Format for describing reductions of the word reachability problem

1 Basic overview

Given an alphabet Σ, the word reachability problem can be defined as the set $\mathsf{WP} = \{\langle u, v, R\rangle \mid u, v \in \Sigma^* \wedge R \subset \Sigma^* \times \Sigma^* \wedge u \to_R^* v\}$, i.e., as the set of tuples $\langle u, v, R\rangle$ where the word u can be rewritten into the word v by using the rewrite rules of R. This is a classical undecidable set, which can be reduced to other sets to prove their undecidability. For instance, to reduce WP to $\{\langle u', v', w', R'\rangle \mid u', v', w' \in \Sigma'^* \wedge R' \subset \Sigma'^* \times \Sigma'^* \wedge u' \to_{R'}^* v' \to_{R'}^* w'\}$ and prove its undecidability we could use the reduction:

$$\langle u, v, R\rangle \;\mapsto\; \langle u, v, \#, R \uplus \{v \to \#\}\rangle$$

where $\#$ is a new alphabet symbol not in Σ. This reduction can be described with the following program:

```
u
v
#
l -> r
v -> #
```

where the first three lines define u', v', w', respectively, and the last two lines define R'. The following sections describe the syntax used for defining words and rewrite rules.

2 Words

Words are defined with non-empty[1] strings over the following characters:

a, **b**, **c**, **d**, **e**, **f**, **g**, **0**, ..., **9**, **#**, **$**, **&**, **@**, **u**, **v**, **w**, **x**, **y**, **z**

where the characters **a**, **b**, **c**, **d**, **e**, **f**, **g**, **0**, ..., **9**, **#**, **$**, **&**, **@** are interpreted as alphabet symbols of Σ', the character **u** is interpreted as the word u of the tuple $\langle u, v, R\rangle$ over which the reduction is applied, and similarly, the character **v** is interpreted as the word v. For exercises where the input tuple has more than the two words u, v, these extra words can be referenced with the characters **w**, **x**, **y**, **z**. Hence, note that the initial example program defines u' as the word u (line 1), v' as the word v (line 2), and w' as the word $\#$ (line 3).

3 Rewrite rules

Rewrite rules are defined in the lines containing the token **->**. A rule definition consists of two words, one at the left-hand side of **->** and one at its right-hand side. Such words are non-empty strings over alphabet symbols (**a**, **b**, **c**, **d**, **e**, **f**, **g**, **0**, ..., **9**, **#**, **$**, **&**, **@**), the special characters **u**, **v**, **w**, **x**, **y**, **z** to refer to the words over which the reduction is applied, and additionally, the special characters **l**, **r**, **i**. A rule definition including any of these last three special characters **l**, **r**, or **i** is actually defining multiple rules simultaneously, one rule for each rule $l \to r$ in the R over which the reduction is applied: each occurrence of

[1] In the definition of WP we have considered that both u and v might be empty. However, for the input of the reductions we consider a variant of WP where the variables satisfy $u, v \in \Sigma^+$ and $R \subset \Sigma^+ \times \Sigma^+$, i.e., the empty word is excluded, even for the rewrite rules. This simplification is not significant from the point of view of decidability, but eases the description of the reductions.

the character `l` is substituted by l, each occurrence of the character `r` is substituted by r, and each occurrence of `i` is substituted by a special symbol that uniquely identifies the rule $l \to r$. For example, if $R = \{a \to aab, aa \to b, bb \to bab\}$, then:

- `l -> r` becomes the original rules: $\begin{bmatrix} a \to aab \\ aa \to b \\ bb \to bab \end{bmatrix}$
- `r -> l` becomes the original rules inverted: $\begin{bmatrix} aab \to a \\ b \to aa \\ bab \to bb \end{bmatrix}$
- `#l -> $r$` is the original rules with symbols added: $\begin{bmatrix} \#a \to \$aab\$ \\ \#aa \to \$b\$ \\ \#bb \to \$bab\$ \end{bmatrix}$
- `l -> i` sets the r.h.s. to symbols identifying the original rules: $\begin{bmatrix} a \to \mathfrak{s}_1 \\ aa \to \mathfrak{s}_2 \\ bb \to \mathfrak{s}_3 \end{bmatrix}$
- `i -> r` sets the l.h.s. to symbols identifying the original rules: $\begin{bmatrix} \mathfrak{s}_1 \to aab \\ \mathfrak{s}_2 \to b \\ \mathfrak{s}_3 \to bab \end{bmatrix}$

(The $\mathfrak{s}_1, \mathfrak{s}_2, \mathfrak{s}_3$ are distinct symbols of Σ', different from `a`, `b`, `c`, `d`, `e`, `f`, `g`, `0`, ..., `9`, `#`, `$`, `&`, `@`, that uniquely identify the rules in R.) Hence, in the initial example program, `l -> r` (line 4) adds to R' all the original rules, and `v -> #` (line 5) adds to R' a rule $v \to \#$.

4 Morphisms

In some cases, it is necessary or convenient to use a morphism to describe the reduction. The program assumes the existence of a morphism `s` that is initially defined as the identity for all alphabet symbols. This morphism can be updated with lines of the form:

`s(`a`) =` w

to specify that, from that line onwards, the image of the alphabet symbol a is the (non-empty) word w. The morphism can be applied to any word α with the syntax `s(`α`)`. For example, if $u = aaa$, $v = bba$, and $R = \{a \to aab, aa \to b, bb \to bab\}$, the program

```
s(u)
s(a) = #
s(v)
s(b) = $
as(l) -> s(r)a
```

is executed as follows:

- line 1: defines the word `aaa`.
- line 2: specifies that the image of the symbol `a` is now `#`.
- line 3: defines the word `bb#`.
- line 4: specifies that the image of the symbol `b` is now `$`.
- line 5: defines the rules `a#`→`##$a`, `a##`→`$a`, and `a$$`→`$#$a`.