What Do Reversible Programs Compute?

Holger Bock Axelsen and Robert Glück

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

Abstract. Reversible computing is the study of computation models that exhibit both forward and backward determinism. Understanding the fundamental properties of such models is not only relevant for reversible programming, but has also been found important in other fields, *e.g.*, bidirectional model transformation, program transformations such as inversion, and general static prediction of program properties.

Historically, work on reversible computing has focussed on *reversible simulations of irreversible computations*. Here, we take the viewpoint that the property of reversibility itself should be the starting point of a computational theory of reversible computing. We provide a novel semantics-based approach to such a theory, using *reversible Turing machines* (RTMs) as the underlying computation model.

We show that the RTMs can compute *exactly all injective, computable functions*. We find that the RTMs are *not* strictly classically universal, but that they support another notion of universality; we call this *RTM-universality*. Thus, even though the RTMs are sub-universal in the classical sense, they are powerful enough as to include a self-interpreter. Lifting this to other computation models, we propose *r-Turing completeness* as the 'gold standard' for computability in reversible computation models.

1 Introduction

The computation models that form the basis of programming languages are usually deterministic in *one direction* (forward), but non-deterministic in the opposite (backward) direction. Most other well-studied programming models exhibit non-determinism in both computation directions. Common to both of these classes is that they are information lossy, because generally a previous computation state cannot be recovered from a current state. This has implications on the analysis and application of these models. *Reversible computing* is the study of computation models wherein all computations are organized *two-way* deterministically, without any logical information loss.

Reversible computation models have been studied in widely different areas ranging from cellular automata [11], program transformation concerned with the inversion of programs [19], reversible programming languages [3,21], the view-update problem in bidirectional computing and model transformation [14,6], static prediction of program properties [15], digital circuit design [18,20], to quantum computing [5]. However, between all these cases, the definition and use of reversibility varies significantly (and subtly), making it difficult to apply

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 42–56, 2011.

results learned in one area to others. For example, even though reversible Turing machines were introduced several decades ago [4], the authors have found that there has been a blurring of the concepts of *reversibility* and *reversibilization*, which makes it difficult to ascertain exactly what is being computed, in later publications.

This paper aims to establish the foundational computability aspects for reversible computation models from a formal semantics viewpoint, using reversible Turing machines (RTMs, [4]) as the underlying computation model, to answer the question: *What do reversible programs compute?*

In reversible computation models, each atomic computation step *must* be reversible. This might appear as too restrictive to allow general and useful computations in reversible computation models. On the other hand, it might appear from the seminal papers by Landauer [9] and Bennett [4], that reversibility is not restrictive at all, and that all computations can be performed reversibly. We show that both of these viewpoints are wrong, under the view that the functional (semantical) behavior of a reversible machine should be logically reversible.

This paper brings together many different streams of work in an integrated semantics formalism that makes reversible programs accessible to a precise analysis, as a stepping stone for future work that makes use of reversibility. Thus, this paper is also an attempt to give a precise structure and basis for a foundational computability theory of reversible languages, in the spirit of the semantics approach to computability of Jones [8].

We give a formal presentation of the reversible Turing machines (Sect. 2), and, using a semantics-based approach (Sect. 3), outline the foundational results of reversible computing (Sect. 4). We show the computational robustness of the RTMs under reductions of the number of symbols and tapes (Sect. 5). Following a proof tactic introduced by Bennett, we show that the RTMs can compute exactly all injective, computable functions (Sect. 6). We study the question of universality, and give a novel interpretation of the concept (*RTM-universality*) that applies to RTMs, and prove constructively that the RTMs are RTM-universal (Sect. 7). We propose *r-Turing completeness* (Sect. 8) as the measure for computability of reversible computation models. Following a discussion of related work (Sect. 9) we present our conclusions (Sect. 10).

2 Reversible Triple-Format Turing Machines

The computation model we shall consider here is the *Turing machine* (TM). Recall that a Turing machine consists of a (doubly-infinite) *tape* of cells along which a *tape head* moves in discrete steps, reading and writing on the tape according to an *internal state* and a fixed *transition relation*. We shall here adopt a *triple format* for the rules which is similar to Bennett's quadruple format [4], but has the advantage of being slightly easier to work with[1].

[1] It is straightforward to translate back and forth between triple, quadruple, and the usual quintuple formats.

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,*

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

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).

The form of a triple in δ is either a *symbol rule* $(q, (s, s'), q')$ or a *shift rule* (q, d, q') where $q, q' \in Q$, $s, s' \in \Sigma$, and $d \in \{\leftarrow, \downarrow, \rightarrow\}$. Intuitively, a symbol rule says that in state q, if the tape head is reading symbol s, write s' and change into state q'. A shift rule says that in state q, move the tape head in direction d and change into state q'. It is easy to see how to extend the definition to k-tape machines by letting

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

Definition 2 (Configuration). *The* configuration *of a TM is a tuple $(q, (l, s, r)) \in Q \times (\Sigma^* \times \Sigma \times \Sigma^*) = \mathcal{C}$, where $q \in Q$ is the internal state, $l, r \in \Sigma^*$ are the parts of the tape to the left and right of the tape head represented as strings, and $s \in \Sigma$ is the symbol being scanned by the tape head*[2].

Definition 3 (Computation step). *A TM $T = (Q, \Sigma, \delta, b, q_s, q_f)$ in configuration $C \in \mathcal{C}$* leads to *configuration $C' \in \mathcal{C}$, written as $T \vdash C \rightsquigarrow C'$, defined for $s, s' \in \Sigma$, $l, r \in \Sigma^*$ and $q, q' \in Q$ by*

$$\begin{array}{lll}
T \vdash (q, (l, s, r)) & \rightsquigarrow (q', (l, s', r)) & \text{if} \quad (q, (s, s'), q') \in \delta \ , \\
T \vdash (q, (ls', s, r)) & \rightsquigarrow (q', (l, s', sr)) & \text{if} \quad (q, \leftarrow, q') \in \delta \ , \\
T \vdash (q, (l, s, r)) & \rightsquigarrow (q', (l, s, r)) & \text{if} \quad (q, \downarrow, q') \in \delta \ , \\
T \vdash (q, (l, s, s'r)) & \rightsquigarrow (q', (ls, s', r)) & \text{if} \quad (q, \rightarrow, q') \in \delta \ .
\end{array}$$

Definition 4 (Local forward/backward determinism). *A TM $T = (Q, \Sigma, \delta, b, q_s, q_f)$ is* locally forward deterministic *iff for any distinct pair of transition rule triples $(q_1, a_1, q_1'), (q_2, a_2, q_2') \in \delta$, 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* locally backward deterministic *iff for any distinct pair of triples $(q_1, a_1, q_1'), (q_2, a_2, q_2') \in \delta$, if $q_1' = q_2'$ then $a_1 = (s_1, s_1')$ and $a_2 = (s_2, s_2')$, and $s_1' \neq s_2'$.*

As an example, the pair $(\mathtt{q}, (\mathtt{a}, \mathtt{b}), \mathtt{p})$ and $(\mathtt{q}, (\mathtt{a}, \mathtt{c}), \mathtt{p})$ respects backward determinism (but not forward determinism); the pair $(\mathtt{q}, (\mathtt{a}, \mathtt{b}), \mathtt{p})$ and $(\mathtt{r}, (\mathtt{c}, \mathtt{b}), \mathtt{p})$ is not backward deterministic; and neither is the pair $(\mathtt{q}, (\mathtt{a}, \mathtt{b}), \mathtt{p})$ and $(\mathtt{r}, \rightarrow, \mathtt{p})$[3].

Definition 5 (Reversible Turing machine). *A TM T is* reversible *iff it is locally forward and backward deterministic.*

[2] When describing tape contents we shall use the empty string ε to denote the infinite string of blanks b^ω, and shall usually omit it when unambiguous.

[3] When we use `typewriter` font we usually refer to concrete instances, rather than variables. Thus, in this example `q` and `p` refers to different concrete states.

The reversible Turing machines (RTMs) are thus a proper subset of the set of all Turing machines, with an easily decidable property. We need the following important lemma. Note that this applies to *each* computation step.

Lemma 1. *If T is a reversible Turing machine, then the induced computation step relation $T \vdash \cdot \rightsquigarrow \cdot$ is an injective function on configurations.*

3 Semantics for Turing Machines

What do Turing machines compute? In other words, what is the codomain and definition of the semantics function $[\![\cdot]\!] : \text{TMs} \to ?$ for Turing machines? This might seem an odd question seeing as we have just defined how TMs work, but the answer depends on a concrete semantical choice, and has a profound effect on the computational strength of the RTMs. (Note: For the rest of this paper, we shall consider the relationship mainly between *deterministic* and *reversible* Turing machines. Thus, all TMs are assumed to be fwd deterministic).

At this point, the expert reader might object that the original results by Landauer [9] and Bennett [4] (cf. Lemmas 4 and 5) show exactly how we can "reversibilize" any computation, and that the RTMs should therefore be able to compute exactly what the TMs in general can compute. Furthermore, Morita and Yamaguchi [13] exhibited a universal reversible Turing machine, so the universality of the RTMs should already be established. As we shall see, however, if one takes reversibility as also including the input/output behaviour of the machines, neither of these claims hold: Reversibilization is *not* semantics preserving, and the RTMs are *not* universal in the classical sense.

There are several reasons for considering the extensional behavior of RTMs to itself be subject to reversibility.

- The *reversible* machines, computation models and programming languages, form a much *larger* class than just the *reversibilized* machines of Landauer and Bennett performing *reversible simulations of irreversible machines.*
- It leads to a richer and more elegant (functional) theory for reversible programs: Program composition becomes function composition, program inversion becomes function inversion, etc., and we are able to use such properties directly and intuitively in the construction of new reversible programs. This is *not* the case for reversible simulations.
- If we can *ad hoc* dismiss part of the output configuration, there seems to be little to constrain us from allowing such irreversibilities as part of the computation process as well.

In order to talk about input/output behavior on tapes in a regular fashion, we use the following definition.

Definition 6 (Standard configuration). *A tape containing a finite, blank-free string $s \in (\Sigma \backslash \{b\})^*$ is said to be given in* standard configuration *for a TM $(Q, \Sigma, \delta, b, q_s, q_f)$ iff the tape head is positioned to the* immediate left *of s on the tape, i.e. if for some $q \in Q$, the configuration of the TM is $(q, (\varepsilon, b, s))$.*

We shall consider the tape input/output (function) behavior. Here, the semantic function of a Turing machine is defined by its effect on the entire configuration.

Definition 7 (String transformation semantics). *The* semantics $[\![T]\!]$ *of a TM* $T = (Q, \Sigma, \delta, b, q_s, q_f)$ *is given by the relation*

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

Intuitively, a computation is performed as follows. In starting state q_s, with input s given in standard configuration $(q_s, (\varepsilon, b, s))$, repeatedly apply $\rightsquigarrow$, until the machine halts (if it halts) in standard configuration $(q_f, (\varepsilon, b, s'))$. To differentiate between semantics and mechanics, we shall write $T(x)$ to mean the computation of $[\![T]\!](x)$ by the specific machine T. We say that T *computes* function f iff $[\![T]\!] = f$. Thus, the *string transformation semantics* of a TM T has type

$$[\![T]\!] : \Sigma^* \rightharpoonup \Sigma^* \ .$$

Under this semantics there is a one-to-one correspondence between input/output strings and the configurations that represent them, so the machine input/output behaviour is logically reversible. In contrast to this, the (implicit) semantics used for decision problems (*language recognition*) gives us programs of type

$$[\![T]\!]_{dp} : \Sigma^* \rightharpoonup \{accept, reject\} \ ,$$

where halting configurations are projected down to a single bit. It is well known that for classical Turing machines it does not matter computability-wise which of these two semantics we choose. (There is a fairly straightforward translation from languages to functions and *vice versa*.) Anticipating the Landauer embedding of Lemma 4 it is easy to see that under the language recognition semantics then the RTMs are universal: Given a TM T recognizing language L, there exists an RTM T' that also recognizes L. However, under the string transformation semantics the RTMs *cannot* be universal.

Theorem 1. *If* T *is an RTM, then* $[\![T]\!]$ *is injective.*

Proof. By induction, using Lemma 1. □

It thus makes little sense to talk about what RTMs compute without explicitly specifying the semantics.

4 Foundations of Reversible Computing

At this point it becomes necessary to recast the foundational results of reversible computing in terms of the strict semantical interpretation above.

4.1 Inversion

If f is a computable injective function, is the inverse function f^{-1} computable?

Lemma 2 (TM inversion, McCarthy [10]). *Given a TM T computing an injective function $[\![T]\!]$, there exists a TM $M(T)$, such that $[\![M(T)]\!] = [\![T]\!]^{-1}$.*

It is interesting to note that McCarthy's generate-and-test approach [10] does not actually give the *program inverter* (computing the transformation M), but rather an *inverse interpreter*, cf. [1]. However, we can turn an inverse interpreter into a program inverter by specialization [7], so the transformation M is computable.

The generate-and-test method used by McCarthy is sufficient to show the existence of an inverse interpreter, but unsuitable for practical usage as it is very inefficient. For the RTMs there is an appealing alternative.

Lemma 3 (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 $inv : \Delta \to \Delta$ is defined as*

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

This remarkably simply transformation is one of the great insights in Bennett's seminal 1973 paper [4] that may superficially seem trivial. Here, we have additionally shown that it is an example of local (peephole) program inversion. Note that the transformation *only* works as a program inversion under the string transformation semantics, and *not* under language recognition. In the following, we shall make heavy use of program inversion, so the direct coupling between the mechanical and semantical transformation is significant.

4.2 Reversibilization

How can irreversible TMs computing (possibly non-injective) functions be *reversibilized*, i.e., transformed into RTMs?

Lemma 4 (Landauer embedding [9]). *Given a 1-tape TM $T = (Q, \Sigma, \delta, b, q_s, q_f)$, there is a 2-tape RTM $L(T)$ such that $[\![L(T)]\!] : \Sigma^* \rightharpoonup \Sigma^* \times R^*$, and*

$$[\![L(T)]\!] = \lambda x.([\![T]\!](x), trace(T, x)),$$

where $trace(T, x)$ is a complete trace of the specific rules from δ (enumerated as R) that are applied during the computation $T(x)$.

The Landauer embedding is named in honor of Rolf Landauer, who suggested the idea of a trace to ensure reversibility [9]. It is historically the first example of what we call "reversibilization," the addition of *garbage data* to the output in order to guarantee reversibility. The Landauer embedding shows that any computable function can be injectivized such that it is computable by a reversible TM.

The size of the garbage data $trace(T, x)$ is of order of the number of steps in the computation $T(x)$, which makes it in general unsuited for practical programming. The trace is also machine-specific: Given functionally equivalent TMs T_1 and T_2,

i.e., $[\![T_1]\!] = [\![T_2]\!]$, it will almost always be the case that $[\![L(T_1)]\!] \neq [\![L(T_2)]\!]$. The addition of the trace also changes the space consumption of the original program.

It is preferable that an injectivization generates *extensional* garbage data (specific to the function) rather than *intensional* garbage data (specific to the machine), since we would like to talk about semantics and ignore the mechanics. This is attained in the following Lemma, known colloquially as "Bennett's trick."

Lemma 5 (Bennett's method [4]). *Given a 1-tape TM* $T = (Q, \Sigma, \delta, b, q_s, q_f)$*, there exists a 3-tape RTM* $B(T)$*, s.t.*

$$[\![B(T)]\!] = \lambda x.(x, [\![T]\!](x)) \ .$$

While the construction (shown below) is defined for 1-tape machines, it can be extended to Turing machines with an arbitrary number of tapes. It is important to note that neither Landauer embedding nor Bennett's method are semantics preserving as both reversibilizations lead to garbage:

$$[\![L(T)]\!] \neq [\![T]\!] \neq [\![B(T)]\!] \, .$$

4.3 Reversible Updates

Bennett's method implements a special case of a *reversible update* [3], where D (below) is a simple "copying machine", and the second input is initially blank:

Theorem 2. *Assume that* $\odot : (\Sigma^* \times \Sigma^*) \to \Sigma^*$ *is a (computable) operator injective in its first argument: If* $b \odot a = c \odot a$*, then* $b = c$*. Let* D *be an RTM computing the injective function* $\lambda(a, b).(a, b \odot a)$*, and let* T *be any TM. Let* $L_1(T)$ *be an RTM that applies* $L(T)$ *to the first argument* x *of a pair* (x, y) *(using an auxiliary tape for the trace.) We have*[4]

$$[\![L_1(T)^{-1} \circ D \circ L_1(T)]\!] = \lambda(x, y).(x, y \odot [\![T]\!](x)) \ .$$

Reversible updates models many reversible language constructs [21], and is also useful when designing reversible circuits [16,17]. We found this generalization to be of great practical use in the translation from high-level to low-level reversible languages [2], as it directly suggests a translation strategy for reversible updates.

5 Robustness

The Turing machines are remarkably computationally robust. Using multiple symbols, tapes, heads etc. has no impact on computability. Above, we have been silently assuming that the same holds true for the RTMs: The Landauer embedding takes n-tape machine to $n+1$-tape machines, the Bennett trick takes 1-tape machines to 3-tape machines, etc.

[4] The mechanical concatenation of two machines $T_2 \circ T_1$ is straightforward, and it is an easy exercise to see that $[\![T_2 \circ T_1]\!] = [\![T_2]\!] \circ [\![T_1]\!]$.

Are these assumptions justified? We have seen that a precise characterization of the semantics turned out to have a huge impact on computational expressiveness (limiting us to injective functions.) It would not be unreasonable to expect the RTMs to suffer additional restrictions *wrt* the parameters of machine space.

First, we consider the question of multiple symbols. Morita *et al.* [12] showed how to simulate a 1-tape, 32-symbol RTM by a 1-tape 2-symbol RTM. One can generalize this result to an arbitrary number of symbols. Furthermore, we also need to adapt it to work when applying our string transformation semantics such that the encodings can be efficient[5].

Lemma 6 (m-symbol RTM to 3-symbol RTM). *Given a 1-tape, m-symbol RTM $T = (Q, \Sigma, \delta, b, q_s, q_f)$, $|\Sigma| = m$, there is a 1-tape, 3-symbol RTM $T' = (Q', \{b, \mathtt{0}, \mathtt{1}\}, \delta', b, q_s, q_f)$ s.t. $[\![T]\!](x) = y$ iff $[\![T']\!](e(x)) = e(y)$, where $e : (\Sigma \backslash \{b\})^* \to \{\mathtt{0}, \mathtt{1}\}^*$ is an injective binary encoding of (blank-free) strings, with b encoded by a sequence of blanks.*

Thus, the number of symbols in a tape alphabet is not important, and a fixed-size alphabet (with at least 3 distinct symbols) can be used for all computations.

We now turn to the question of multiple tapes.

Lemma 7 (2-tape RTM to 1-tape RTM). *Given a 2-tape RTM T, there exists a 1-tape RTM T' s.t. $[\![T]\!](x, y) = (u, v)$ iff $[\![T']\!](\langle x, y\rangle) = \langle u, v\rangle$, where $\langle x, y\rangle = x_1 y_1, x_2 y_2, \ldots$ is the pairwise character sequence (convolution) of strings x and y (with blanks at the end of the shorter string as necessary.)*

The main difficulty in proving this is that the original 2-tape machine may allow halting configurations where the tape heads end up displaced an unequal number of cells from their starting positions. Thus "zipping" the tapes into one tape will not necessarily give the convolution of the outputs in standard configuration. This is corrected by realigning the simulated tapes for each rule where the original tape heads move differently.

This result generalizes to an arbitrary number of tapes. Combining these two lemmas yields the following statement of robustness.

Theorem 3 (Robustness of the RTMs). *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 retroactively justifies the use of the traditional transformational approaches.

6 Exact Computational Expressiveness of the RTMs

We have outlined the two classical reversibilizations that turn TMs into RTMs. However, they are not semantics-preserving, and do not tell us anything about

[5] A 2-symbol machine can only have unary input strings in standard configuration, as one of the two symbols must be the blank symbol.

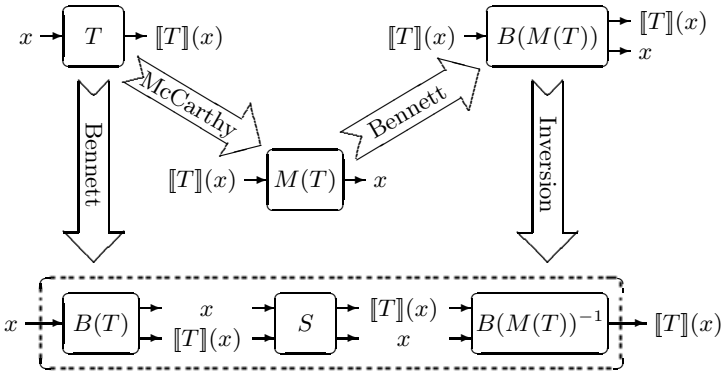

Fig. 1. Generating an RTM computing (injective) $[\![T]\!]$ from an irreversible TM T

the *a priori* computational expressiveness of RTMs. By Theorem 1 the RTMs compute only injective functions. How many such functions *can* they compute?

Theorem 4 (Reversibilizing injections, Bennett [4]). *Given a 1-tape TM S_1 s.t. $[\![S_1]\!]$ is injective, and given a 1-tape TM S_2 s.t. $[\![S_2]\!] = [\![S_1]\!]^{-1}$, there exists a 3-tape RTM T s.t. $[\![T]\!] = [\![S_1]\!]$.*

We can use this to establish the exact computational expressiveness of the RTMs.

Theorem 5 (Expressiveness). *The RTMs can compute* exactly all *injective computable functions. That is, given a 1-tape TM T such that $[\![T]\!]$ is an injective function, there is a 3-tape RTM T' such that $[\![T]\!] = [\![T']\!]$.*

This theorem then follows from the existence of a TM inverter (Lemma 2) and Theorem 4. We make use of the construction used by Bennett to prove Theorem 4, but now operating purely at the semantics level, making direct use of the transformations, and without assuming that an inverse for T is given *a priori*.

Proof. We construct and concatenate three RTMs (see Fig. 1 for a graphical representation.) First, construct $B(T)$ by applying Lemma 5 directly to T:

$$[\![B(T))]\!] = \lambda x.(x, [\![T]\!](x)) \ , \quad B(T) \in \text{RTMs}$$

Second, construct the machine $B(M(T))^{-1}$ by successively applying the transformations of Lemmas 2, 5 and 3 to T:

$$[\![B(M(T))^{-1}]\!] = (\lambda y.(y, [\![T]\!]^{-1}(y)))^{-1} \ , \quad B(M(T))^{-1} \in \text{RTMs}$$

Third, we can construct an RTM S, s.t. $[\![S]\!] = \lambda(a, b).(b, a)$, that is, a machine to exchange the contents of two tapes (in standard configuration). To see that $[\![B(M(T))^{-1} \circ S \circ B(T)]\!] = [\![T]\!]$, we apply the machine to an input, x:

$$\begin{aligned}
[\![B(M(T))^{-1} \circ S \circ B(T)]\!](x) &= [\![B(M(T))^{-1} \circ [\![S]\!]\ (x, [\![T]\!](x)) \\
&= [\![B(M(T))^{-1}]\!]\ ([\![T]\!](x), x)) \\
&= (\lambda y.(y, [\![T]\!]^{-1}(y)))^{-1}\ ([\![T]\!](x), x) \\
&= (\lambda y.(y, [\![T]\!]^{-1}(y)))^{-1}\ ([\![T]\!](x), [\![T]\!]^{-1}([\![T]\!](x))) \\
&= [\![T]\!](x)\ . \qquad \square
\end{aligned}$$

Thus, the RTMs can compute exactly all the injective computable functions. This suggests that the RTMs have the maximal computational expressiveness we could hope for in *any* (effective) reversible computing model.

7 Universality

Having characterized the computational *expressiveness* of the RTMs, an obvious next question is that of computation *universality*. A universal machine is a machine that can simulate the functional behaviour of any other machine. For the classical, irreversible Turing machines, we have the following definition.

Definition 8 (Classical universality). *A TM U is* classically universal *iff for all TMs T, all inputs $x \in \Sigma^*$, and Gödel number $\ulcorner T \urcorner \in \Sigma^*$ representing T:*

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

The actual Gödel numbering $\ulcorner \cdot \urcorner$: TMs $\to \Sigma^*$ for a given universal machine is not important, but we do require that it is computable and injective (up to renaming of symbols and states).

Because $[\![U]\!]$ in this definition is a *non-injective* function, it is clear that *no classically universal RTM exists!* Bennett [4] suggests that if U is a (classically) universal machine, $B(U)$ is a machine for reversibly simulating any irreversible machine. However, $B(U)$ is not itself universal, $[\![B(U)]\!] \neq [\![U]\!]$, and furthermore we should not use *reversible simulation of irreversible machines* as a benchmark.

The appropriate question to ask is whether the RTMs are classically universal for just their own class, i.e. where the interpreted machine T is restricted to being an RTM. The answer is, again, no: Different programs may compute the same function, so there exists RTMs $T_1 \neq T_2$ such that $[\![T_1]\!](x) = [\![T_2]\!](x)$, so $[\![U]\!]$ is *inherently* non-injective, and therefore not computable by any RTM.

Classical universality is thus unsuitable if we want to capture a similar notion wrt RTMs. We propose that a universal machine should be allowed to remember which machine it simulates.

Definition 9 (Universality). *A TM U_{TM} is* universal *iff for all TMs T and all inputs $x \in \Sigma^*$,*

$$[\![U_{TM}]\!](\ulcorner T \urcorner, x) = (\ulcorner T \urcorner, [\![T]\!](x))\ .$$

This is equivalent to the original definition of classical universality[6]. Importantly, it now suggests a concept of universality that *can* apply to RTMs.

Definition 10 (RTM-universality). *An RTM U_{RTM} is* RTM-universal *iff for all RTMs T and all inputs $x \in \Sigma^*$,*

$$[\![U_{RTM}]\!](\ulcorner T \urcorner, x) = (\ulcorner T \urcorner, [\![T]\!](x)) \ .$$

Now, is there an RTM-universal reversible Turing machine, a *URTM*?

Theorem 6 (URTM existence). *There exists an RTM-universal RTM U_R.*

Proof. We show that an RTM U_R exits, such that for all RTMs T, $[\![U_R]\!](\ulcorner T \urcorner, x) = (\ulcorner T \urcorner, [\![T]\!](x))$. Clearly, $[\![U_R]\!]$ is computable, since T is a TM (so $[\![T]\!]$ is computable), and $\ulcorner T \urcorner$ is given as input. We show that $[\![U_R]\!]$ is injective: Assuming $(\ulcorner T_1 \urcorner, x_1) \neq (\ulcorner T_2 \urcorner, x_2)$ we show that $(\ulcorner T_1 \urcorner, [\![T_1]\!](x_1)) \neq (\ulcorner T_2 \urcorner, [\![T_2]\!](x_2))$. Either $\ulcorner T_1 \urcorner \neq \ulcorner T_2 \urcorner$ or $x_1 \neq x_2$ or both. Because the program text is passed through to the output, the first and third cases are trivial. Assuming that $x_1 \neq x_2$ and $\ulcorner T_1 \urcorner = \ulcorner T_2 \urcorner$, we have that $[\![T_1]\!] = [\![T_2]\!]$, i.e. T_1 and T_2 are the same machine, and so compute the same function. Because they are RTMs this function is injective (by Theorem 1), so $x_1 \neq x_2$ implies that $[\![T_1]\!](x_1) \neq [\![T_2]\!](x_2)$. Therefore, $[\![U_R]\!]$ is injective, and by Theorem 5 computable by some RTM U_R. □

We remark that this works out very nicely: RTM-universality is now simply universality restricted to interpreting the RTMs, and while general universality is non-injective, RTM-universality becomes exactly injective by virtue of the semantics of RTMs. Also, by interpreting just the RTMs, we remove the redundancy (and reliance on reversibilization) inherent in the alternatives.

Given an irreversible TM computing the function of RTM-universality, Theorem 5 provides us with a possible construction for an RTM-universal RTM. However, we do not actually directly have such machines in the literature, and in any case the construction uses the very inefficient generate-and-test inverter by McCarthy. We can do better.

Lemma 8. *There exists an RTM pinv, such that pinv is a program inverter for RTM programs,*

$$[\![pinv]\!](\ulcorner T \urcorner) = \ulcorner T^{-1} \urcorner \ .$$

This states that the RTMs are expressive enough to perform the program inversion of Lemma 3. For practical Gödelizations this will only take linear time.

Theorem 7 (UTM to URTM). *Given a classically universal TM U s.t. $[\![U]\!](\ulcorner T \urcorner, x) = [\![T]\!](x)$, the RTM U_R defined as follows is RTM-universal.*

$$U_R = pinv_1 \circ (B(U))^{-1} \circ S_{23} \circ pinv_1 \circ B(U) \, ,$$

where $pinv_1$ is an RTM that applies RTM program inversion on its first argument, $[\![pinv_1]\!](p, x, y) = ([\![pinv]\!]p, x, y)$, and S_{23} is an RTM that swaps its second and third arguments, $[\![S_{23}]\!] = \lambda(x, y, z).(x, z, y)$.

[6] Given U_{TM} universal by Definition 9, *snd* $\circ\, U_{TM}$ is classically universal, where *snd* is a TM s.t. $[\![snd]\!] = \lambda(x, y).y$. The converse is analogous.

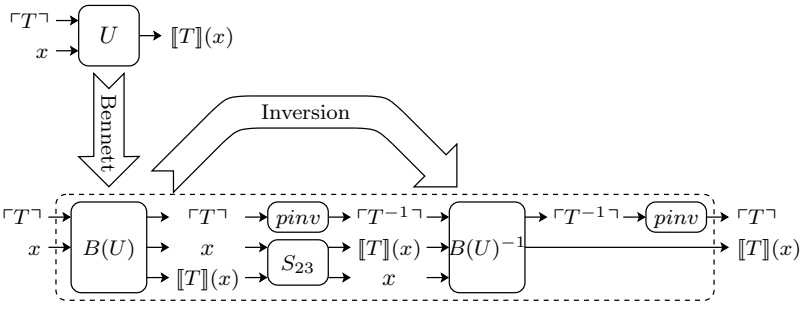

Fig. 2. Constructing an RTM-universal RTM U_R from a classically universal TM U

Proof. We must show that $[\![U_R]\!](\ulcorner T \urcorner, x) = (\ulcorner T \urcorner, [\![T]\!](x))$ for any RTM T. To show this, we apply U_R to an input $(\ulcorner T \urcorner, x)$. Fig. 2 shows a graphical representation of the proof.

$$\begin{aligned}
[\![U_R]\!](\ulcorner T \urcorner, x) &= [\![pinv_1 \circ (B(U))^{-1} \circ S_{23} \circ pinv_1 \circ B(U)]\!](\ulcorner T \urcorner, x) \\
&= [\![pinv_1 \circ (B(U))^{-1} \circ S_{23} \circ pinv_1]\!](\ulcorner T \urcorner, x, [\![T]\!](x)) \\
&= [\![pinv_1 \circ (B(U))^{-1} \circ S_{23}]\!](\ulcorner T^{-1} \urcorner, x, [\![T]\!](x)) \\
&= [\![pinv_1 \circ (B(U))^{-1}]\!](\ulcorner T^{-1} \urcorner, [\![T]\!](x), x) \\
&= [\![pinv_1]\!](\ulcorner T^{-1} \urcorner, [\![T]\!](x)) \\
&= (\ulcorner T \urcorner, [\![T]\!](x)) \ . \qquad \square
\end{aligned}$$

Note that this implies that RTMs can simulate themselves exactly as time-efficiently as the TMs can simulate themselves, but the space usage of the constructed machine will, by using Bennett's method, be excessive. However, there is nothing that forces us to start with an irreversible (universal) machine, when constructing an RTM-universal RTM, nor are reversibilizations necessarily required (as will be seen below).

A first principles approach to an RTM-universal reversible Turing machine, which does not rely on reversibilization, remains for future work.

8 r-Turing Completeness

With a theory of the computational expressiveness and universality of the RTMs at hand, we shall lift the discussion to computation models in general. What, then, do reversible programs compute, and what can they compute?

Our fundamental assumption is that the RTMs (with the given semantics) are a good and exhaustive model for reversible computing. Thus, for every program $\mathtt{p}$ in a reversible programming language R, we assume there to be an RTM $T_\mathtt{p}$, s.t. $[\![\mathtt{p}]\!]_R = [\![T_\mathtt{p}]\!]$. Thus, because the RTMs are restricted to computing injective functions, reversible programs too compute injective functions only. On the other hand, we have seen that the RTMs are maximally expressive wrt these functions,

and support a natural notion of universality. For this reason we propose the following standard of computability for reversible computing.

Definition 11 (r-Turing completeness). *A (reversible) programming language R is called* r-Turing complete *iff for all RTMs T computing function $[\![T]\!]$, there exists a program $\mathtt{p} \in R$, such that $[\![\mathtt{p}]\!]_R = [\![T]\!]$.*

Note that we are here quite literal about the semantics: Given an RTM T, it will *not* suffice to compute a Landauer embedded version of $[\![T]\!]$, or apply Bennett's trick, or, indeed any injectivization of $[\![T]\!]$. Program $\mathtt{p}$ *must* compute $[\![T]\!]$, *exactly*. Only if this is respected can we truly say that a reversible computation model can compute *all* the injective, computable functions, i.e. is as computationally expressive as we can expect reversible computation models to be.

Demonstrating r-Turing Completeness. A common approach to proving that a language, or computational model, is Turing-complete, is to demonstrate that a classically universal TM (a TM interpreter) can be implemented, and specialized to any TM T. However, that is for *classically* universal machines and (in general) irreversible languages, which compute non-injective functions. What about *our* notion of RTM-universality (Definition 10) and reversible languages?

Assume that $u \in R$ (where R is a reversible programming language) is an R-program computing an RTM-universal interpreter $[\![u]\!]_R(\ulcorner T \urcorner, x) = (\ulcorner T \urcorner, [\![T]\!](x))$. Assume also that R is expressive enough to guarantee the existence of programs $w_T \in R$ s.t. $[\![w_T]\!]_R = \lambda x.(\ulcorner T \urcorner, x)$, (whose sole purpose is to *hardcode* $\ulcorner T \urcorner$ as an input for u) and its inverse $w_T^{-1} \in R$, $[\![w_T^{-1}]\!]_R = [\![w_T]\!]_R^{-1}$, for any RTM T. Note that $[\![w_T]\!]_R$ is injective, so we do not violate our rule of R computing injective functions by assuming w_T and its inverse. Now $[\![u \circ w_T]\!]_R = \lambda x.(\ulcorner T \urcorner, [\![T]\!](x)) \neq [\![T]\!]$, because it leaves the representation $\ulcorner T \urcorner$ as part of the output. To complete the specialization, we need to apply w_T^{-1} as well. Thus, $[\![w_T^{-1} \circ u \circ w_T]\!]_R = [\![T]\!]$.

Therefore, completely analogous to the classical case, we may demonstrate r-Turing completeness of a reversible computation model by implementing RTM-universality (essentially, an RTM interpreter), keeping in mind that we must respect the semantics exactly (by *clean simulation* that doesn't leave *garbage*).

The authors have done exactly that to demonstrate r-Turing completeness of the imperative, high level, reversible language *Janus*, and for reversible flowchart languages in general, cf. [21,22] (where the r-Turing completeness concept was informally proposed.) In these cases, we were able to exploit the reversibility of the interpreted machines directly, and did not have to rely on reversibilization of any kind, which eased the implementation greatly. Furthermore, the RTM-interpreters are complexity-wise *robust*, in that they preserve the space and time complexities of the simulated machines, which no reversibilization is liable to do.

9 Related Work

Morita *et al.* have studied reversible Turing machines [13,12] and other reversible computation models, including cellular automata [11], with a different approach

to semantics (and thus different results wrt computational strength) than the present paper. Most relevant here is the universal RTM proposed in [13]. With our strict semantics viewpoint, the construction therein does *not* directly demonstrate neither RTM-universality nor classical universality, but rather a sort of "traced" universality: Given a program for a *cyclic tag system* (a Turing complete formalism) and an input, the halting configuration encompasses both the program and output, but also the entire string produced by the tag system along the way. We believe that this machine could possibly be transformed fairly straightforwardly into a machine computing a function analogous to $[\![B(U)]\!]$. However, it is not clear that cyclic tag systems should have a notion of reversibility, so the construction in Fig. 2 is therefore not immediately applicable.

10 Conclusion

The study of reversible computation models complements that of deterministic and non-deterministic computation models. We have given a foundational treatment of a computability theory for reversible computing using a strict semantics-based approach (where input/output behaviour must also be reversible), taking reversible Turing machines as the underlying computational model. By formulating the classical transformational approaches to reversible computation in terms of this semantics, we hope to have clarified the distinction between *reversibility* and *reversibilization*, which may previously have been unclear.

We found that starting directly with reversibility leads to a clearer, cleaner, and more useful functional theory for RTMs. Natural (mechanical) *program* transformations such as composition and inversion now correspond directly to the (semantical) *function* transformations. This carries over to other computation models as well.

We showed that the RTMs compute exactly all injective, computable functions, and are thus not classically universal. We also showed that they are expressive enough to be universal for their own class, with the concept of *RTM-universality*. We introduced the concept of *r-Turing completeness* as the measure of the computational expressiveness in reversible computing. As a consequence, a definitive practical criterion for deciding the computation universality of a reversible programming computation model is now in place: Implement an RTM-interpreter, in the sense of an RTM-universal machine.

Acknowledgements. The authors wish to thank Michael Kirkedal Thomsen for help with the figures and Tetsuo Yokoyama for discussions on RTM-computability.

References

1. Abramov, S., Glück, R.: Principles of inverse computation and the universal resolving algorithm. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 269–295. Springer, Heidelberg (2002)

2. 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)
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. Glück, R., Sørensen, M.: A roadmap to metacomputation by supercompilation. In: Danvy, O., Thiemann, P., Glück, R. (eds.) Partial Evaluation. LNCS, vol. 1110, pp. 137–160. Springer, Heidelberg (1996)
8. Jones, N.D.: Computability and Complexity: From a Programming Language Perspective. In: Foundations of Computing. MIT Press, Cambridge (1997)
9. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development 5(3), 183–191 (1961)
10. McCarthy, J.: The inversion of functions defined by Turing machines. In: Automata Studies, pp. 177–181. Princeton University Press, Princeton (1956)
11. Morita, K.: Reversible computing and cellular automata — A survey. Theoretical Computer Science 395(1), 101–131 (2008)
12. Morita, K., Shirasaki, A., Gono, Y.: A 1-tape 2-symbol reversible Turing machine. Trans. IEICE, E 72(3), 223–228 (1989)
13. 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)
14. 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)
15. Schellekens, M.: MOQA; unlocking the potential of compositional static average-case analysis. Journal of Logic and Algebraic Programming 79(1), 61–83 (2010)
16. Thomsen, M.K., Axelsen, H.B.: Parallelization of reversible ripple-carry adders. Parallel Processing Letters 19(2), 205–222 (2009)
17. Thomsen, M.K., Glück, R., Axelsen, H.B.: Reversible arithmetic logic unit for quantum arithmetic. Journal of Physics A: Mathematics and Theoretical 42(38), 2002 (2010)
18. Toffoli, T.: Reversible computing. In: de Bakker, J.W., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 632–644. Springer, Heidelberg (1980)
19. van de Snepscheut, J.L.A.: What computing is all about. Springer, Heidelberg (1993)
20. Van Rentergem, Y., De Vos, A.: Optimal design of a reversible full adder. International Journal of Unconventional Computing 1(4), 339–355 (2005)
21. 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)
22. 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)