Mechanizing Undecidable Properties of Rewriting Systems using the Rocq Prover

Lecture 1: Formalizing String Rewriting

Andrej Dudenhefner

ISR 2026 – 15th International School on Rewriting

2026-07-12

Course Roadmap

1. **Formalizing string rewriting (SR) in Rocq**
   - Strings as lists over `nat`
   - Rewrite systems as lists of pairs of strings
   - Rewriting relation as inductive predicate
   - Specification of the string rewriting decision problem
2. Synthetic computability
3. Undecidability of Turing machine halting
4. Turing machine halting $\preceq$ SR

String Rewriting in Rocq

String Rewriting

A **string rewriting system** (semi-Thue system) over alphabet Σ:

$$R \subseteq \Sigma^* \times \Sigma^* \qquad \text{rules written } u/v$$

One rewriting step:

$$x\,u\,y \;\to_R\; x\,v\,y \qquad \text{if } u/v \in R$$

Problem (String Rewriting (SR))

Given a triple (R, x, y), does $x \to_R^* y$ hold?

Example

For $R := \{ab/ba\}$ we have: $aabb \to_R abab \to_R abba \to_R baba \to_R bbaa$.

Known since Post (1947): **SR is undecidable.**
What does that statement *formally* mean, and how do we state it in Rocq?

Strings and Systems as Data

Everything is concrete, finite data, no abstract alphabets.

```
(* A string is a list of symbols. *)
Abbreviation string X := (list X).
(* A rule is a pair. *)
Notation "x / y" := (x, y).
(* A string rewriting system is a list of rules. *)
Abbreviation SRS X := (list (string X * string X)).
```

- Symbols: `nat`
  - countably infinite alphabet
  - fresh symbols for free
- Rules: pairs of strings, written `u / v`
- Systems: *lists* of rules, not sets
  - finite by construction
  - membership via `In`

Inference Rules as `Inductive` Predicates

On paper

One step $\to_R$:

$$\textsc{rewB}\ \frac{u/v \in R}{x\,u\,y\ \to_R\ x\,v\,y}$$

Closure $\to_R^*$:

$$\textsc{rewR}\ \frac{}{z \to_R^* z} \qquad \textsc{rewS}\ \frac{x \to_R y \qquad y \to_R^* z}{x \to_R^* z}$$

In Rocq

```
Inductive rew (R : SRS X)
  : string X -> string X -> Prop :=
| rewB x y u v :
    In (u / v) R ->
    rew R (x++u++y) (x++v++y).

Inductive rewt (R : SRS X)
  : string X -> string X -> Prop :=
| rewR z : rewt R z z
| rewS x y z :
    rew R x y -> rewt R y z ->
    rewt R x z.
```

- Premises above the line $\rightsquigarrow$ hypotheses (arguments) of the constructor
- Conclusion below the line $\rightsquigarrow$ the constructor's result type

Inference Rules as `Inductive` Predicates

```
Inductive rew {X} (R : SRS X) : string X -> string X -> Prop :=
  rewB x y u v : In (u / v) R ->
                 rew R (x ++ u ++ y) (x ++ v ++ y).

Inductive rewt {X} (R : SRS X) : string X -> string X -> Prop :=
  rewR z : rewt R z z
| rewS x y z : rew R x y -> rewt R y z -> rewt R x z.
```

- `rew`: exactly one constructor, exactly one way to make a step
- `rewt R x y` is evidence of $x \to_R^* y$

Example

For $R := \{ab/ba\}$ we have: $aabb \to_R abab \to_R abba \to_R baba \to_R bbaa$.
Using $a := 0$ and $b := 1$ this means: `rewt [([0;1], [1;0])] [0;0;1;1] [1;1;0;0].`

Rocq Interlude: `Inductive` vs. `Definition`

`Inductive` predicate

- generated by constructors
- *smallest* relation closed under the rules
- proofs by `constructor` / case analysis / `induction`

`Definition`

- just an abbreviation
- unfolds definitionally

```
(* Given a triple (R, x, y), does rewt R x y hold? *)
Definition SR : SRS nat * string nat * string nat -> Prop :=
  fun '(R, x, y) => rewt R x y.
```

A decision **problem** is a predicate on instances.
Here: triples (R, x, y).

Summary

Summary

- SR formalized: `list nat` strings, rule lists, inductive `rewt`

Exercise session (now):

1. concrete `rew`/`rewt` derivations for small systems
2. `rewt` facts: transitivity, `rewt_subset`