4.1 Turing Machines

Turing machines, first described by Alan Turing in 1936, are abstract computational models used to formally capture the informal notion of a *computing system*. Having a formal definition of computability allows us to investigate and rigorously understand what can be and what cannot be computed using computing devices of mechanical or electrical nature. Intuitively, by a computing device we understand a piece of machinery that carries out tasks by successively applying sequences of instructions from a finite set of instructions, possibly using unlimited memory; such sequences of instructions are today called *programs*, or *procedures*, or *algorithms*.

The *Church-Turing thesis* postulates that any computing system, any algorithm, or any program in any programming language running on any computer, can be equivalently simulated by a Turing machine. Like any thesis, this is unprovable; however, as far as we believe it, we can formally reason about computability, in particular we can formally prove that certain problems can or cannot be solved using computers. The advantage of using Turing machines is that they are very simple devices, reducing the concept of computability to its basics. There are many other abstract machines and calculi in the literature that are equivalent to Turing machines; they all serve as equally good formal models of computation.

A Turing machine is a finite state device with infinite memory. The memory is very primitively organized, as one or more infinite "tapes" of cells that are sequentially accessible through a "head" that can move to the left or to the right cell only. Each cell can hold a bounded piece of data, typically a Boolean, or bit, value. The tape is also used as the input/output of the machine. The computational steps carried out by a Turing machine are also very primitive: in a state, depending on the value in the current cell, a Turing machine can only rewrite the current cell on the tape and/or move the head to the left or to the right. Therefore, a Turing machine does not have the direct capability to perform random memory access, but it can be shown that it can simulate it.

In this section we first give a formal definition of a (variant of) Turing machine, then we give two rewriting logic representations of it and let as exercise to the reader the completion of a third one. The first representation is more compact, but it requires some support from the underlying rewrite system in order to be executed (lazy evaluation strategy). The second representation is a bit more involved, but it can be executed on any rewrite engine, without any worry about evaluation strategies. Nevertheless, both representations are easily executable in systems like Maude or Haskell. We conclude this section with some historical remarks on Turing machines and Turing computability.

4.1.1 Formal Definition

There are many equivalent definitions of Turing machines in the literature. We prefer one with a tape that is infinite at both ends and describe it informally in the sequel. Consider a mechanical device which has associated with it a tape of infinite length in both directions, partitioned in spaces of equal size, called *cells*, which are able to hold either a 0 or an 1 and are rewritable. The device examines exactly one cell at any time, and can, potentially nondeterministically, perform any of the following four operations (or *commands*):

1. Write a 1 in the current cell;
2. Write a 0 in the current cell;
3. Shift one cell to the right;

4. Shift one cell to the left.

The device performs one operation per unit time, called a *step*. Figure 4.1 shows an artistic representation of a Turing machine. We next give a formal definition.

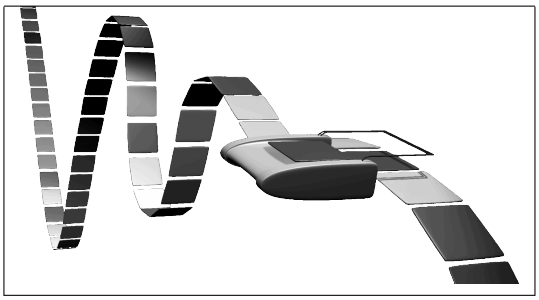


Figure 4.1: The Turing machine
— art by Alessio Damato —

Definition 20. *A **(deterministic) Turing machine** $\mathcal{M}$ is a 6-tuple (Q, B, q_s, q_h, C, M), where:*

- *Q is a finite set of **internal states**;*
- *$q_s \in Q$ is the **starting state** of $\mathcal{M}$;*
- *$q_h \in Q$ is the **halting state** of $\mathcal{M}$;*
- *B is the set of **symbols** of $\mathcal{M}$; we assume without loss of generality that $B = \{0, 1\}$;*
- *$C = B \cup \{\rightarrow, \leftarrow\}$ is the set of **commands** of $\mathcal{M}$;*
- *$M : (Q - \{q_h\}) \times B \rightarrow Q \times C$ is a total function, the **transition function** of $\mathcal{M}$.*

We assume that the tape contains only 0's before the machine starts performing.

Our definition above is for *deterministic* Turing machines. One can also define nondeterministic Turing machines by changing the transition function M into a relation. It can be shown that the non-deterministic Turing machines have the same computational power as the deterministic Turing machines, so, for our purpose in this section, we limit ourselves to deterministic machines.

A *configuration* of a Turing machine is a 4-tuple consisting of an internal state, a current cell, and two infinite strings (notice that the two infinite strings contain only 0's starting with a certain cell), standing for the cells on the left and for the cells on the right of the current cell, respectively. We let $(q, L\underline{b}R)$ denote the configuration in which the machine is in state q, with current cell b, left tape L and right tape R. For convenience, we write the left tape L backwards, that is, its head is at its right end; for example, Lb appends a b to the left tape L. Given a configuration $(q, L\underline{b}R)$, the content of the tape is LbR, which is infinite at both ends. We also let $(q, L\underline{b}R) \rightarrow_{\mathcal{M}} (q', L'\underline{b'}R')$ denote a configuration transition under one of the four commands. Given a configuration in which the internal state is q and the examined cell contains b, and if $M(q, b) = (q', c)$, then exactly one of the following configuration transitions can take place:

$(q, L\underline{b}R) \rightarrow_{\mathcal{M}} (q', L\underline{c}R)$ if $c = 0$ or $c = 1$;

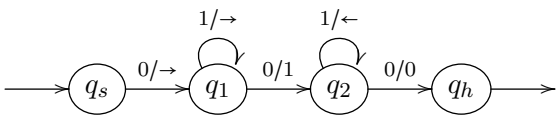
$(q, L\underline{b}b'R) \rightarrow_{\mathcal{M}} (q', Lb\underline{b'}R)$ if $c = \rightarrow$;

$(q, Lb'\underline{b}R) \rightarrow_{\mathcal{M}} (q', L\underline{b'}bR)$ if $c = \leftarrow$.

Since our Turing machines are deterministic (M is a function), the relation $\rightarrow_{\mathcal{M}}$ on configurations is also deterministic. The machine starts performing in the internal state q_s. If there is no input, the initial configuration on which the Turing machine is run is $(q_s, 0^\omega\underline{0}0^\omega)$, where 0^ω is the infinite string of zeros. If the Turing machine is intended to be run on a specific input, say $x = b_1b_2\cdots b_n$, its initial

States and transition function (graphical representation to the right):

$Q = \{q_s, q_h, q_1, q_2\}$
$M(q_s, 0) = (q_1, \rightarrow)$
$M(q_s, 1) =$ anything
$M(q_1, 0) = (q_2, 1)$
$M(q_1, 1) = (q_1, \rightarrow)$
$M(q_2, 0) = (q_h, 0)$
$M(q_2, 1) = (q_2, \leftarrow)$



Sample computation:

$(q_s, 0^\omega \underline{0}1110^\omega) \rightarrow_{\mathcal{M}} (q_1, 0^\omega \underline{1}110^\omega) \rightarrow_{\mathcal{M}} (q_1, 0^\omega 1\underline{1}10^\omega) \rightarrow_{\mathcal{M}} (q_1, 0^\omega 11\underline{1}0^\omega) \rightarrow_{\mathcal{M}}$
$(q_1, 0^\omega 111\underline{0}0^\omega) \rightarrow_{\mathcal{M}} (q_2, 0^\omega 111\underline{1}0^\omega) \rightarrow_{\mathcal{M}} (q_2, 0^\omega 11\underline{1}10^\omega) \rightarrow_{\mathcal{M}} (q_2, 0^\omega 1\underline{1}110^\omega) \rightarrow_{\mathcal{M}}$
$(q_2, 0^\omega \underline{1}1110^\omega) \rightarrow_{\mathcal{M}} (q_2, 0^\omega \underline{0}11110^\omega) \rightarrow_{\mathcal{M}} (q_h, 0^\omega \underline{0}11110^\omega)$

Figure 4.2: Turing machine $\mathcal{M}$ computing the successor function, and sample computation

configuration is $(q_s, 0^\omega \underline{0} b_1 b_2 \cdots b_n 0^\omega)$. For simplicity, we assume that inputs are always encoded as finite sequences of 1's. We let $\rightarrow^\star_{\mathcal{M}}$ denote the transitive and reflexive closure of the binary relation on configurations $\rightarrow_{\mathcal{M}}$ above. A Turing machine $\mathcal{M}$ terminates when it reaches its halting state, q_h; formally, $\mathcal{M}$ *terminates on input* $b_1 b_2 \cdots b_n$ iff $(q_s, 0^\omega \underline{0} b_1 b_2 \cdots b_n 0^\omega) \rightarrow^\star_{\mathcal{M}} (q_h, L\underline{b}R)$ for some $b \in B$ and some tape instances L and R. Note that a Turing machine cannot get stuck in any state but q_h, because the mapping M is defined everywhere except in q_h. Therefore, for any given input, a Turing machine carries out a determined succession of steps, which may or may not terminate. It is well-known that Turing machines compute exactly the partial recursive functions (see, e.g., [67]).

Therefore, a Turing machine can be regarded as an idealized, low-level programming language, which can be used for computations by placing a certain input on the tape and letting it run; if it terminates, the result of the computation can be found on the tape. Since our Turing machines have only symbols 0 and 1, one has to use them ingeniously to encode more complex inputs, such as natural numbers. There are many different ways to do this. A simple tape representation of natural numbers is to represent a number n by $n+1$ consecutive cells containing 1. With this representation for natural numbers, one can then define Turing machines corresponding to functions that take natural numbers as input and produce natural numbers as output. For example, Figure 4.2 shows a Turing machine computing the successor function. Cells containing 0 can then be used as number separators, when more natural numbers are needed. For example, a Turing machine computing a binary operation on natural numbers would run on configurations $(q_s, 0^\omega \underline{0} 1^{m+1} 0 1^{n+1} 0^\omega)$ and would halt on configurations $(q_h, 0^\omega \underline{0} 1^{k+1} 0^\omega)$, where m, n, k are natural numbers.

Exercise 139. *Define Turing machines corresponding to the addition, multiplication, and power operations on natural numbers. For example, the initial configuration of the Turing machine computing addition with 3 and 7 as input is* $(q_s, 0^\omega \underline{0} 1110111111110^\omega)$*, and its final configuration is* $(q_h, 0^\omega \underline{0} 1111111110^\omega)$.

One can similarly have tape encodings of rational numbers; for example, one can encode the number m/n as m (i.e., $m+1$ of 1) followed by n (i.e., $n+1$ of 1) with two 0 cells in-between (and keep the one-0-cell-convention for argument separation). Real numbers are not obviously representable,

though. A Turing machine is said to *compute* a real number r iff it can finitely approximate r (for example using a rational number) with any desired precision; one way to formalize this is as follows: Turing machine $\mathcal{M}_r$ computes r iff when run on input natural number p, it halts with result rational number m/n such that $|r - m/n| < 1/10^p$. If a real number can be computed by a Turing machine then it is called *Turing computable*. Many real numbers, e.g., π, e, $\sqrt{2}$, etc., are Turing computable.

Exercise 140. *Show that there are real numbers which are not Turing computable.*
(Hint: The set of Turing machines is recursively enumerable.)

4.1.2 Lazy Equational and Rewriting Representations

Our first representation of Turing machines in equational and rewriting logic is based on the idea that the infinite tape can be finitely represented by means of self-expanding stream data-structures. In spite of being infinite sequences of cells, like the Turing machine tapes, many interesting streams can be finitely specified using equations. For example, the stream of zeros, $zeros = 0 : 0 : 0 : \cdots$, can be defined as $zeros = 0 : zeros$. Since at any given moment the portions of a Turing machine tape to the left and to the right of the head have a suffix consisting of an infinite sequence of 0 cells, it is natural to represent them as streams of the form $b_1{:}b_2{:}\cdots b_n{:}zeros$. When the head is on cell b_n and the command is to move the head to the right, the self-expanding equational definition of *zeros* can produce one more 0, so that the head can move onto it. To expand *zeros* on a by-need basis and thus to avoid undesired non-termination due to the uncontrolled application of the self-expanding equation of *zeros*, the technique presented here requires a lazy rewriting strategy on streams. The technique in Section 4.1.3 requires no strategies, but it is slightly more involved.

Figure 4.3 shows how a Turing machine $\mathcal{M}$ can be associated a computationally equivalent rewriting logic theory $\mathcal{R}_{\mathcal{M}}^{lazy}$. Except for the self-expanding equation of the *zeros* stream and our stream representation of the two-infinite-end tape, the rules of $\mathcal{R}_{\mathcal{M}}^{lazy}$ are identical to the transition relation on Turing machine configurations discussed below Definition 20. The self-expanding equation of *zeros* guarantees that enough 0's can be provided when the head reaches one or the other end of the sequence of cells visited so far. The result below shows that there is a step-for-step equivalence between computations using $\mathcal{M}$ and rewrites using $\mathcal{R}_{\mathcal{M}}^{lazy}$. Moreover, when all the rules are turned into equations, the rewrite theory $\mathcal{R}_{\mathcal{M}}^{lazy}$ becomes an equational specification, which we denote $\mathcal{E}_{\mathcal{M}}^{lazy}$ and show proof-theoretically equivalent to $\mathcal{M}$:

Theorem 11. *The rewriting logic theory $\mathcal{R}_{\mathcal{M}}^{lazy}$ is confluent. Moreover, the Turing machine $\mathcal{M}$ and the rewrite theory $\mathcal{R}_{\mathcal{M}}^{lazy}$ are step-for-step equivalent, that is,*

$$(q, 0^\omega u\underline{b}v0^\omega) \rightarrow_{\mathcal{M}} (q', 0^\omega u'\underline{b'}v'0^\omega) \quad \textit{if and only if} \quad \mathcal{R}_{\mathcal{M}}^{lazy} \vDash q(\overleftarrow{u}, b : \overrightarrow{v}) \rightarrow^1 q'(\overleftarrow{u'}, b' : \overrightarrow{v'})$$

for any finite sequences of bits $u, v, u', v' \in \{0,1\}^*$, any bits $b, b' \in \{0,1\}$, and any states $q, q' \in Q$, where if $u = b_1 b_2 \ldots b_{n-1} b_n$, then $\overleftarrow{u} = b_n : b_{n-1} : \cdots : b_2 : b_1 : zeros$ and $\overrightarrow{u} = b_1 : b_2 : \cdots : b_{n-1} : b_n : zeros$. Finally, the following are equivalent:

(1) The Turing machine $\mathcal{M}$ terminates on input $b_1 b_2 \ldots b_n$;

(2) $\mathcal{R}_{\mathcal{M}}^{lazy} \vDash q_s(zeros, b_1{:}b_2{:}\cdots{:}b_n{:}zeros) \rightarrow q_h(l, r)$ for some terms l, r of sort Stream; note though that $\mathcal{R}_{\mathcal{M}}^{lazy}$ does not terminate on term $q_s(zeros, b_1{:}b_2{:}\cdots{:}b_n{:}zeros)$ as an unrestricted rewrite system, since the equation $zeros = 0 : zeros$ (regarded as a rewrite rule) can apply forever,

sorts:
Bit, $Stream$, $Configuration$
operations:
$0, 1 \ : \ \rightarrow Bit$
$zeros \ : \rightarrow Stream$
$\_:\_ \ : \ Bit \times Stream \rightarrow Stream$
$q \ : \ Stream \times Stream \rightarrow Configuration$ — one such operation for each $q \in Q$
equations:
$zeros = 0 : zeros$
rules:
$q(L, b{:}R) \rightarrow q'(L, b'{:}R)$ — one such rule for each $q, q' \in Q$, $b, b' \in Bit$ with $M(q, b) = (q', b')$
$q(L, b{:}R) \rightarrow q'(b{:}L, R)$ — one such rule for each $q, q' \in Q$, $b \in Bit$ with $M(q, b) = (q', \rightarrow)$
$q(B{:}L, b{:}R) \rightarrow q'(L, B : b{:}R)$ — one such rule for each $q, q' \in Q$, $b \in Bit$ with $M(q, b) = (q', \leftarrow)$

Figure 4.3: Lazy rewriting logic representation $\mathcal{R}_{\mathcal{M}}^{lazy}$ of Turing machine $\mathcal{M}$

thus yielding infinite equational classes of configurations with no canonical forms, but $\mathcal{R}_{\mathcal{M}}^{lazy}$ *terminates on* $q_s(zeros, b_1{:}b_2{:}\cdots{:}b_n{:}zeros)$ *if the stream construct operation* $\_:\_ : Bit \times Stream \rightarrow Stream$ *has a lazy rewriting strategy on its second argument;*

(3) $\mathcal{E}_{\mathcal{M}}^{lazy} \models q_s(zeros, b_1{:}b_2{:}\cdots{:}b_n{:}zeros) = q_h(l, r)$ *for some terms* l, r *of sort Stream; also, for the same reasons as in (2) above,* $\mathcal{E}_{\mathcal{M}}^{lazy}$ *does not terminate when its equations are used as unrestricted rewrite rules unless a lazy rewrite strategy is assumed.*

Proof. ... □

The equations in the equational specification $\mathcal{E}_{\mathcal{M}}^{lazy}$ in Theorem 11 above can be applied in any direction, so an equational proof of $\mathcal{E}_{\mathcal{M}}^{lazy} \models q_s(zeros, b_1 : b_2 : \cdots : b_n : zeros) = q_h(l, r)$ needs not necessarily correspond step-for-step to the computation of $\mathcal{M}$ on input $b_1 b_2 \ldots b_n$. On the other hand, the rewrite system $\mathcal{R}_{\mathcal{M}}^{lazy}$ faithfully captures, step-for-step, the computational granularity of $\mathcal{M}$. Recall that equational deduction does not count as computational, or rewrite steps in rewriting logic, which allows to apply the self-expanding equation of *zeros* silently in the background. Since there are no artificial rewrite steps, we can conclude that $\mathcal{R}_{\mathcal{M}}$ actually *is* precisely $\mathcal{M}$ and not an encoding of it. Theorem 12 thus showed not only that rewriting logic is Turing complete, but also that it faithfully captures the computational granularity of the represented Turing machines.

Note that in Figure 4.3 we preferred to define a configuration construct $q : Stream \times Stream \rightarrow Configuration$ for each $q \in Q$. A natural alternative could have been to define an additional sort *State* for the Turing machine states, a constant $q : \rightarrow State$ for each $q \in Q$, and one generic configuration construct $\_(\_, \_) \ : \ State \times Stream \times Stream \rightarrow Configuration$, as we do in the subsequent representation of Turing machines as rewriting logic theories (see Figure 4.4). The reason for which we did not do that here is twofold: first, in functional languages like Haskell it is very natural to associate a function to each such configuration construct $q : Stream \times Stream \rightarrow Configuration$, while it would take some additional effort to implement the second approach; second, the approach in this section is more compact than the one in Section 4.1.3.

☆ Lazy Definitions of Turing Machines in Maude

It is straightforward to define and execute Turing machines in Maude using the lazy rewriting logic approach discussed above. One can define streams of bits once and for all, as follows:

```
mod STREAM is
  sorts Bit Stream .
  ops 0 1 : -> Bit .
  op _:_ : Bit Stream -> Stream [strat(1 0)] .
  op zeros : -> Stream .
  eq zeros = 0 : zeros .
endm
```

Note the strategy attribute of the stream construct, which says that the operation is strict in its first argument but lazy in its second. This prevents the application of the self-expanding equation of `zeros` on occurrences of `zeros` within the second argument of `_:_`.

Next, one can define a Turing machine $\mathcal{M}$ in Maude by importing `STREAM`, defining operations "`q : Stream Stream -> Stream`" for all $q \in Q$, and adding all the rules as in Figure 4.3. For example, the Maude representation of the Turing machine in Figure 4.2 that computes the successor function is the following:

```
mod TURING-MACHINE-SUCC is including STREAM .
  sort Configuration .
  ops qs qh q1 q2 : Stream Stream -> Configuration .
  var L R : Stream .  var B : Bit .
  rl qs(L, 0 : R) => q1(0 : L, R) .
  rl q1(L, 0 : R) => q2(L, 1 : R) .
  rl q1(L, 1 : R) => q1(1 : L, R) .
  rl q2(L, 0 : R) => qh(L, 0 : R) .
  rl q2(B : L, 1 : R) => q2(L, B : 1 : R) .
endm
```

To execute Turing machine definitions like the above, one can rewrite terms of the form:

```
rewrite qs(zeros, 0 : 1 : zeros) .
rewrite qs(zeros, 0 : 1 : 1 : 1 : 1 : zeros) .
```

The self-expanding equation of `zeros` is not applied backwards, so resulting streams of the form `0 : zeros` are not compacted back into `zeros` (one needs specific equations/rules to do so).

Exercise 141. *Define in Maude, like above, the three Turing machines corresponding to the addition, the multiplication, and the power operations on natural numbers in Exercise 139.*

4.1.3 Unrestricted Equational and Rewriting Representations

The representation of Turing machines in Section 4.1.2 is almost as simple as it can be (there is precisely one rewrite rule for each Turing machine transition) and, additionally, can be easily executed on rewrite engines or programming languages with support for lazy rewriting or evaluation strategies, such as Maude or Haskell. However, the fact that it requires lazy rewriting or evaluation

sorts:
Bit, $Stream$, $State$, $Configuration$
operations:
$0, 1 \ : \ \rightarrow Bit$
$zeros \ : \rightarrow Stream$
$\_:\_ \ : \ Bit \times Stream \rightarrow Stream$
$\_(\_,\_) \ : \ State \times Stream \times Stream \rightarrow Configuration$
$q \ : \rightarrow State$ — one such constant for each $q \in Q$
equations:
$S(zeros, R) = S(0{:}zeros, R)$
$S(L, zeros) = S(L, 0{:}zeros)$
rules:
$q(L, b{:}R) \rightarrow q'(L, b'{:}R)$ — one such rule for each $q, q' \in Q$, $b, b' \in Bit$ with $M(q, b) = (q', b')$
$q(L, b{:}R) \rightarrow q'(b{:}L, R)$ — one such rule for each $q, q' \in Q$, $b \in Bit$ with $M(q, b) = (q', \rightarrow)$
$q(B{:}L, b{:}R) \rightarrow q'(L, B : b{:}R)$ — one such rule for each $q, q' \in Q$, $b \in Bit$ with $M(q, b) = (q', \leftarrow)$

Figure 4.4: Unrestricted rewriting logic representation $\mathcal{R}_{\mathcal{M}}$ of Turing machine $\mathcal{M}$

and that the equivalence classes of configurations have infinitely many terms, its use is limited to systems that support strategies; one is therefore still left to wonder whether Turing machines admit any elementary, unrestricted and step-for-step equivalent representations as term rewrite systems. In this section we give a positive answer to this question. The idea is very simple: replace the self-expanding and non-terminating (when regarded as a rewrite rule) equation "$zeros = 0{:}zeros$" with configuration equations of the form "$q(zeros, R) = q(0{:}zeros, R)$" and "$q(L, zeros) = q(L, 0{:}zeros)$"; these equations achieve the same role of expanding *zeros* by need, but avoid non-termination.

Figure 4.4 shows our unrestricted representation of Turing machines as rewriting logic theories. There are some minor differences between the representation in Figure 4.4 and the one in Figure 4.3. For, note that in order to add the two equations above for the expanding of *zeros* in a generic manner for any state, we separate the states from the configuration construct. In other words, instead of having an operation $q : Stream \times Stream \rightarrow Configuration$ for each $q \in Q$ like in Figure 4.3, we now have one additional sort *State*, a generic configuration construct $\_(\_,\_) \ : \ State \times Stream \times Stream \rightarrow Configuration$, and a constant $q : \rightarrow State$ for each $q \in Q$. This change still allows us to write configurations as terms $q(l, r)$, so we do not need to change the rules corresponding to the Turing machine transitions. With this modification in the signature, we can now remove the troubling equation $zeros = 0{:}zeros$ from the representation in Figure 4.3 and replace it with the two safe equations in Figure 4.4. Let $\mathcal{R}_{\mathcal{M}}$ be the rewriting logic theory in Figure 4.4 and let $\mathcal{E}_{\mathcal{M}}$ be the equational theory obtained from $\mathcal{R}_{\mathcal{M}}$ by replacing each of its rules by an equation.

Theorem 12. *The rewriting logic theory $\mathcal{R}_{\mathcal{M}}$ is confluent. Moreover, the Turing machine $\mathcal{M}$ and the rewrite theory $\mathcal{R}_{\mathcal{M}}$ are step-for-step equivalent, that is,*

$$(q, 0^{\omega} u \underline{b} v 0^{\omega}) \rightarrow_{\mathcal{M}} (q', 0^{\omega} u' \underline{b'} v' 0^{\omega}) \quad \textit{if and only if} \quad \mathcal{R}_{\mathcal{M}} \vDash q(\overleftarrow{u}, b : \overrightarrow{v}) \rightarrow^{1} q'(\overleftarrow{u'}, b' : \overrightarrow{v'})$$

for any finite sequences of bits $u, v, u', v' \in \{0,1\}^*$, any bits $b, b' \in \{0,1\}$, and any states $q, q' \in Q$, where if $u = b_1 b_2 \ldots b_{n-1} b_n$, then $\overleftarrow{u} = b_n : b_{n-1} : \cdots : b_2 : b_1 : zeros$ and $\overrightarrow{u} = b_1 : b_2 : \cdots : b_{n-1} : b_n : zeros$. Finally, the following are equivalent:

(1) The Turing machine $\mathcal{M}$ terminates on input $b_1 b_2 \dots b_n$;

(2) $\mathcal{R}_\mathcal{M}$ terminates on term $q_s(zeros, b_1 : b_2 : \cdots : b_n : zeros)$ as an unrestricted rewrite system and $\mathcal{R}_\mathcal{M} \vDash q_s(zeros, b_1 : b_2 : \cdots : b_n : zeros) \to q_h(l, r)$ for some terms l, r of sort Stream;

(3) $\mathcal{E}_\mathcal{M} \vDash q_s(zeros, b_1 : b_2 : \cdots : b_n : zeros) = q_h(l, r)$ for some terms l, r of sort Stream.

Proof. ... (proof below taken from ICFP'06 paper, needs to be revised) ...

Since for any $q \in Q$ different from q_h there is precisely one rule whose left-hand side (left-hand side) has the form $q(L, 0 : R)$ and precisely one whose left-hand side has the form $q(L, 1 : R)$, it follows that the left-hand side-es of the rules in the first group (treating the situations in which the tapes are not *nil* when their first elements are requested) cannot overlap. Also, note that left-hand side-es of the rules in the first group cannot overlap with those in the second group, because the right list arguments are non-*nil* in the former while, except for the last, they are *nil* in the second; the last rule in the second group could only possibly overlap with the last rule in the first group, but that is not possible either because of their left list arguments (one is *nil* while the other is non-*nil*). Finally, note that the rules in the second group cannot overlap with each other either; the only ones which could possibly overlap are the last three, but the *nil* vs. non-*nil* characteristics of their list arguments exclude each other. Therefore, $\mathcal{R}_\mathcal{M}$ is orthogonal, so confluent.

It is clear that any computation in $\mathcal{M}$ can be seamlessly simulated by $\mathcal{R}_\mathcal{M}$; indeed, the computation in $\mathcal{M}$ on an input $b_1 b_2 \dots b_n$ can be simulated *step-by-step* by $\mathcal{R}_\mathcal{M}$, starting with the term $q_s(nil, b_1 : b_2 : \cdots : b_n : nil)$. Also, any rewrite sequence in $\mathcal{R}_\mathcal{M}$ generates stepwise a corresponding computation in $\mathcal{M}$, by simply concatenating the reversed left list with the right one, and replacing the two *nil*s by infinite streams of zeros. Consequently, $\mathcal{M}$ reaches its state q_h during a computation if and only if the corresponding rewriting sequence in $\mathcal{R}_\mathcal{M}$ ends with a term of the form $q_h(L, R)$. The equivalence of *(1)* and *(2)* above follows from the fact that there is only one way to reduce the term $q_s(nil, b_1 : b_2 : \cdots : b_n : nil)$ to 1, namely reducing it to $q_h(L, R)$ and then in one step to 1, and the equivalence of *(2)* and *(3)* follows by the Church-Rosser property of equational logic and the confluence of $\mathcal{R}_\mathcal{M}$. □

Like for the other representation of Turing machines in rewriting logic discussed in Section 4.1.2, the rewrite theory $\mathcal{R}_\mathcal{M}$ *is* the Turing machine $\mathcal{M}$, in that there is a step-for-step equivalence between computational steps in $\mathcal{M}$ and rewrite steps in $\mathcal{R}_\mathcal{M}$. Recall, again, that equations do not count as rewrite steps, their role being to structurally rearrange the term so that rewrite rules can apply; indeed, that is precisely the intended role of the two equations in Figure 4.4 (they reveal new blank cells on the tape whenever needed). One could, however, argue that $\mathcal{R}_\mathcal{M}$ regarded as an ordinary term rewrite system and not as a theory in rewriting logic, does not faithfully capture $\mathcal{M}$ step-for-step because the applications of the two equations as rewrite rules are an artifact of our representation and have no equivalent in the original Turing machine $\mathcal{M}$. In fact, the equations can be completely eliminated, at the expense of more rules. For example, if $M(q, b) = (q', \to)$ then, in addition to the last rule in Figure 4.4, we can also include the rule:

$$q(zeros, b : R) \to q'(zeros, 0 : b : R)$$

This way, one can expand *zeros* and apply the transition in *one* rewrite step, instead of one equational step and one rewrite step. Doing that systematically for all the transitions allows us to eliminate the need for equations entirely; the price to pay is, of course, that the number of rules increases.

Exercise 142. *Eliminate the two equations in Figure 4.3 as discussed above and prove that the resulting term rewrite system is confluent and captures the computation in $\mathcal{M}$ faithfully, step-for-step.*

☆ Unrestricted Definitions of Turing Machines in Maude

The unrestricted representation of Turing machines in rewriting logic discussed above can be easily defined and then executed on any rewrite system, with or without support for rewrite strategies. We here show one way to do in Maude. Since the self-expanding equation of *zeros* is not needed anymore, we can now define streams as a plain algebraic signature, with no strategies or equations:

```
mod STREAM is
  sorts Bit Stream .
  ops 0 1 : -> Bit .
  op _:_ : Bit Stream -> Stream .
  op zeros : -> Stream .
endm
```

The two new equations in Figure 4.4 can be defined generically as follows, for any state:

```
mod TURING-MACHINE is including STREAM .
  sorts State Configuration .
  op _‘(_,_‘) : State Stream Stream -> Configuration .
  var S : State .  var L R : Stream .
  eq S(zeros,R) = S(0 : zeros, R) .
  eq S(L,zeros) = S(L, 0 : zeros) .
endm
```

Particular Turing machines can now be defined by including the module `TURING-MACHINE` above and adding specific states and rules. For example, below is the Maude definition of a Turing machine computing the successor function (see Figure 4.2):

```
mod TURING-MACHINE-SUCC is including TURING-MACHINE .
  ops qs qh q1 q2 : -> State .
  var L R : Stream .  var B : Bit .
  rl qs(L, 0 : R) => q1(0 : L, R) .
  rl q1(L, 0 : R) => q2(L, 1 : R) .
  rl q1(L, 1 : R) => q1(1 : L, R) .
  rl q2(    L, 0 : R) => qh(L, 0 : R) .
  rl q2(B : L, 1 : R) => q2(L, B : 1 : R) .
endm
```

Exercise 143. ☆ *Define the four Turing machines in Exercise 139 as equational specifications in Maude using the representation in Figure 4.4 and execute them using Maude's rewrite engine on several examples.*

Exercise 144. ☆ *Same as Exercise 143, but using the representation in Exercise 142.*

4.1.4 Notes

The abstract theoretical machine that today we call Turing machine has been proposed by Alan Turing, a British mathematician, in 1936-37 [75]. In the original article, Turing imagined not a mechanical machine, but a person whom he calls the "computer", who executes these deterministic

mechanical rules "in a desultory manner". In his 1948 essay "Intelligent Machinery", Turing calls the machine he proposed the *logical computing machine*, a name which has not been broadly adopted, everybody preferring to call it the *Turing machine*.

It may be interesting to understand the scientific context in which Turing proposed his machine. In the 1930s, there were several approaches attempting to address Hilbert's tenth question of 1900, the *Entscheidungsproblem* (the *decision problem*). Partial answers have been given by Gödel in 1930 (and published in 1931), under the form of his famous *incompleteness theorems*, and in 1934, under the form of his *recursion theory*. Alonzo Church is given the credit for being the first to effectively show that the Entscheidungsproblem was indeed *undecidable*, introducing also λ-calculus (discussed in Section 4.4); Church published his results in 1936, about one month before Turing submitted his paper. In his paper, Turing also proved the equivalence of his machines to Church's λ-calculus. Interestingly, Turing submitted his paper a few months before Emil Post, another great logician, submitted for publication a paper independently proposing yet another computational model that turned out to be equivalent to Turing machines and λ-calculus.

Even though Turing was not the first to propose what we call today a Turing-complete model of computation, many believe that his result was stronger than Church's, in that his computational model was more direct, easier to understand, and based on first, low-level computational principles. For example, it is typically straightforward to implement Turing Machines in any programming language, which is not necessarily the case for other Turing-complete models, such as, for example, the popular λ-calculus. As seen in Section 4.4, λ-calculus relies on a non-trivial notion of substitution, which comes with the infamous *variable capture* problem.

The original machine proposed by Turing had its tape infinite to only one end. Our Turing machine variant in this section, which is entirely equivalent to the original Turing machine, had the tape infinite at both ends and it was inspired by the book by Rogers [67].

Encodings of Turing computability into equational deduction and/or rewriting are well-known. For example, Bergstra and Tucker [9] (1995) and Baader and Nipkow [2] (1998) give encodings that have properties similar to ours in this section, in the sense that their resulting equational specifications, if regarded as term rewrite systems, are confluent and terminate whenever the original computation terminates. It seems, however, that due to the particular two-infinite-end-tape variant of Turing machine chosen in this chapter, our encoding is somewhat simpler. The ideas underlying our encodings presented in this section ware first published by the author in 2006 [65].