Q_r\}$. Since the k -tape Turing machine $T_c = T_r^{-1} \circ T \circ T_r$ is reversible from Proposition 3.14, it suffices to show the existence of a state involution of T_c . The state involution $\varphi_c : Q_c \rightarrow Q_c$ with $Q_c = \bar{Q}_r \uplus Q \uplus Q_r$ can be defined by $\varphi_c(\bar{q}_r) = q_r$ for $\bar{q}_r \in \bar{Q}_r$, $\varphi_c(q) = \varphi(q)$ for $q \in Q$, and $\varphi_c(q_r) = \bar{q}_r$ for $q_r \in Q_r$. $\square$

For an injective function f , a function $g(x, y) = (f(y), f^{-1}(x))$ is involutory because $g(g(x, y)) = (f(f^{-1}(x)), f^{-1}(f(y))) = (x, y)$ holds as long as $g(x, y)$ is defined. Similarly, a 2-tape involutory Turing machine can be constructed from a 1-tape reversible Turing machine. The following lemma shows a more general statement.

Lemma 4.5. (k -tape reversible to $2k$ -tape involutory Turing machine).

Given a k -tape reversible Turing machine T , there exists a $2k$ -tape involutory Turing machine T' such that $\llbracket T' \rrbracket(w_1, \dots, w_k, v_1, \dots, v_k) = (\llbracket T \rrbracket^{-1}(v_1, \dots, v_k), \llbracket T \rrbracket(w_1, \dots, w_k))$ for any input words $w_1, \dots, w_k, v_1, \dots, v_k$ of T .

Proof. Let $T = (Q, \Sigma, q_{\text{ini}}, q_{\text{fin}}, \Delta)$ be a k -tape reversible Turing machine. The corresponding $2k$ -tape Turing machine T' is constructed as

$$\text{Ext}_{k \rightarrow 2k}(T)^{-1} \circ \text{Perm}(\pi) \circ \text{Ext}_{k \rightarrow 2k}(T)$$

where π is an involutory permutation on $[2k]$ such that $\pi(i) = k + i$ and $\pi(k + i) = i$ hold for any $i \in [k]$. The Turing machine T' is involutory by Lemma 4.4 since $\text{Perm}(\pi)$ in the middle is involutory.

As for the semantics of T' , we can check the present statement by

$$\begin{aligned} & \llbracket T' \rrbracket(w_1, \dots, w_k, v_1, \dots, v_k) \\ &= \llbracket \text{Ext}_{k \rightarrow 2k}(T)^{-1} \circ \text{Perm}(\pi) \circ \text{Ext}_{k \rightarrow 2k}(T) \rrbracket(w_1, \dots, w_k, v_1, \dots, v_k) \\ &= \llbracket \text{Ext}_{k \rightarrow 2k}(T)^{-1} \rrbracket(\llbracket \text{Perm}(\pi) \rrbracket(\llbracket \text{Ext}_{k \rightarrow 2k}(T) \rrbracket(w_1, \dots, w_k, v_1, \dots, v_k))) \\ &= \llbracket \text{Ext}_{k \rightarrow 2k}(T)^{-1} \rrbracket(\llbracket \text{Perm}(\pi) \rrbracket(\llbracket T \rrbracket(w_1, \dots, w_k), v_1, \dots, v_k)) \\ &= \llbracket \text{Ext}_{k \rightarrow 2k}(T)^{-1} \rrbracket(v_1, \dots, v_k, \llbracket T \rrbracket(w_1, \dots, w_k)) \\ &= (\llbracket T \rrbracket^{-1}(v_1, \dots, v_k), \llbracket T \rrbracket(w_1, \dots, w_k)). \end{aligned}$$

□

Now we show one of the main theorems which states any involutory computable function can be implemented by an involutory Turing machine. For any non-involutory Turing machine, an equivalent involutory Turing machine can be constructed whenever its semantics is involutory. Recall that in our function semantics of Turing machines a function some of whose arguments and results are always empty words is regarded as that with fewer arguments and results as mentioned in the previous section. In the following statement, for a given k -tape non-involutory Turing machine, a $2k$ -tape involutory Turing machine is constructed whose semantics function has k arguments and k outputs that are always empty.

Theorem 4.6 (Expressiveness of involutory Turing machines). The involutory Turing machines can compute any involutory computable function. More specifically, given a k -tape Turing machine T such that $\llbracket T \rrbracket$ is involutory, there is a $2k$ -tape involutory Turing machine T' such that $\llbracket T' \rrbracket = \llbracket T \rrbracket$.

Proof. Let T be a k -tape Turing machine which computes an involution. Since an involution is injective, there exists a k -tape reversible Turing machine T_r such that $\llbracket T_r \rrbracket = \llbracket T \rrbracket$ by Theorem 3.12. Thus we have a $2k$ -tape involutory Turing machine T_i such that $\llbracket T_i \rrbracket(w_1, \dots, w_k, v_1, \dots, v_k) = (\llbracket T_r \rrbracket^{-1}(w_1, \dots, w_k),$

$\llbracket T_r \rrbracket(v_1, \dots, v_k)$ by Lemma 4.5. Consider a $2k$ -tape Turing machine given by $T' = \text{Dup}(k)^{-1} \circ T_i \circ \text{Dup}(k)$ which is involutory by Lemma 4.4. Then,

$$\begin{aligned} \llbracket T' \rrbracket(w_1, \dots, w_k) &= \llbracket \text{Dup}(k) \rrbracket^{-1}(\llbracket T_i \rrbracket(\llbracket \text{Dup}(k) \rrbracket(w_1, \dots, w_k))) \\ &= \llbracket \text{Dup}(k) \rrbracket^{-1}(\llbracket T_i \rrbracket(w_1, \dots, w_k, w_1, \dots, w_k)) \\ &= \llbracket \text{Dup}(k) \rrbracket^{-1}(\llbracket T_r \rrbracket^{-1}(w_1, \dots, w_k), \llbracket T_r \rrbracket(w_1, \dots, w_k)) \end{aligned}$$

holds where Theorem 3.11 and Theorem 3.15 are used. Since $\llbracket T_r \rrbracket = \llbracket T \rrbracket$ is involutory, two k -tuples of the argument of $\llbracket \text{Dup}(k) \rrbracket^{-1}$ are equal. By the definition of $\llbracket \text{Dup}(k) \rrbracket^{-1}$, we have $\llbracket T' \rrbracket(w_1, \dots, w_k) = \llbracket T \rrbracket(w_1, \dots, w_k)$. $\square$

How expressive are multi-tape involutory Turing machines compared to those equipped with fewer tapes? It is well known that a multi-tape Turing machine can be simulated by a single-tape Turing machine with an injective encoding from a tuple of words to a single word. The simulation is called a *tape reduction*, which is also possible for reversible Turing machines [1, 2]. Concerning involutory Turing machines, any multi-tape involutory Turing machine can be simulated by a 2-tape involutory Turing machine whose semantics is a function over words that encode a tuple of words (where one of the tapes is always the empty word in the initial and final configuration).

Theorem 4.7 (k -tape to 2-tape involutory Turing machine). Given a k -tape involutory Turing machine $T = (Q, \Sigma, q_{\text{ini}}, q_{\text{fin}}, \Delta)$, there exists a 2-tape involutory Turing machine T' such that

$$\llbracket T \rrbracket(v_1, \dots, v_k) = (w_1, \dots, w_k) \quad \text{iff} \quad \llbracket T' \rrbracket(\text{enc}(v_1, \dots, v_k)) = \text{enc}(w_1, \dots, w_k)$$

for any $v_1, \dots, v_k, w_1, \dots, w_k \in \Sigma^*$ where $\text{enc} : (\Sigma^*)^k \rightarrow (\Sigma \uplus \{\#\})^*$ is a simple encoding function for tuples with a separator $\# \notin \Sigma$ defined by $\text{enc}(x_1, \dots, x_k) = x_1 \# \dots \# x_k$.

Proof. Consider a function $f : (\Sigma \uplus \{\#\})^* \rightarrow (\Sigma \uplus \{\#\})^*$ such that $f(\text{enc}(v_1, \dots, v_k)) = \text{enc}(w_1, \dots, w_k)$ if and only if $\llbracket T \rrbracket(v_1, \dots, v_k) = (w_1, \dots, w_k)$ for any $v_1, \dots, v_k, w_1, \dots, w_k \in \Sigma^*$. The function f is obviously computable. Since f is injective, there exists a 1-tape reversible Turing machine T_r whose semantics is equivalent to f . Furthermore, since $\llbracket T \rrbracket$ is involutory from Theorem 4.2, f is also involutory. By Theorem 4.6, there exists a 2-tape involutory Turing machine which concludes the statement of this theorem. $\square$

A tape reduction to a single-tape involutory Turing machine will be discussed in Sect. 6.

5 Universality of Involutory Turing Machine

A standard Turing machine is said to be universal if it simulates an arbitrary Turing machine. More exactly, a universal Turing machine takes a pair of words:

one is a Gödel number $\lceil T \rceil$ (as a word) representing a Turing machine T and another is an input word of T . It returns the output word $\llbracket T \rrbracket(x)$. The Gödel numbering $\lceil - \rceil$ is an injective computable function that generates a word representation for a Turing machine. The notion of universality is called classical universality here to distinguish with another universality introduced later. For simplicity, only 1-tape Turing machines are considered which are to be simulated by a universal Turing machine. The results can be easily generalized to multi-tape Turing machines.

Definition 5.1 (Classical universality). A Turing machine U is said to be *classically universal* if $\llbracket U \rrbracket(\lceil T \rceil, x) = \llbracket T \rrbracket(x)$ holds for any Turing machine T and its input word x of T .

No involutory Turing machine is classically universal since the function $\llbracket U \rrbracket$ in the definition above is obviously not involutory. Because the function is not even injective, no universal reversible Turing machine exists. Axelsen and Glück relaxed the definition of universality in a natural way; they showed that there exists a universal reversible Turing machine under the definition. We follow their definition for the universality of involutory Turing machines.

Definition 5.2 (Universality). An involutory Turing machine U is said to be *ITM-universal* if $\llbracket U \rrbracket(\lceil T \rceil, x) = (\lceil T \rceil, \llbracket T \rrbracket(x))$ holds for any involutory Turing machine T and its input word x of T .

The existence of an ITM-universal involutory Turing machine is easily obtained by the completeness of involutory Turing machines shown in Theorem 4.6.

Theorem 5.3. There exists an ITM-universal involutory Turing machine.

Proof. Let f be a function equivalent to the semantics of an involutory Turing machine to be ITM-universal given in Definition 5.2, i.e., $f(\lceil T \rceil, x) = (\lceil T \rceil, \llbracket T \rrbracket(x))$ for any involutory Turing machine T and its input words of T . What we must show is that there exists an involutory Turing machine whose semantics is equivalent to f . Note that f is computable. By Theorem 4.6, it suffices to show that f is involutory. This is verified by $f(f(\lceil T \rceil, x)) = f(\lceil T \rceil, \llbracket T \rrbracket(x)) = (\lceil T \rceil, \llbracket T \rrbracket(\llbracket T \rrbracket(x))) = (\lceil T \rceil, x)$ where the last equality comes from Theorem 4.2. $\square$

Theorem 5.3 only shows the existence of an ITM-universal involutory Turing machine. The proof of the theorem relies on Theorem 3.12 (via Theorem 4.6) which requires an impractical generate-and-test inversion method [2, Lemma 3]. This kind of problem has already been recognized in proving the existence of universal reversible Turing machines by Axelsen and Glück [2]. They gave a solution to that by constructing a universal reversible Turing machine from a classically universal Turing machine. We employ their idea for the direct construction of an ITM-universal involutory Turing machine. Since their construction gives a non-involutory Turing machine, we shall show a different construction using Landauer embedding and Bennett's trick for a classically universal Turing machine.

Proposition 5.4 (Landauer embedding [2,8]). Let $T = (Q, \Sigma, q_{\text{ini}}, q_{\text{fin}}, \Delta)$ be a k -tape Turing machine. Then, there exists a $(k+1)$ -tape reversible Turing machine $\text{Lan}(T)$ such that $\llbracket \text{Lan}(T) \rrbracket(w_1, \dots, w_k, \varepsilon) = (\llbracket T \rrbracket(w_1, \dots, w_k), \text{trace}(T, w_1, \dots, w_k))$ where trace is a function encoding a history of applied rules on the corresponding run into Σ^* .

As mentioned in [2], Bennett's trick gives an input-preserving reversible Turing machine for an arbitrary Turing machine.

Proposition 5.5 (Bennett's trick [3]). Let T be a k -tape Turing machine. Then, there exists a $(2k+1)$ -tape reversible Turing machine $\text{Ben}(T)$ such that $\llbracket \text{Ben}(T) \rrbracket(w_1, \dots, w_k) = (w_1, \dots, w_k, \llbracket T \rrbracket(w_1, \dots, w_k))$.

For the proposition above, a similar proof to that of [2, Lemma 6] does work by $\text{Ben}(T) = \text{Perm}(\pi^{-1}) \circ \text{Ext}_{k+1 \rightarrow 2k+1}(\text{Lan}(T)^{-1}) \circ \text{Perm}(\pi) \circ \text{Ext}_{2k \rightarrow 2k+1}(\text{Dup}(k)) \circ \text{Perm}(\pi^{-1}) \circ \text{Ext}_{k+1 \rightarrow 2k+1}(\text{Lan}(T))$ with permutation $\pi = (k+1 \dots 2k+1)$.

Theorem 5.6 (Universal Turing machine to ITM-universal involutory Turing machine). Let U be a classically universal Turing machine, i.e., $\llbracket U \rrbracket(\ulcorner T \urcorner, x) = \llbracket T \rrbracket(x)$ for any Turing machine T . Then, an involutory Turing machine U' defined by $U' = \text{Ben}(U)^{-1} \circ \text{Perm}((2\ 3)) \circ \text{Ben}(U)$ is ITM-universal.

Proof By Proposition 5.4 and Proposition 5.5, we obtain $\llbracket \text{Ben}(U) \rrbracket(\ulcorner T \urcorner, x) = (\ulcorner T \urcorner, x, \llbracket T \rrbracket(x))$ for any Turing machine $\ulcorner T \urcorner$ and its input x . Let T be an involutory Turing machine, i.e., $\llbracket T \rrbracket(\llbracket T \rrbracket(x)) = x$ by Theorem 4.2. Thus we have

$$\begin{aligned} \llbracket U' \rrbracket(\ulcorner T \urcorner, x) &= \llbracket \text{Ben}(U) \rrbracket^{-1}(\llbracket \text{Perm}((2\ 3)) \rrbracket(\llbracket \text{Ben}(U) \rrbracket(\ulcorner T \urcorner, x))) \\ &= \llbracket \text{Ben}(U) \rrbracket^{-1}(\llbracket \text{Perm}((2\ 3)) \rrbracket(\ulcorner T \urcorner, x, \llbracket T \rrbracket(x))) \\ &= \llbracket \text{Ben}(U) \rrbracket^{-1}(\ulcorner T \urcorner, \llbracket T \rrbracket(x), x) \\ &= \llbracket \text{Ben}(U) \rrbracket^{-1}(\ulcorner T \urcorner, \llbracket T \rrbracket(x), \llbracket T \rrbracket(\llbracket T \rrbracket(x))) \\ &= (\ulcorner T \urcorner, \llbracket T \rrbracket(x)) \end{aligned}$$

which indicates the ITM-universality of U' . □

6 Discussion

This section describes the design choice of permutation rules and the limitation of tape reduction for our involutory Turing machines. The applications of our computational model will also be discussed.

Design Choice. One may feel it unusual for the definition of a multi-tape Turing machine to have tape permutation rules. A k -tape Turing machine is typically defined by specifying a set of transition rules as

$$\Delta \subseteq (Q \setminus \{q_{\text{fin}}\}) \times ((\Sigma_{\perp} \times \Sigma_{\perp}) \cup \{\leftarrow, \diamond, \rightarrow\})^k \times (Q \setminus \{q_{\text{ini}}\}).$$

The typical definition forces one to rewrite or move at all tapes simultaneously¹. An involutory Turing machine could be defined on this model with the same restriction $\tilde{\varphi}(\Delta) = \Delta$ as ours. It is easy to see that the model always computes an involution by a similar proof to that of Theorem 4.2. However, it is not easy to show the expressiveness like Theorem 4.6 for the model without permutation rules. Although the author believes but cannot prove that the expressiveness does not hold without permutation rules, the readers may surmise it intuitively by trying to define a 2-tape Turing machine $\text{Perm}((1\ 2))$ that swaps an input pair. The simplest way to do this is first to swap each from left to right and then return the head back to the left end. Note that it does not conform to the restriction above to be involutory. The reversed run is no longer valid for any renaming states. There might exist a tricky way to implement the swap function without permutation rules. The author speculates that it is impossible, though.

Limitation of Tape Reduction. Theorem 4.7 states that any k -tape involutory Turing machine can be simulated by a 2-tape involutory Turing machine. A natural question is whether it is possible by a 1-tape involutory Turing machine. Again, the author believes it is impossible. Consider a 1-tape Turing machine T_{bnot} in Example 3.6, which is not involutory but whose semantics is involutory. By Theorem 4.6, there exists a 2-tape involutory Turing machine equivalent to T_{bnot} . It is found hard to define a 1-tape involutory Turing machine for a similar reason to the observation above on defining the swap function without permutation rules. The author speculates the reduction to a single tape is impossible in general but leaves the proof for future work as well.

Application to Bidirectional Transformation. This work is originally motivated by characterizing programming languages for bidirectional transformation, which enables us to synchronize multiple data and maintain their consistency. A typical bidirectional transformation is specified by a pair of forward ($\text{get} : S \rightarrow V$) and backward ($\text{put} : S \times V \rightarrow S$) transformations over sources S and views V where two functions are required to be consistent in some sense. More specifically, the following three laws are to be satisfied for the consistency (called *very-well-behavedness* in [4]):

$$\text{put}(s, \text{get}(s)) = s \quad \text{get}(\text{put}(s, v)) = v \quad \text{put}(\text{put}(s, v), v') = \text{put}(s, v')$$

for any $s \in S$ and $v, v' \in V$. Some bidirectional programming languages impose a syntactic restriction to programmers so that it conforms to the consistency. However, due to the restriction, it is hard to characterize how powerful each programming language is. As expressiveness of general programming languages can be characterized by Turing machines, we need a computational

¹ The union symbol $\cup$ may be replaced with $\times$ so that a read/write and a move action may happen at the same time. However, this makes it hard to give the inverse of a pair of actions, which is often required in the present work.

model to characterize all pairs of computable functions *get* and *put* that satisfy consistency.

An involutory Turing machine gives a partial solution to this problem because of its expressiveness. Consistency implies that a function $f : S \times V \rightarrow S \times V$ defined by $f(s, v) = (\text{put}(s, v), \text{get}(s))$ is involutory, that is, $f(f(s, v)) = (s, v)$ holds. Thereby any correct bidirectional program can be computed by a 2-tape involutory Turing machine under some encoding. Conversely, a 2-tape involutory Turing machine does not always specify a correct bidirectional program. The author believes that there exists a computational model given by an appropriately-restricted involutory Turing machine, which exactly covers all consistent bidirectional transformations. The present work will be a good starting point to characterize bidirectional programming languages.

Program Inversion. A program inverter *pinv* is a function that takes the Gödel number of a Turing machine T and returns that of a Turing machine whose semantics is the inverse of $\llbracket T \rrbracket$. i.e., $\text{pinv}(\ulcorner T \urcorner) = \ulcorner T \urcorner^{-1}$ holds. Axelsen and Glück have shown that there exists a 2-tape reversible Turing machine that computes *pinv*. They employ the result as a lemma to construct a universal reversible Turing machine from a classically universal Turing machine. Although we do not need such a lemma for the construction of a universal involutory Turing machine, the same statement also holds for the involutory Turing machine.

Theorem 6.1 (Program inverter as an involutory Turing machine).

There exists a 2-tape involutory Turing machine T which computes a program inverter for reversible Turing machines, that is, $\llbracket T \rrbracket(\ulcorner R \urcorner) = \ulcorner R^{-1} \urcorner$ holds for any reversible Turing machine R .

Proof. There exists a 1-tape reversible Turing machine T_r such that $\llbracket T_r \rrbracket(\ulcorner R \urcorner) = \ulcorner R^{-1} \urcorner$ by [2, Lemma 15]. Since $\llbracket T_r \rrbracket$ is involutory, there exists a 2-tape involutory Turing machine whose semantics is equivalent to $\llbracket T_r \rrbracket$ by Theorem 4.6. $\square$

7 Related Work

This work is strongly inspired by Axelsen and Glück's [2]. They showed that (1) any injective computable function can be implemented by a reversible Turing machine, (2) the number of tapes of a reversible Turing machine can be reduced with preserving its semantics, (3) there exists a universal reversible Turing machine, and (4) a universal reversible Turing machine can be constructed from a universal Turing machine. In the present paper, we have addressed these properties for involutory Turing machines instead of reversible Turing machines. Regarding (1), (3), and (4), similar results have been obtained. Interestingly, their proofs are rather different from those of the corresponding theorems for reversible Turing machines. Regarding (2), we failed to find an equivalent single-tape involutory Turing machine for an arbitrary multi-tape involutory Turing machine. This might be an inherent limitation of involutory Turing machines as we have discussed in Sect. 6.

An involutory Turing machine may remind some readers of a symmetric Turing machine introduced by Lewis and Papadimitriou [9], where its computation step is symmetric. A symmetric Turing machine can be considered as a variant of an involutory Turing machine whose state involution is the identity function by ignoring the requirement of $\varphi(q_{\text{ini}}) = q_{\text{fin}}$ and $\varphi(q_{\text{fin}}) = q_{\text{ini}}$ ². However, symmetric Turing machines give a computational model completely different from involutory Turing machines. A symmetric Turing machine defines an undirected graph specified by its configurations and computation steps; its run is a path from the initial one to the final one. This model has been introduced to specify the computational complexity of the undirected st-connectivity (USTCON) problem and is known to be at least as powerful as a deterministic Turing machine.

Gajardo, Kari, and Moreira [5] introduced *time symmetry* for cellular automata to import a notion in physical theories where forward and backward time directions cannot be distinguished. The time symmetry is specified by an involution which connects the corresponding states and transitions as involutory Turing machines do. Kutrib and Worsch [7] later ported the notion of time symmetry into finite automata and pushdown automata. Involutory Turing machines may be called *time-symmetric Turing machines* along this line. Nevertheless, our computational model should be called an involutory Turing machine in the present paper owing to our purpose to characterize all computable involutions unlike the previous time-symmetric machines: Gajardo et al.'s time-symmetric cellular automata compute a composition of two involutions but not a single involution. Although it would be worthwhile to investigate the time symmetry and other related properties of involutory Turing machines, they are left for future work.

8 Conclusion

A computational model for involution has been presented. The model is a variant of a multi-tape Turing machine, called an involutory Turing machine, which imposes a restriction on state transition rules so that the reversed run of every valid run should be valid. The model has been shown to be expressive enough in the sense that not only does an involutory Turing machine always compute an involution but also any involution can be computed by an involutory Turing machine. It has also been shown that there exists a universal involutory Turing machine that can simulate an arbitrary involutory Turing machine.

This work naturally introduces a notion of *i*-Turing completeness, i.e., a programming language L is said to be *i-Turing complete* if every program in L defines an involution and every involutory Turing machine can be simulated by a program in L . A similar notion called *r*-Turing completeness [1, 2] has been employed for characterizing reversible programming languages [6, 11]. As dis-

² Although the definition of a symmetric Turing machine in [9] differs from ours in another point that it allows transition rules to view the next cell of the head, this cannot change its expressiveness as mentioned in the paper.

cussed in Sect. 6, *i*-Turing completeness or its variant will be used to characterize bidirectional programming languages in the future.

Acknowledment. I am grateful to Robert Glück who has kindly lectured to me about reversible Turing machines and their expressiveness. I also thank Kanae Tsushima for her observation on the involutoriness of bidirectional transformation and Mirai Ikebuchi for carefully proofreading the manuscript. Furthermore, I want to appreciate anonymous reviewers' fruitful comments on a close connection with time-symmetric machines. This work was partially supported by JSPS KAKENHI Grant Numbers JP17K00007, JP18H03204, and JP18H04093.

References

1. Axelsen, H.B., Glück, R.: What do reversible programs compute? In: Hofmann, M. (ed.) FoSSaCS 2011. LNCS, vol. 6604, pp. 42–56. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19805-2_4
2. Axelsen, H.B., Glück, R.: On reversible turing machines and their function universality. *Acta Inf.* **53**(5), 509–543 (2016). <https://doi.org/10.1007/s00236-015-0253-y>
3. Bennett, C.H.: Logical reversibility of computation. *IBM J. Res. Dev.* **17**(6), 525–532 (1973)
4. Foster, N., Matsuda, K., Voigtländer, J.: Three complementary approaches to bidirectional programming. In: Gibbons, J. (ed.) *Generic and Indexed Programming*. LNCS, vol. 7470, pp. 1–46. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32202-0_1
5. Gajardo, A., Kari, J., Moreira, A.: On time-symmetry in cellular automata. *J. Comput. Syst. Sci.* **78**(4), 1115–1126 (2012). <https://doi.org/10.1016/j.jcss.2012.01.006>
6. Glück, R., Yokoyama, T.: A linear-time self-interpreter of a reversible imperative language. *Comput. Softw.* **33**(3), 3_108–3_128 (2016)
7. Kutrib, M., Worsch, T.: Time-symmetric machines. In: Dueck, G.W., Miller, D.M. (eds.) *RC 2013*. LNCS, vol. 7948, pp. 168–181. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38986-3_14
8. Landauer, R.: Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.* **5**(3), 183–191 (1961). <https://doi.org/10.1147/rd.53.0183>
9. Lewis, H.R., Papadimitriou, C.H.: Symmetric space-bounded computation. *Theor. Comput. Sci.* **19**, 161–187 (1982)
10. Sipser, M.: *Introduction to the theory of computation*. PWS Publishing Company, Boston (1997)
11. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: *Proceedings of the 5th Conference on Computing Frontiers*, Ischia, Italy, 5–7 May 2008. pp. 43–54 (2008)
12. Zagier, D.: A one-sentence proof that every prime $p \equiv 1 \pmod{4}$ is a sum of two squares. *Am. Math. Monthly* **97**(2), 144 (1990)