Mechanizing Undecidable Properties of Rewriting Systems using the Rocq Prover

Lecture 4: Turing Machine Halting Reduces to String Rewriting

Andrej Dudenhefner

ISR 2026 – 15th International School on Rewriting

2026-07-13

Course Roadmap

1. Formalizing string rewriting (SR) in Rocq
2. Synthetic computability
3. Undecidability of Turing machine halting
4. **Turing machine halting $\preceq$ SR**
   - Encode a TM run as a rewriting derivation
   - Build the reduction SBTM_HALT $\preceq$ SR
   - Both directions: every step simulated, and *nothing else* can happen
   - Conclude: `undecidable SR`

Encoding Configurations as Strings

From Tape to String: The Symbols

We work over `nat` (the SR alphabet). Reserve a handful of symbols:

```
Notation "⟦" := 0.    (* left end marker  *)
Notation "⟧" := 1.    (* right end marker *)
Definition encode_symbol (a: bool) := if a then 2 else 3.
Definition encode_state (q: state M) := 4 + proj1_sig (Fin.to_nat q).
```

- `0`, `1`: end markers $[\![\ldots]\!]$ delimiting the used tape
- `2`, `3`: tape symbols `true`/`false`
- `4`, `5`, `...`: one symbol per state, kept disjoint from the rest

A *fresh symbol* for every syntactic role: the countably infinite `nat` alphabet pays off.

From Tape to String: The Configuration

```
Definition encode_config (q : state M) (t : tape) : SR.string nat :=
  match t with
  | (ls, a, rs) =>
      ⟦ :: rev (map encode_symbol ls)
        ++ [encode_symbol a] ++ [encode_state q]
        ++ map encode_symbol rs ++ [⟧]
  end.
```

$$⟦\ \underbrace{\overline{l_k}\cdots\overline{l_1}}_{\text{rev } ls}\ \ \overline{a}\ \ \mathbf{q}\ \ \underbrace{\overline{r_1}\cdots\overline{r_m}}_{rs}\ ⟧$$

- the state symbol **q** sits *immediately right* of the head symbol $\overline{a}$
- `ls` is reversed: tape positions read left-to-right in the string
- only finitely many cells appear; markers fence in the rest

Worked Encoding

Machine in state `q` (say `encode_state q = 4`), tape $\ldots$ `t` **`f`** `t` $\ldots$
with head on the `false`, left part `[true]`, right part `[true]`:

$$\texttt{(ls, a, rs)} = \texttt{([true], false, [true])}$$

encodes to

$$[\![\overbrace{2}^{\text{true}} \ \overbrace{3}^{\text{false}} \ \overbrace{4}^{\textbf{q}} \ \overbrace{2}^{\text{true}}]\!] \quad = \quad \texttt{[0; 2; 3; 4; 2; 1]}$$

- head symbol `3` (the `false`), state `4` right after it
- `rev (map _ [true]) = [2]` on the left, `map _ [true] = [2]` on the right

The Rule Construction

One Transition, A Handful of Rules

`encode_rule (q, a)` inspects `trans' M (q, a)` and emits local rewrite rules:

```
Definition encode_rule '(q, a) :=
  match trans' M (q, a) with
  | Some (q', a', go_left) => [
      ([⟦; #a; enc q], [⟦; #false; enc q'; #a']);
      ([#true;  #a; enc q], [#true;  enc q'; #a']);
      ([#false; #a; enc q], [#false; enc q'; #a'])]
  | Some (q', a', go_right) => [ (* analogous *) ... ]
  | None => [ (* halting rules, see next slide *) ... ]
  end.
```

- `#a` = `encode_symbol a`, `enc q` = `encode_state q`
- the state symbol moves *left* past the neighbour: `[#b; #a; q] -> [#b; q'; a']`
- the marker case ⟦ handles the blank conjured at the tape edge

The Move Cases in Detail

Move left, head over `a`, neighbour `b` to the left:

$$\cdots\ \overline{b}\,\overline{a}\,\mathbf{q}\ \cdots\ \rightarrow\ \cdots\ \overline{b}\,\mathbf{q}'\,\overline{a'}\ \cdots$$

The state symbol hops over `a` (now rewritten `a'`) to land left of it.
At the left marker, `b` is a fresh blank: `[⟦; #a; q]` becomes `[⟦; #false; q'; #a']`.

Move right, analogous, using the right marker ⟧.
The state symbol moves *right*, past the right neighbour.

The Halting Cases

When `trans' M (q, a) = None` the configuration must rewrite to a fixed target.

```
| None => [
    ([⟦; #a; enc q; ⟧], [⟦; ⟧]);          (* collapse *)
    ([#true;  #a; enc q], [#a; enc q]);    (* eat left  *)
    ([#false; #a; enc q], [#a; enc q]);
    ([#a; enc q; #true],  [#a; enc q]);    (* eat right *)
    ([#a; enc q; #false], [#a; enc q])]
```

- "eat" rules delete neighbours one by one, shrinking the tape
- once head + state sit alone between markers, "collapse" fires
- the canonical halting witness is the string $⟦⟧ =$ `[1; 0]`

Assembling the System

```
Definition srs : SR.SRS nat :=
  flat_map encode_rule
    (list_prod (all_fins (num_states M)) [true; false]).
```

- `all_fins n`: lists every state, `all_fins_spec` proves completeness
- `list_prod`: all `(state, symbol)` pairs, the whole transition table
- `flat_map encode_rule`: concatenate each pair's local rules

Correctness: Both Directions

Forward: Simulation

Every machine step is a single rewrite step; halting reaches the witness.

```
Lemma simulation_step q t q' t' : step M (q, t) = Some (q', t') ->
  rew srs (encode_config q t) (encode_config q' t').

Lemma simulation_halt q t : step M (q, t) = None ->
  rewt srs (encode_config q t) [⟦]; ⟦].

Lemma simulation q t k : steps M k (q, t) = None ->
  rewt srs (encode_config q t) [⟦]; ⟦].
```

- `simulation_step`: pick the right `In_srs` rule, frame with `rewB`
- `simulation_halt`: induct on the tape, "eat" neighbours, then "collapse"
- `simulation`: induct on `k` via `steps_plus`, chain with `rewS`

Backward: Inverse Simulation

The hard direction: the rewriting system can do *nothing but* simulate.

```
Lemma inverse_simulation_step q t x :
  rew srs (encode_config q t) x ->
  step M (q, t) = Some (q', t') -> x = encode_config q' t'.

Lemma inverse_simulation q t :
  rewt srs (encode_config q t) [⟧; ⟦] ->
  exists k, steps M k (q, t) = None.
```

Why could it fail?

A rule might fire at a *unwanted* position, or the string might decode to no configuration at all.

We must rule that out.

The Invariant: Exactly One State Symbol

`encode_config` places **exactly one** state symbol, right after the head.

```
Lemma encode_config_eq_app q ls a rs u v a' q' :
  encode_config q (ls, a, rs) = u ++ [#a'; enc q'] ++ v ->
  q = q' /\ a = a' /\
  u = ⟦ :: rev (map encode_symbol ls) /\ v = map encode_symbol rs ++ []⟧.
```

- if a `[#a'; enc q']` pattern appears, it *is* the head/state pair
- proof: no `encode_state` symbol hides among the encoded tape symbols (2/3) or markers, so the only match is the real one
- consequence: a rule's left-hand side, which always contains `enc q`, can match at *one* place only

This is what makes `inverse_simulation_step` deterministic: rewriting `encode_config q t` forces the genuine machine step.

Putting It Together

The reduction function: `M, (q,t) ↦ (srs M, encode_config M q t, [1;0])`.

```
Theorem reduction : SBTM_HALT ⪯ SR.SR.
Proof.
  exists (fun '(existT _ M (q, t)) =>
            (srs M, encode_config M q t, [1; 0])).
  move=> [M [q t]] /=. split.
  - by move=> [?] /simulation.       (* forward  *)
  - by apply: inverse_simulation.    (* backward *)
Qed.
```

Summary

Summary

- configurations encoded as marked strings, state symbol pinpointing the head
- `encode_rule`: local rules per transition; "eat" + "collapse" on halt
- simulation (forward) + inverse simulation (backward, via the one-state-symbol invariant)
- `SBTM_HALT` $\preceq$ `SR`
- `undecidable SR`

Exercise session (now):

1. run `encode_config` on small machines by hand
2. single-step `simulation_step` derivations
3. the eat/collapse halting derivation for a tiny halting machine