A Simple and Efficient Universal Reversible Turing Machine

Holger Bock Axelsen and Robert Glück

DIKU, Department of Computer Science, University of Copenhagen, Denmark
funkstar@diku.dk, glueck@acm.org

Abstract. We construct a universal reversible Turing machine (URTM) from first principles. We take a strict approach to the semantics of reversible Turing machines (RTMs), under which they can compute exactly all *injective*, computable functions, but not non-injective ones. The natural notion of universality for RTMs is *RTM-universality*, where programs are considered part of both input and output of a URTM.

The machine described here is the first URTM which does not depend on reversibilizing any existing universal machine. The interpretive overhead of the URTM is limited to a (program dependent) constant factor slowdown, with no other complexity-wise cost *wrt* time and space. The URTM is also able to function as an *inverse interpreter* for RTMs at no asymptotic cost, simply by reversing the string representing the interpreted machine.

1 Introduction

Reversible computation models are time-invertible, forward and backward deterministic. For stateful computation models this means that not only the next, but also the previous state is uniquely defined at all times. There is a wide range of reversible computation models, from cellular automata [9] and pushdown automata [7], over logic circuits [16,13] to quantum computing [5,14]. Reversible computing principles also finds use in program transformations such as inversion [15], reversible programming [3,17] and translation [1], and bidirectional model transformation [12,6].

Here, we are concerned with the foundational question of *universality* for reversible Turing machines (RTMs.) Universality is the (semantic) notion of a specific machine being able to perform any computation possible within a computation model. In programming languages, the equivalent notion is that of a *self-interpreter*.

Previous work by Morita *et al.* exists asserting the universality (in the sense of full Turing completeness) of an RTM [11]. On close examination, their construction implicitly allows for a semantical relaxation with respect to what function is being computed. Specifically, one is allowed to *extract* part of the tape and consider it the output of the computation, whereby information is irreversibly lost.

In recent work [2] we studied the RTMs under the stricter viewpoint that the *entire* configuration must be considered for the output. This decouples the

A.-H. Dediu, S. Inenaga, and C. Martín-Vide (Eds.): LATA 2011, LNCS 6638, pp. 117–128, 2011.

semantic functionality of RTMs from any particular transformation, such as Bennett's method [4]. However, under this interpretation reversibility of a machine implies injectivity of the computed function. The RTMs are then *not* fully Turing complete: They cannot simulate irreversible Turing machines without changing their functional behavior, and in fact cannot even faithfully simulate all *reversible* Turing machines. To amend this, we introduced a more natural notion of universality for reversible Turing machines, *RTM-universality*: An RTM is RTM-universal (is a URTM) if it can simulate any RTM while also remembering the simulated machine's program text.

Here, for the first time, we show how to construct a 3-tape URTM from first principles. The URTM is efficient in the sense that the asymptotic complexity of the interpreted RTM is preserved. The interpretive overhead is limited to a program dependent constant time factor, and there is no change in the space behavior except for adding a short string encoding the simulated internal state. Furthermore, the URTM can function as an inverse interpreter at *no* asymptotic cost. This is the first demonstration of a URTM with such properties.

2 Reversible Turing Machines

We here define the reversible Turing machines (RTMs). We state only results and properties relevant here. For a more complete exposition, see [2,4].

Definition 1 (Turing machine). *A TM T is a tuple $(Q, \Sigma, \delta, b, q_s, q_f)$ where Q is a finite set of states, Σ is a finite set of tape symbols, $b \in \Sigma$ is the blank symbol,*

$$\delta \subseteq \Delta = (Q \times [(\Sigma \times \Sigma) \cup \{\leftarrow, \downarrow, \rightarrow\}] \times Q)$$

is a partial relation defining the transition relation, $q_s \in Q$ is the starting state, and $q_f \in Q$ is the final state. There must be no transitions leading out of q_f nor into q_s. Symbols $\leftarrow$, $\downarrow$, $\rightarrow$ represent the three shift directions (left, stay, right).

Note that we use a *triple format* for the transition relation, with two kinds of rule. A *symbol rule* $(q, (s, s'), q') \in \delta$ says that in state q, if the tape head is reading symbol s, write s' and change into state q'. A *move rule* $(q, d, q') \in \delta$ says that in state q, move the tape head in direction d and change into state q'. This is easily extended to k-tape machines, where we have

$$\Delta = (Q \times [(\Sigma \times \Sigma)^k \cup \{\leftarrow, \downarrow, \rightarrow\}^k] \times Q) \ .$$

The *configuration* of a TM is a tuple $(q, (l, s, r)) \in Q \times (\Sigma^* \times \Sigma \times \Sigma^*)$, where q is the internal state, $l, r \in \Sigma^*$ are the left and right parts of the tape (as strings), and $s \in \Sigma$ is the current symbol being scanned. A TM T in configuration C *leads to* configuration C', written as $T \vdash C \rightsquigarrow C'$, in a single *computation step* in the obvious manner defined by the transition relation.

Definition 2 (Local fwd/bwd determinism). *A TM $T = (Q, \Sigma, \delta, b, q_s, q_f)$ is* local forward deterministic *iff for any distinct pair of triples $(q_1, a_1, q_1') \in \delta$ and*

$(q_2, a_2, q_2') \in \delta$, *if* $q_1 = q_2$ *then* $a_1 = (s_1, s_1')$ *and* $a_2 = (s_2, s_2')$, *and* $s_1 \neq s_2$. *A TM* T *is* local backward deterministic *iff for any distinct pair of triples* $(q_1, a_1, q_1') \in \delta$ *and* $(q_2, a_2, q_2') \in \delta$, *if* $q_1' = q_2'$ *then* $a_1 = (s_1, s_1')$ *and* $a_2 = (s_2, s_2')$, *and* $s_1' \neq s_2'$.

We say a TM is *reversible* iff it is locally forward and backward deterministic.

As examples, the rules $(\mathtt{q}, (\mathtt{a}, \mathtt{b}), \mathtt{p})$ and $(\mathtt{q}, (\mathtt{a}, \mathtt{c}), \mathtt{p})$ respect backward determinism (but not forward determinism); rules $(\mathtt{q}, (\mathtt{a}, \mathtt{b}), \mathtt{p})$ and $(\mathtt{r}, (\mathtt{c}, \mathtt{b}), \mathtt{p})$ are not backward deterministic; and neither are $(\mathtt{q}, (\mathtt{a}, \mathtt{b}), \mathtt{p})$ and $(\mathtt{r}, \rightarrow, \mathtt{p})$.

The semantic function of a TM is defined by its input/output behavior on standard configurations. A TM is in *standard configuration* iff the tape head is to the immediate left of a finite, blank-free string $s \in (\Sigma \backslash \{b\})^*$, and the rest of the tape is blank, i.e., it is in configuration $(q, (\varepsilon, b, s))$ for some state q.[1]

Definition 3 (String transformation semantics). *The* semantics $[\![T]\!]$ *of a TM* $T = (Q, \Sigma, \delta, b, q_s, q_f)$ *is given by the relation*

$$[\![T]\!] = \{(s, s') \in ((\Sigma \backslash \{b\})^* \times (\Sigma \backslash \{b\})^*) \mid T \vdash (q_s, (\varepsilon, b, s)) \rightsquigarrow^* (q_f, (\varepsilon, b, s'))\} \ .$$

A computation with a TM is as follows: From starting state q_s with input s in standard configuration $(q_s, (\varepsilon, b, s))$, run the machine until it halts in standard configuration $(q_f, (\varepsilon, b, s'))$ with output s', or diverges. We say T *computes* function f iff $[\![T]\!] = f$. Under this semantics, the reversibility of individual steps leads directly to injectivity in terms of functional behavior.

Theorem 1 (RTMs are injective [2]). *If* T *is an RTM, then* $[\![T]\!]$ *is an injective function.*

Lemma 1 (RTM inversion, Bennett [4]). *Given an RTM* $T = (Q, \Sigma, \delta, b, q_s, q_f)$, *the RTM* $T^{-1} \stackrel{\text{def}}{=} (Q, \Sigma, inv(\delta), b, q_f, q_s)$ *computes the inverse function of* $[\![T]\!]$, *i.e.* $[\![T^{-1}]\!] = [\![T]\!]^{-1}$, *where* $inv : \Delta \rightarrow \Delta$ *is defined as*

$$\begin{aligned} inv(q, (s, s'), q') &= (q', (s', s), q) & inv(q, \leftarrow, q') &= (q', \rightarrow, q) \\ inv(q, \downarrow, q') &= (q', \downarrow, q) & inv(q, \rightarrow, q') &= (q', \leftarrow, q) \ . \end{aligned}$$

This means that an RTM can be inverted into another RTM very easily.

Theorem 2 (Bennett's method [4]). *Given a 1-tape TM* T, *there exists a 3-tape RTM* $B(T)$, *s.t.* $[\![B(T)]\!](x) = (x, [\![T]\!](x))$.

This seminal result states that any TM can be *reversibilized*, *i.e.* turned into an RTM. It is important to note that Bennett's method does *not* not preserve semantics, $[\![T]\!] \neq [\![B(T)]\!]$, as the output of $B(T)$ includes the input x to T.[2]

Theorem 3 (Expressiveness [4,2]). *The RTMs can compute* exactly all *injective computable functions. That is, for every 1-tape TM* T *such that* $[\![T]\!]$ *is an injective function, there is a 3-tape RTM* T' *such that* $[\![T]\!] = [\![T']\!]$.

[1] The empty string ε denotes the infinite string of blanks b^ω, and is usually omitted.

[2] Output values that are added to ensure reversibility are known as *garbage* in reversible computing.

Thus, even though Theorem 1 restricts RTMs to injective (computable) functions, *all* of these are nevertheless in scope. Finally, we only need 1 tape and 3 symbols for any RTM computation.

Theorem 4 (Robustness [2]). *Let T be a k-tape, m-symbol RTM. Then there exists a 1-tape, 3-symbol RTM T' s.t.*

$$[\![T]\!](x_1, \ldots, x_k) = (y_1, \ldots, y_k) \quad \textit{iff} \quad [\![T']\!](e(\langle x_1, \ldots, x_k \rangle)) = e(\langle y_1, \ldots y_k \rangle)\,,$$

where $\langle \cdot \rangle$ is the convolution of tape contents, and $e(\cdot)$ is a binary encoding.

This is used to simplify our construction of a universal RTM, below.

2.1 Universality for RTMs

A universal machine is a machine that can simulate the (functional) behavior of any other machine. Usually, a universal TM U is defined as a *self-interpreter* for Turing machines:

$$[\![U]\!](\ulcorner T \urcorner, x) = [\![T]\!](x)\ .$$

Here, $\ulcorner T \urcorner \in \Sigma^*$ is a Gödel number (or *program text*) representing some TM T. However, $[\![U]\!]$ is a non-injective function (even if T has to be an RTM), so the RTMs are not universal in the classical sense. In [2], the authors therefore argued to define universality for the RTMs as follows.

Definition 4 (RTM-universality [2]). *An RTM U_R is* RTM-universal *(or, a* URTM*) iff for all RTMs T and all (blank-free) inputs $x \in \Sigma^*$,*

$$[\![U_R]\!](\ulcorner T \urcorner, x) = (\ulcorner T \urcorner, [\![T]\!](x))\ .$$

Theorem 5 (URTM existence [2]). *There exists an RTM U_R, such that U_R is RTM-universal.*

This follows directly from the expressiveness of the RTMs, Theorem 3.

3 A First-Principles URTM

We shall now describe the design and inner workings of a URTM constructed from first principles.

Besides novelty, our main motivation for constructing such a machine is to avoid any use of reversibilization and to limit the interpretive overhead to a constant factor. Nothing forces us to mechanically reversibilize an irreversible (classically universal or RTM-universal) machine when constructing a URTM, and reversibilizations come with considerable drawbacks. Most importantly, they change the asymptotic complexities of the programs. For instance, the URTM of Theorem 5 [2] relies on Bennett's trick [4], which means that the URTM uses as much (temporary) space as time, regardless of the space usage of the interpreted

program. This change in complexity is a highly undesirable side effect of using reversibilization.[3]

Such asymptotic inefficiency is not necessary: A URTM can (and should) conserve the asymptotic complexities of the interpreted machines (up to a program dependent constant.) However, no URTM with such properties has been exhibited until now.

Structure and Scope. We know that the 1-tape, 3-symbol RTMs are sufficient to express every injective computable function. We can therefore restrict our URTM to interpret exactly these machines. (We shall refer to the interpreted machine as T.)

We aim for a "textbook" structure for our URTM, with three tapes:

1. The first *work tape* will directly correspond to T's single tape.
2. The second *program tape* will hold the program text $\ulcorner T \urcorner$ (described below.)
3. The third *state tape* (blank at start and halt) will hold the encoding of the internal state of the interpreted machine T during the simulation (using the same encoding used for states in $\ulcorner T \urcorner$.)

Our focus here is on simplicity and efficiency, rather than minimization, so it shall not concern us that the URTM has a large number of states and symbols. However, for the same reason we shall not show the complete rule table here.

Program Encoding. We encode the interpreted 1-tape, 3-symbol machine $T = (Q, \{b, \mathtt{0}, \mathtt{1}\}, \delta, b, q_s, q_f)$ by a program text $\ulcorner T \urcorner$ as follows: The program is a string listing the rules in δ. The first rule in this program text *must* be the single rule that leaves the starting state q_s, and the final rule *must* be the single rule that enters the halting state q_f. With rules individually translated as below, such a string is sufficient to uniquely specify T.[4]

For the actual string $\ulcorner T \urcorner$ we use the alphabet $\Sigma = \{b, \mathtt{0}, \mathtt{1}, \mathtt{B}, \mathtt{S}, \mathtt{M}, \mathtt{\#}\}$. The two rule types (symbol and move) are translated by

$$\begin{aligned} \mathit{trans}(q, (s, s'), q') &= \mathtt{S\#}\mathrm{enc}_Q(q)\mathtt{\#}\mathrm{enc}_\Sigma(s)\mathrm{enc}_\Sigma(s')\mathtt{\#}\mathrm{rev}(\mathrm{enc}_Q(q'))\mathtt{\#S} \\ \mathit{trans}(q, d, q') &= \mathtt{M\#}\mathrm{enc}_Q(q)\mathtt{\#}\mathrm{enc}_D(d)\mathtt{\#}\mathrm{rev}(\mathrm{enc}_Q(q'))\mathtt{\#M}\ , \end{aligned}$$

where $\mathrm{enc}_Q : Q \to \{\mathtt{0},\mathtt{1}\}^{\lceil \log |Q| \rceil}$ is some injective binary encoding of the states in Q, and where

$$\mathrm{enc}_\Sigma(s) = \begin{cases} \mathtt{B} & \text{if } s = b \\ s & \text{otherwise}\ , \end{cases} \qquad \mathrm{enc}_D(d) = \begin{cases} \mathtt{10} & \text{if } d = \leftarrow \\ \mathtt{BB} & \text{if } d = \downarrow \\ \mathtt{01} & \text{if } d = \rightarrow\ , \end{cases}$$

encode the 3 symbols of T and the possible directions. Finally, $\mathrm{rev}(\cdot)$ simply reverses a string. The use of the special symbol $\mathtt{B}$ rather than the actual blank

[3] In addition to the functional redundancy of conserving the inputs, this complexity-wise inefficiency is a reason for discarding Bennett's suggestion of $B(U)$ as a universal machine.

[4] Note that a specific T may have more than one representation, as δ is unordered.

symbol b ensures that the encoding of a given machine will be a blank-free string that can be given in standard configuration form. As an example, the simple RTM $T = (\{q_0, q_1, q_2, q_3\}, \{b, \mathtt{0}, \mathtt{1}\}, \delta, b, q_0, q_3)$, with transition relation

$$\delta = \{(q_0, \rightarrow, q_1), (q_1, (\mathtt{0}, \mathtt{1}), q_2), (q_1, (\mathtt{1}, \mathtt{0}), q_2), (q_2, \leftarrow, q_3)\} \ ,$$

can be represented by program text

$$\ulcorner T \urcorner = \mathtt{M\#00\#01\#10\#MS\#01\#01\#01\#SS\#01\#10\#01\#SM\#10\#10\#11\#M} \ ,$$

where enc_Q is given by $q_0 \mapsto \mathtt{00}$, $q_1 \mapsto \mathtt{01}$, $q_2 \mapsto \mathtt{10}$ and $q_3 \mapsto \mathtt{11}$.

This encoding has the great advantage that we can perform program inversion by simply reversing the string, $\mathrm{rev}(\ulcorner T \urcorner) = \ulcorner T^{-1} \urcorner$. (String reversal can be performed in linear time by a simple 2-tape RTM.)

URTM Program. The URTM program has the following overall structure.

1. Copy the starting state q_s (first state of first rule) onto the state tape.
2. Sequentially try to apply each rule on the program tape, from left to right.
3. When all rules have been tried, compare the halting state q_f (last state of last rule) with the encoding of the current state q_c (on the state tape). If identical, clear q_c reversibly, rewind the program tape, and halt.
4. If q_c and q_f are not identical, rewind the program tape head and go to 2.

Thus, the URTM program consists of two nested loops: an inner loop where we try to apply each of the rules in the interpreted program to the current simulated configuration, and an outer loop where we test for the halting condition.

This is a straightforward and well-known design for a universal machine. The difficulty lies in that it now has to be done completely reversibly, which poses considerable challenges. For instance, there is control flow confluence at step 2, which is a source of backward non-determinism. In this case the situation is resolved by testing the equality of the simulated state with the starting state: They are only equal at the start of the simulation (*i.e.*, if we came from step 1), and (by definition) must be different thereafter (*i.e.*, if we came from step 4). This is closely related to the use of entry assertions in loops in reversible programming [17].

Fig. 1 shows a state diagram of the program: nodes are states, and edges are the actions of the associated rules. We have used a notation mixing ordinary state diagrams with reversible flowcharts [18]. The *diamond* (test) and *circle* (assertion) with inscribed expressions are reversible control flow operators (CFOs.) The expression of the assertion must be true when entering from the branch marked "true", and false if entering from the branch marked "false". The CFOs are here used as shorthand for RTM programs for comparing strings, as described below.

String Comparison. A central functionality that we rely on throughout the machine is string comparison. We must continuously compare the encoding of the current state (on the state tape) with the encodings of states in the rules

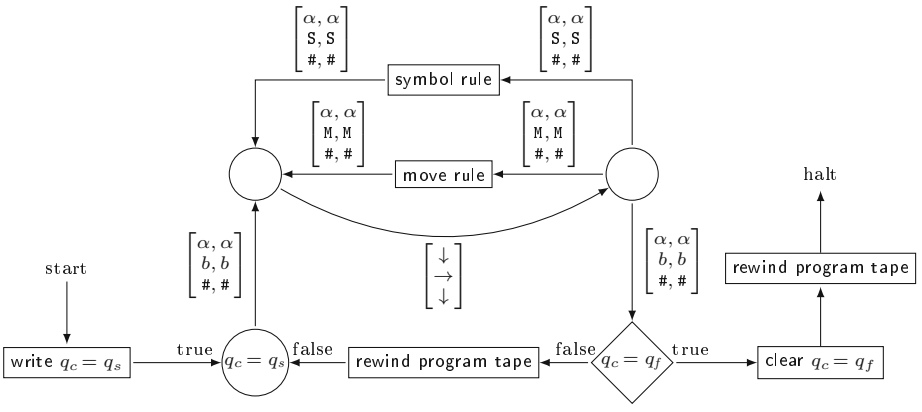


Fig. 1. Overall URTM program structure. The inner loop (topmost, written in ordinary state diagram form) sequentially tries to apply each rule on the program tape, and the outer loop (bottommost, written with shorthand reversible flowchart notation) tests for the halting (and starting) conditions. The symbol rule and move rule phases test for and apply the current rule on the program tape, passing over it in process. The write and clear phases initialize and clear the state tape. Edges with the symbolic variable α encode for three different transitions, with $\alpha \in \{b, 0, 1\}$.

of the interpreted machine (on the program tape). How does this work in the reversible setting?

Assume that we want to check strings $\#s_1 \dots s_n\#$ and $\#t_1 \dots t_n\#$ (each on a separate tape) for equality, moving the tape heads from the starting # to the terminating # in the process. As usual, we can scan the cells from left to right until either the termination symbol # is reached, or a mismatch is found. If the rightmost # symbol is reached without a mismatch, then the strings are equal (and this information is stored in the internal state). If there is a mismatch, however, then we cannot simply pass directly over the rest of the string, like one would do in the irreversible case.[5] Instead, we rewind until the starting # and then pass obliviously over both strings until the string terminator is found. Fig. 2 shows a transition diagram for this functionality. Note that this is done without writing anything to any of the tapes. Also note that the comparison still takes only linear time, the same as the irreversible case.

A central insight from reversible flowcharts tells us that the inverse of a *test* (a conditional split in control flow) is an *assertion* (a conditional join of control flow) [18]. Thus, if we invert the transition diagram in Fig. 2 as specified in Lemma 1, we get an RTM program that can *merge* two control branches if we know two strings are equal in one branch, and different in the other. We did this in the overall URTM program in Fig. 1.

[5] The first mismatch is special, in that the prefixes preceding it are equal. If we only do a single pass of the strings, then the resulting machine cannot be reversible: In reverse mode, it would have to predict exactly when the prefixes are equal without having visited them which is clearly impossible.

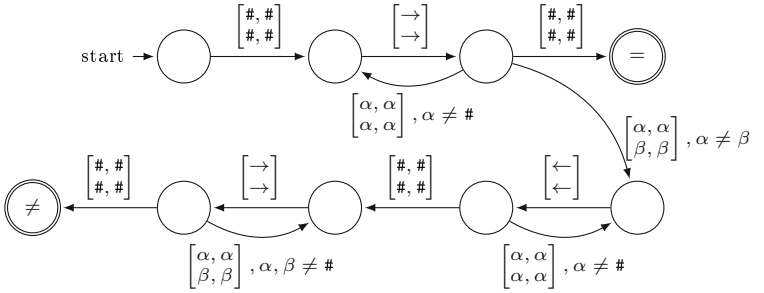


Fig. 2. 2-tape RTM state transition diagram for comparing strings $\#s_1 s_2 \ldots s_n\#$ and $\#t_1 t_2 \ldots t_n\#$ reversibly. (The reader is encouraged to verify the reversibility of the transitions.) The tape heads start at the left # and end at the right #. Edges with symbolic variables α and β encode multiple rules, subject to side conditions (e.g., that $\alpha \neq \#$).

Testing and Applying a Rule. We can distinguish symbol and move rules by their two encompassing `S` or `M` symbols, so we have separate subroutines for each case. We shall here only show the more involved symbol rule.

Application of a given transition rule $(q, (s, s'), q')$ given current state q_c and current symbol s_c is done as follows: We first compare states q and q_c. If they match, we compare symbols s and s_c. If these also match, we apply rule, *i.e.*, perform the substitution specified by the rule, changing the current state to q' and the current symbol to s'. (Appendix A shows the straight-line state transition diagram for this.)

This means that there are three distinct branches for the control flow. An irreversible machine would be able to simply merge these directly at the exit. This is *not* an option in the reversible setting, as it breaks backward determinism. We solve this problem as follows.

We have two disjoint branches where the rule was not applicable: One with a state mismatch $q \neq q_c$, and one where the states matched $q = q_c$ but the tape symbols were different $s \neq s_c$. Thus, we can reversibly merge the control flow of the two possible failures by comparing the current state to the source state of the symbol rule, using the inverted string comparator above.

The key problem is to merge the branch where the rule was applied to the one where it was not. For this we exploit the reversibility of the interpreted machine. Specifically, *only* if rule $(q, (s, s'), q')$ was just applied to cause the step $T \vdash (q, (l, s, r)) \rightsquigarrow (q', (l, s', r))$ can the current state and symbol be q' and s' simultaneously: reversibility guarantees it. With this insight, we can perform essentially the inverse of the testing we did for applicability to merge the branches.

A reversible flowchart for testing and application of a symbol rule is shown in Fig. 3. We have left out a few unimportant details (such as moving the program tape head across string terminators, etc.) Note the use of (reversible) string comparison for both splits and joins in control flow.

Since there is a fixed number of string comparisons, the symbol rule subroutine only takes time linear in the size of the rule, and importantly does not need to inspect any of the other rules in the program. So, even though reversibility is

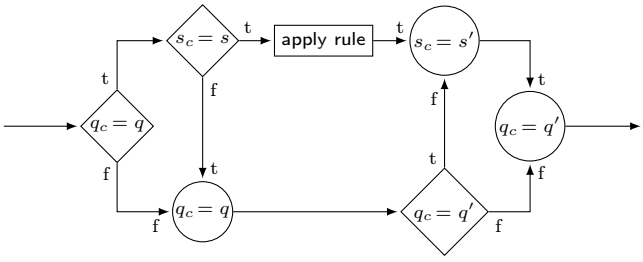


Fig. 3. Reversible flowchart for application of a symbol rule $(q, (s, s'), q')$, given current state q_c and current symbol s_c. The string comparisons are (implicitly) organized such that the program tape head moves from the left `S` in the encoding of the rule to the right `S`.

a *global* property of a program, we can still rely on it for the application of individual rule as a *peephole* property.[6] Note also that the sole change to the tape contents is the substitution of symbols on the work tape, and possibly an update of the state tape.

The case of the move rule is similar.

Asymptotic Complexity. In each pass over the program text at least one rule is applied and the work and state tapes are updated. This means that the interpretation cost in terms of time behavior is a slowdown proportional to the size of the program. A similar program dependent slowdown is also seen for irreversible UTMs. With respect to space, the URTM completely follows the interpreted machine, with only the addition of the string encoding the current simulated state (dominated by the size of the program).

Given a specific program $\ulcorner T \urcorner$, the URTM thus conserves the asymptotic complexities of the machine T.

Inverse Interpretation. As mentioned, our chosen encoding allows for extremely simple program inversion, $\mathrm{rev}(\ulcorner T \urcorner) = \ulcorner T^{-1} \urcorner$. This means that we can use the URTM for reversible *inverse interpretation*: Simply apply string reversal to the program before and after running the URTM. Let R_1 be an RTM that reverses its first argument, $R_1(x, y) = (\mathrm{rev}(x), y)$. We have that

$$[\![R_1 \circ U \circ R_1]\!] (\ulcorner T \urcorner, y) = (\ulcorner T \urcorner, [\![T^{-1}]\!](y)) \ .$$

This completes our presentation of the URTM.

4 Related Work

In the recent work [2] the authors studied reversible Turing machines from a programming language viewpoint, defined RTM-universality, and showed the results summarized in Sect. 2 about the expressiveness and robustness of the

[6] The reliance on the reversibility of the interpreted machine explains why our URTM is not a general UTM: It will get stuck if the interpreted program is not reversible.

1-tape 3-symbol RTMs. These results form the theoretical basis for the URTM developed in this paper.

Morita *et al.* have also studied RTMs [11,10] and other powerful reversible computation models, including cellular automata [9]. In [11] a small universal RTM was proposed, which implements an interpreter for a *cyclic tag system* (a Turing complete formalism.) However, under our strict semantics approach, the proposed machine does not demonstrate universality (or RTM-universality), as the halting configuration encompasses not just the program and output, but also the entire string produced by the tag system along the way, analogous to a Landauer embedding [8]. Of course, the intent with their machine also appears to be rather different from ours. We did not aim for a minimal machine in terms of states and symbols, and allowed ourselves 3 tapes compared to only 1 in [11].

5 Conclusion

The study of reversible computation models complements that of deterministic and non-deterministic models. Despite a long history, the fundamental properties of reversible computation models are still not well-understood. In our approach, where reversibility of a Turing machine implies injectivity of its semantical function, we have that reversible Turing machines (RTMs) are not quite Turing complete, but still expressive enough to be universal for their own class with the natural concept of *RTM-universality* (which allows a universal machine to remember the interpreted program text).

We here showed the first RTM-universal reversible Turing machine (URTM) constructed from first principles. The resulting machine is a very clean and simple design. We did not have to rely on reversibilization techniques, and the URTM never writes any temporary values to tape, except for the constant-sized string encoding the current simulated state. The URTM interprets 1-tape 3-symbol RTMs with a program dependent constant factor slowdown (bounded by the size of the interpreted program). Importantly, there are no other change in the time and space behavior as compared to the interpreted machine. Thus, for individual machines the URTM is effectively complexity preserving, which has not been seen before with reversible Turing machines. In addition, the URTM is also able to function as an *inverse interpreter* for running RTM programs *backwards* by simply reversing the program text, again with no impact on asymptotic complexity. These positive qualities were made possible by explicitly exploiting the promise of reversibility of the interpreted machine in the URTM design.

We conclude that the RTMs can simulate themselves efficiently.

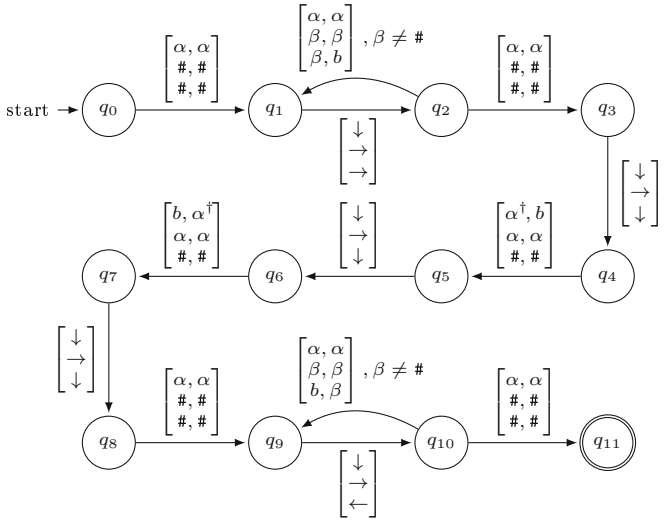
References

1. Axelsen, H.B.: Clean translation of an imperative reversible programming language. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 144–163. Springer, Heidelberg (2011)
2. 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)

3. Axelsen, H.B., Glück, R., Yokoyama, T.: Reversible machine code and its abstract processor architecture. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) CSR 2007. LNCS, vol. 4649, pp. 56–69. Springer, Heidelberg (2007)
4. Bennett, C.H.: Logical reversibility of computation. IBM Journal of Research and Development 17, 525–532 (1973)
5. Feynman, R.: Quantum mechanical computers. Optics News 11, 11–20 (1985)
6. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. ACM Trans. Prog. Lang. Syst. 29(3), Article 17 (2007)
7. Kutrib, M., Malcher, A.: Reversible pushdown automata. In: Dediu, A.-H., Fernau, H., Martín-Vide, C. (eds.) LATA 2010. LNCS, vol. 6031, pp. 368–379. Springer, Heidelberg (2010)
8. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development 5(3), 183–191 (1961)
9. Morita, K.: Reversible computing and cellular automata — A survey. Theoretical Computer Science 395(1), 101–131 (2008)
10. Morita, K., Shirasaki, A., Gono, Y.: A 1-tape 2-symbol reversible Turing machine. Trans. IEICE E 72(3), 223–228 (1989)
11. Morita, K., Yamaguchi, Y.: A universal reversible turing machine. In: Durand-Lose, J., Margenstern, M. (eds.) MCU 2007. LNCS, vol. 4664, pp. 90–98. Springer, Heidelberg (2007)
12. Mu, S.C., Hu, Z., Takeichi, M.: An algebraic approach to bi-directional updating. In: Chin, W.N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 2–20. Springer, Heidelberg (2004)
13. Thomsen, M.K., Axelsen, H.B.: Parallelization of reversible ripple-carry adders. Parallel Processing Letters 19(2), 205–222 (2009)
14. Thomsen, M.K., Glück, R., Axelsen, H.B.: Reversible arithmetic logic unit for quantum arithmetic. Journal of Physics A: Mathematical and Theoretical 42(38), 382002 (2010)
15. van de Snepscheut, J.L.A.: What computing is all about. Springer, Heidelberg (1993)
16. Van Rentergem, Y., De Vos, A.: Optimal design of a reversible full adder. International Journal of Unconventional Computing 1(4), 339–355 (2005)
17. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Proceedings of Computing Frontiers, pp. 43–54. ACM Press, New York (2008)
18. Yokoyama, T., Axelsen, H.B., Glück, R.: Reversible flowchart languages and the structured reversible program theorem. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 258–270. Springer, Heidelberg (2008)

Appendix A

Rule Application



Above is given an RTM state transition diagram for applying rule $\big(q, (s, s'), q'\big)$ (represented on the program tape) given that the current interpreted state q_c (on the state tape) is q, and the current interpreted symbol (on the work tape) is s. States q_0 through q_3 (corresponding to clear $q_c = q$) deletes the current interpreted state q from the state tape; states q_4 through q_7 (corresponding to write $q_c = q'$) substitutes the current interpreted symbol s on the first tape with s'; states q_8 though q_{11} writes the new interpreted state q' on the state tape. $\alpha^\dagger$ in the q_4 to q_5 and q_6 to q_7 transitions is b if $\alpha = \mathtt{B}$ and α otherwise.