Rewriting Logic from a ρLog Point of View

Mauricio Ayala-Rincón

Department of Computer Science
University of Brasilia, Brazil

Besik Dundua

Ilia Vekua Institute of Applied Mathematics
Ivane Javakhishvili Tbilisi State University, Georgia

Temur Kutsia

Research Institute for Symbolic Computation
Johannes Kepler University Linz, Austria

Mircea Marin

Department of Computer Science
West University of Timişoara, Romania

Abstract

Rewriting logic is a well-known logic that emerged as an adequate logical and semantic framework for the specification of languages and systems. ρLog is a calculus for rule-based programming with labeled rules. Its expressive power stems from the usage of a fragment of higher-order logic (e.g., sequence variables, and function variables) to express atomic formulas. Its adequacy as a computational model for rule-based programming is derived from theoretical results concerning E-unification and E-matching in the fragment of logic adopted by ρLog.
In this paper we choose a fragment of the ρLog calculus and argue that it can be used to perform deduction in rewriting logic. More precisely, we define a mapping between the entailment systems of rewriting logic and ρLog for which the conservativity theorem holds. It implies that, like rewriting logic, ρLog also can be used as a logical and semantic framework.

1 Introduction

Rewriting logic [19] emerged as a simple computational logic based on the use of rewrite theories to represent with great generality (1) various models of computation (concurrency, programming languages, etc.), and (2) logical deduction. For computation, it represents states by equivalence classes in an equational theory, and local concurrent transitions by rewrite rules. For deductive purposes, it can represent formula-based data structures (e.g., sequents or sets of formulas) by terms, and the inference rules of the logic by conditional rewrite rules. The rewrite theories, which

are at the core of this logic, provide an adequate representation for a wide variety of applications, including automated deduction, software and hardware specification and verification, security, real-time and cyber-space systems, probabilistic systems, bioinformatics, and chemical systems. (See [19] for a convincing account.) Rewriting logic is the theoretical basis of Maude [3], a powerful reflective language with wide range of applications.

In [15], the authors described rewriting logic as a logical and semantic framework. They showed that it has a flexibility to represent in a natural way many other logics, maintaining the direct correspondence between proofs in object logics and proofs in rewriting logic (as the framework logic). This correspondence is often conservative, given by means of maps of logics, so that an implementation of the object logic is directly supported by an implementation of rewriting logic. Besides, the authors explored similarities of rewriting logic with Milner's CCS, concurrent object-oriented programming, and structural operational semantics.

An interesting refinement of rewriting logic, inspired by the use of rewriting logic as a logical framework for deduction, was to make a clear *separation of concerns* between the specification of the inference system and the heuristics which guide the way in which rules are applied. This point of view introduced the usage of strategy languages to define theory transformations parameterized by strategy modules [16].

ρLog [13,14] is a system for rule-based programming with labeled rules based on a calculus which makes all the ingredients of rewriting logic explicit. Terms, conditional rewrite rules, and strategies that specify the heuristics which guide the way in which rules are applied, can all be explicitly represented in its syntax. A novelty of ρLog is its definition in a fragment of logic with sequence variables, function variables, context variables, and membership constraints for their bindings in the rewrite process. These capabilities make the rule-based specifications of ρLog natural and concise. The calculus served as the basis for a strategy-based programming tool [12,7] and has found applications in constraint logic programming [5], XML transformation and Web reasoning [4], modeling rewriting strategies [6], in extraction of frequent patterns from data mining workflows [20], and for automatic derivation of multiscale models of arrays of micro- and nanosystems [2].

In this paper we choose a fragment of the ρLog calculus and argue that it can be used to perform deduction in rewriting logic. More precisely, we define a mapping between the entailment systems of rewriting logic and ρLog for which the conservativity theorem holds. It implies that, like rewriting logic, ρLog also can be used as a logical and semantic framework.

The paper is organized as follows: In Section 2 we review syntax and semantics of rewriting logic. ρLog is introduced in Section 3. Section 4 is the main part of the paper, where the mapping between the entailment systems of these two formalisms is defined. Section 5 concludes.

2 Syntax and Inference System of Rewriting Logic

In this section, we mainly follow the description of rewriting logic as it is given in [18]. The syntax of rewriting logic is given by *signatures*, which are pairs (F, E) of a set of ranked function symbols F and a set of equations E. Given a countable

set of variables V, terms over F and V are defined in the usual way:

$$\mathsf{t} ::= \mathsf{x} \mid \mathsf{f}(\mathsf{t}_1, \ldots, \mathsf{t}_n),$$

where $\mathsf{x} \in \mathsf{V}$ and $\mathsf{f} \in \mathsf{F}$ is n-ary. The set of terms over F and V is denoted by $\mathcal{T}(\mathsf{F}, \mathsf{V})$. The letters $\mathsf{t}, \mathsf{r}, \mathsf{s}$ and u are used to denote its elements, while $\mathsf{x}, \mathsf{y}, \mathsf{z}$ stand for variables.

The equivalence class of a term t modulo E is denoted by $[\mathsf{t}]_{\mathsf{E}}$. The subscript E is usually omitted, when it causes no confusion. The set of E-equivalence classes of terms from $\mathcal{T}(\mathsf{F}, \mathsf{V})$ is denoted by $\mathcal{T}_{\mathsf{E}}(\mathsf{F}, \mathsf{V})$.

Given a signature (F, E), the considered *sentences* are sequents $[\mathsf{t}] \longrightarrow [\mathsf{r}]$.

A *substitution* σ is a mapping from variables to terms such that all but finitely many variables are mapped to themselves. Each substitution σ is represented as a finite set of pairs $\{\mathsf{x}_1 \mapsto \sigma(\mathsf{x}_1), \ldots, \mathsf{x}_n \mapsto \sigma(\mathsf{x}_n)\}$ where the x's are all the variables for which $\sigma(\mathsf{x}_i) \neq \mathsf{x}_i$.

A rewrite theory $\mathcal{R}$ is a 4-tuple $\mathcal{R} := (\mathsf{F}, \mathsf{E}, \mathsf{L}, \mathsf{R})$, where F and E are sets of function symbols, L is a set of labels, and R is a set of conditional rewrite rules. The latter are defined as pairs of a label and a nonempty sequence of pairs of E-equivalence classes of terms from $\mathcal{T}(\mathsf{F}, \mathsf{V})$. Usually, a rewrite rule of the form $(\mathsf{l}, ([\mathsf{t}_0], [\mathsf{r}_0])([\mathsf{t}_1], [\mathsf{r}_1]) \cdots ([\mathsf{t}_n], [\mathsf{r}_n]))$ is written as

$$\mathsf{l} : [\mathsf{t}_0] \to [\mathsf{r}_0] \text{ if } [\mathsf{t}_1] \to [\mathsf{r}_1] \wedge \cdots \wedge [\mathsf{t}_n] \to [\mathsf{r}_n],$$

where $[\mathsf{t}_1] \to [\mathsf{r}_1] \wedge \ldots \wedge [\mathsf{t}_n] \to [\mathsf{t}_n]$ is called the condition of the rule.

A rewrite theory $\mathcal{R}$ *entails* a sequent $[\mathsf{t}] \longrightarrow [\mathsf{r}]$, written $\mathcal{R} \vdash [\mathsf{t}] \longrightarrow [\mathsf{r}]$, iff $[\mathsf{t}] \longrightarrow [\mathsf{r}]$ can be proved by finite application of the following four inference rules:

Reflexivity: For each $[\mathsf{t}] \in \mathcal{T}_E(\mathsf{F}, \mathsf{V})$,

$$\frac{}{[\mathsf{t}] \longrightarrow [\mathsf{t}]}.$$

Congruence: For each $\mathsf{t}_1, \ldots, \mathsf{t}_n, \mathsf{r}_1, \ldots, \mathsf{r}_n \in \mathcal{T}_E(\mathsf{F}, \mathsf{V})$ and $f \in \mathsf{F}$ with the arity $n \geqslant 1$,

$$\frac{[\mathsf{t}_1] \longrightarrow [\mathsf{r}_1] \quad \cdots \quad [\mathsf{t}_n] \longrightarrow [\mathsf{r}_n]}{[f(\mathsf{t}_1, \ldots, \mathsf{t}_n)] \longrightarrow [f(\mathsf{r}_1, \ldots, \mathsf{r}_n)]}.$$

Replacement: For each rule $\mathsf{l} : [\mathsf{t}_0] \to [\mathsf{r}_0]$ if $[\mathsf{t}_1] \to [\mathsf{r}_1] \wedge \cdots \wedge [\mathsf{t}_n] \to [\mathsf{r}_n] \in R$ and for each terms $\mathsf{s}_1, \ldots, \mathsf{s}_k, \mathsf{u}_1, \ldots, \mathsf{u}_k \in \mathcal{T}_E(\mathsf{F}, \mathsf{V})$.

$$\frac{[\mathsf{s}_1] \longrightarrow [\mathsf{u}_1] \quad \cdots \quad [\mathsf{s}_k] \longrightarrow [\mathsf{u}_k] \quad [\mathsf{t}_1\sigma] \longrightarrow [\mathsf{r}_1\sigma] \quad \cdots \quad [\mathsf{t}_n\sigma] \longrightarrow [\mathsf{r}_n\sigma]}{[\mathsf{t}_0\sigma] \longrightarrow [\mathsf{r}_0\vartheta]},$$

where the set of variables occurring in the rule is $\{\mathsf{x}_1, \ldots, \mathsf{x}_k\}$, and the substitutions are $\sigma = \{\mathsf{x}_1 \mapsto \mathsf{s}_1, \ldots, \mathsf{x}_k \mapsto \mathsf{s}_k\}$ and $\vartheta = \{\mathsf{x}_1 \mapsto \mathsf{u}_1, \ldots, \mathsf{x}_k \mapsto \mathsf{u}_k\}$.

Transitivity: For each $[\mathsf{t}], [\mathsf{r}], [\mathsf{s}] \in \mathcal{T}_E(\mathsf{F}, \mathsf{V})$:

$$\frac{[\mathsf{t}] \longrightarrow [\mathsf{s}] \qquad [\mathsf{s}] \longrightarrow [\mathsf{r}]}{[\mathsf{t}] \longrightarrow [\mathsf{r}]}.$$

The entailment relation $\mathcal{R} \vdash [t] \longrightarrow [r]$ is defined to model the concurrent rewriting of $[t]$ to $[r]$, using the rewrite rules of $\mathcal{R}$. When $[t]$ is used to specify the rules of change in a concurrent system, an entailed sequent $[t] \longrightarrow [r]$ has the intended reading "$[t]$ becomes $[r]$." Concurrent rewriting, as it was shown in [18], actually coincides with deduction in rewriting logic.

The version of rewriting logic described above does not contain sorts for simplicity. What we are interested in this paper, however, is rewriting logic with ordered sorts. The notions defined above are easily transferred to the order-sorted case, provided that the signature satisfies a simple technical property called pre-regularity, which guarantees the existence of the least sort for each term. We briefly recall the main notions of ordered signatures and theories here, slightly adjusting them to our terminology. For details one can see, e.g., [8].

In order-sorted setting, in the role of F we have an alphabet, a triple $(B, \leqslant, S)$, where B is called the set of basic sorts, S is a $B^* \times B$-sorted set of sets of function symbols $\{F_{w,b} \mid w \in B^*, b \in B\}$, B is partially ordered by the ordering $\leqslant$, and the function symbols satisfy the following monotonicity condition:

$$f \in F_{w_1,b_1} \cap F_{w_2,b_2} \text{ and } w_1 \leqslant w_2 \text{ imply } b_1 \leqslant b_2.$$

When $f \in F_{w,b}$, we say that w is the arity of f and b is the result sort of f. When w is the empty word, then f is called a constant. The monotonicity condition excludes overloaded constants.

We assume that the set of variables is also sorted, which means that $V = \{V_b \mid b \in B\}$ is a family of disjoint sets V_b of variables for each $b \in B$. The set of order-sorted terms of sort $b \in B$ over $F = (B, \leqslant, S)$, denoted $\mathcal{T}_b(F, V)$, is defined as the least set satisfying the following properties:

- $V_b \subseteq \mathcal{T}_b(F, V)$.
- Let λ be the empty word of sorts. Then $F_{\lambda,b} \subseteq \mathcal{T}_b(F, V)$.
- $\mathcal{T}_{b'}(F, V) \subseteq \mathcal{T}_b(F, V)$ if $b' \leqslant b$.
- If $f \in F_{w,b}$ where $w = b_1 \cdots b_n \neq \lambda$ and $t_i \in \mathcal{T}_{b_i}(F, V)$ for all $1 \leqslant i \leqslant n$, then $f(t_1, \ldots, t_n) \in \mathcal{T}_b(F, V)$.

The terms defined in this way might have different, even incomparable sorts. It has some unpleasant consequences (e.g., the generated term algebra is not initial, see [8]). However, with the above mentioned property of pre-regularity this problem disappears. The alphabet $(B, \leqslant, S)$ is called pre-regular iff the following property is satisfied: Let $w_0 \in B^*$. Then for any $w_1 \in B^*$ with $w_0 \leqslant w_1$ and $f \in F_{w_1,b_1}$, there is a least sort $b \in B$ such that $w_0 \leqslant w_1$ and $f \in F_{w,b}$ for some $w \in B^*$. Goguen and Meseguer in [8] proved that any term built over a pre-regular alphabet has the least sort. Sides of equalities are assumed to belong to a set of terms of the same sort.

We write $f : w \to b$ if $f \in F_{w,b}$.

Example 2.1 Let $\mathcal{R} = (F, E, L, R)$ be an order-sorted rewrite theory with two basic sorts Nat and Tree, ordered as $\mathsf{Nat} < \mathsf{Tree}$. The signature F contains sorted function symbols $0 : \lambda \to \mathsf{Nat}$, $+ : \mathsf{Nat}\ \mathsf{Nat} \to \mathsf{Nat}$, $\mathsf{suc} : \mathsf{Nat} \to \mathsf{Nat}$, $\mathsf{rev} : \mathsf{Tree} \to \mathsf{Tree}$, and $\bowtie$: $\mathsf{Tree}\ \mathsf{Tree} \to \mathsf{Tree}$. The set of equations E contains the commutativity axiom for $+$, In L there are the labels l_1, l_2, l_3, l_4, and the set R consists of the following four

rules:

$\mathsf{l}_1 : [\mathsf{x}] + [0] \to [\mathsf{x}]$.
$\mathsf{l}_2 : [\mathsf{suc}(\mathsf{x}) + \mathsf{suc}(\mathsf{y})] \to [\mathsf{suc}(\mathsf{suc}(\mathsf{x} + \mathsf{y}))]$.
$\mathsf{l}_3 : [\mathsf{rev}(\mathsf{x})] \to [\mathsf{x}]$.
$\mathsf{l}_4 : [\mathsf{rev}(\mathsf{x} \bowtie \mathsf{y})] \to [\mathsf{rev}(\mathsf{y}) \bowtie \mathsf{rev}(\mathsf{x})]$.

3 The ρLog Calculus

In this section we describe a fragment of ρLog calculus [13] that is relevant for our goal: to express the deduction system of rewriting logic.

The ρLog signature $\mathcal{F}$ consists of unranked function symbols. The symbols f, g, h, a, b, and c are used to denote them. The countably infinite set of variables $\mathcal{V}$ is split into three disjoint subsets: individual variables $\mathcal{V}_{\text{Ind}}$, whose elements are denoted by letters x, y, z; sequence variables $\mathcal{V}_{\text{Seq}}$, denoted by $\overline{x}, \overline{y}, \overline{z}$; and function variables $\mathcal{V}_{\text{Fun}}$, usually written as X, Y, Z. As usual, it is assumed that $\mathcal{F} \cap \mathcal{V} = \varnothing$.

Definition 3.1 The set of *terms over $\mathcal{F}$ and $\mathcal{V}$*, denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$, and the set of *term sequences over $\mathcal{F}$ and $\mathcal{V}$*, denoted by $\mathcal{S}(\mathcal{F}, \mathcal{V})$, are the least sets satisfying the properties:

- $\mathcal{V}_{\text{Ind}} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V})$.
- $\epsilon \in \mathcal{S}(\mathcal{F}, \mathcal{V})$, where ϵ denotes the empty sequence of terms and sequence variables.
- $s_1, \ldots, s_n \in \mathcal{S}(\mathcal{F}, \mathcal{V})$,[1] $n \geqslant 1$, if $s_i \in \mathcal{T}(\mathcal{F}, \mathcal{V}) \cup \mathcal{V}_{\text{Seq}}$ for each $1 \leqslant i \leqslant n$.
- $f(s_1, \ldots, s_n) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, if $s_1, \ldots, s_n \in \mathcal{S}(\mathcal{F}, \mathcal{V})$.
- $X(s_1, \ldots, s_n) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, if $s_1, \ldots, s_n \in \mathcal{S}(\mathcal{F}, \mathcal{V})$.

Note that $\mathcal{T}(\mathcal{F}, \mathcal{V}) \subseteq \mathcal{S}(\mathcal{F}, \mathcal{V})$. In other words, we do not distinguish between a term and a singleton term sequence. Terms of the form $a(\epsilon)$ are abbreviated with a. For readability, we may write sequences within parentheses, usually when there is more than one element in the sequence.

We denote terms by t, r, terms or sequence variables by s, u, and sequences (of terms or sequence variables) by $\tilde{s}, \tilde{u}$.

If $\tilde{s} = (s_1, \ldots, s_n)$ and $\tilde{u} = (u_1, \ldots, u_m)$, $n, m \geqslant 0$, we slightly overload the comma, writing $(\tilde{s}, \tilde{u})$ for the sequence $(s_1, \ldots, s_n, u_1, \ldots, u_m)$. Obviously, when $n = 0$, i.e., when $\tilde{s} = \epsilon$, then $(\tilde{s}, \tilde{u}) = \tilde{u}$. Similarly, for $\tilde{u} = \epsilon$ we have $(\tilde{s}, \tilde{u}) = \tilde{s}$.

The set of variables of a sequence $\tilde{s}$ is denoted by $var(\tilde{s})$. We call $\tilde{s}$ ground if $var(\tilde{s}) = \varnothing$. These notions extend to sets of term sequences, etc.

3.1 Substitutions and Matching Problems

A *substitution* σ is a mapping $\sigma : \mathcal{V} \to \mathcal{S}(\mathcal{F}, \mathcal{V}) \cup \mathcal{F} \cup \mathcal{V}_{\text{Fun}}$ such that the following properties are satisfied:

[1] Note that $s_1, \ldots, s_n \in \mathcal{S}(\mathcal{F}, \mathcal{V})$ means that the sequence $s_1, \ldots, s_n$ of terms and sequence variables belongs to $\mathcal{S}(\mathcal{F}, \mathcal{V})$. It should *not* be read as $s_1 \in \mathcal{S}(\mathcal{F}, \mathcal{V}), \ldots, s_n \in \mathcal{S}(\mathcal{F}, \mathcal{V})$.

- for all $x \in \mathcal{V}_{\text{Ind}}$, $\sigma(x) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$,
- for all $\overline{x} \in \mathcal{V}_{\text{Seq}}$, $\sigma(\overline{x}) \in \mathcal{S}(\mathcal{F}, \mathcal{V})$,
- for all $X \in \mathcal{V}_{\text{Fun}}$, $\sigma(X) \in \mathcal{F} \cup \mathcal{V}_{\text{Fun}}$, and
- all but finitely many variables are mapped to themselves.

Substitutions are denoted by lowercase Greek letters σ, ϑ, and ε, where ε stands for the identity substitution. A substitution σ can apply to a term t or a sequence $\tilde{s}$ and result in the *instances* (under σ): $t\sigma$ of t and $\tilde{s}\sigma$ of $\tilde{s}$. They are defined as $x\sigma = \sigma(x)$, $f(\tilde{s})\sigma = f(\tilde{s}\sigma)$, $X(\tilde{s})\sigma = \sigma(X)(\tilde{s}\sigma)$, $\overline{x}\sigma = \sigma(\overline{x})$ and $(s_1, \ldots, s_n)\sigma = (s_1\sigma, \ldots, s_n\sigma)$. For instance, if $\sigma = \{\overline{x} \mapsto (g(a), \overline{y}), \overline{y} \mapsto \epsilon, z \mapsto a, X \mapsto f\}$, then $(\overline{x}, X(\overline{x}, z), b, \overline{y}, z)\sigma = (g(a), \overline{y}, f(g(a), \overline{y}, a), b, a)$.

The notion of substitution composition is defined in the standard way. (See, e.g., [1].) We use juxtaposition $\sigma\vartheta$ for composition of σ with ϑ

Matching with sequence variables is finitary, see, e.g., [9,10]. For instance, if $t = \{f(\overline{x}, b, \overline{y})$ and $r = f(b, f(a, c), b)\}$, then the complete set of matchers of t to r consists of the following two solutions: $\sigma_1 = \{\overline{x} \mapsto \epsilon, \overline{y} \mapsto (f(a, c), b)\}$ and $\sigma_2 = \{\overline{x} \mapsto (b, f(a, c)), \overline{y} \mapsto \epsilon\}$. If f is orderless (a generalization of commutativity for unranked function symbols), then we have more solutions: $\sigma_3 = \{\overline{x} \mapsto \epsilon, \overline{y} \mapsto (b, f(a, c))\}$, $\sigma_4 = \{\overline{x} \mapsto (f(a, c), b), \overline{y} \mapsto \epsilon\}$, $\sigma_5 = \{\overline{x} \mapsto b, \overline{y} \mapsto f(a, c)\}$, and $\sigma_6 = \{\overline{x} \mapsto f(a, c), \overline{y} \mapsto b\}$. These six substitutions form the complete set of matchers modulo the orderless theory for f.

3.2 Definite Fragment of ρLog

ρLog atoms are triples $(s, \tilde{t}, \tilde{r})$, usually written as a labeled rule $s :: \tilde{t} \Rightarrow \tilde{r}$, where s is called a strategy term, and $\tilde{t}$ and $\tilde{r}$ are term sequences. The intuition is that s denotes a transformation of $\tilde{t}$ into $\tilde{r}$.

In this paper we consider only definite ρLog programs (no negative literals involved) that are sets of nonnegative Horn clauses, constructed from ρLog atoms. The clauses are written as usual, e.g., $s_0 :: \tilde{t}_0 \Rightarrow \tilde{r}_0$ if $s_1 :: \tilde{t}_1 \Rightarrow \tilde{r}_1, \ldots, s_n :: \tilde{t}_n \Rightarrow \tilde{r}_n$, $n \geqslant 0$. Goals are conjunctions of atoms, e.g., $s_1 :: \tilde{t}_1 \Rightarrow \tilde{r}_1, \ldots, s_n :: \tilde{t}_n \Rightarrow \tilde{r}_n$. We often use the term *rule* when we refer to a ρLog clause. We also say that a clause defines the strategy in its head (i.e., the clause above defines the strategy s_0).

The inference system of our fragment of ρLog consists of two rules: one is resolution, and the other one is for the special strategy **id** (which, intuitively, denotes the identity transformation of a sequence to itself). Given a program $\mathcal{P}$ and a set of equations E, these rules are defined as follows (they should be read bottom up: To prove the query in the lower part, prove the query in the upper part.)

Resolution:

$$\frac{(s_1 :: \tilde{t}_1 \Rightarrow \tilde{r}_1, \ldots, s_n :: \tilde{t}_n \Rightarrow \tilde{r}_n, \mathbf{id} :: \tilde{r}_0 \Rightarrow \tilde{u},\ \mathcal{Q})\sigma}{s :: \tilde{t} \Rightarrow \tilde{u},\ \mathcal{Q},}$$

where $s_0 :: \tilde{t}_0 \Rightarrow \tilde{r}_0$ if $s_1 :: \tilde{t}_1 \Rightarrow \tilde{r}_1, \ldots, s_n :: \tilde{t}_n \Rightarrow \tilde{r}_n$ is a clause from $\mathcal{P}$, $s \neq \mathbf{id}$, and $s_0\sigma =_E s$, $\tilde{t}_0\sigma =_E \tilde{t}$.

Identity:

$$\frac{\mathcal{Q}\sigma}{\mathbf{id} :: \tilde{t} \Rightarrow \tilde{u},\ \mathcal{Q},}$$

where $\tilde{u}\sigma =_E \tilde{t}$.

Here we look at these inference rules as logical deduction rules.

There are some predefined ρLog strategies with fixed meaning, which are useful in the next section for the mapping from Rewriting Logic to ρLog:

- If s is a strategy term, then the strategy $\mathbf{map}(s) :: (t_1, \ldots, t_n) \Rightarrow (r_1, \ldots, r_n)$ succeeds iff each strategy $s :: t_i \Rightarrow r_i$, $1 \leqslant i \leqslant n$, succeeds.
- If $s_1, \ldots, s_n$ are strategy terms, then the strategy $\mathbf{choice}(s_1, \ldots, s_n) :: \tilde{t} \Rightarrow \tilde{r}$ succeeds iff at least one of the strategies $s_i :: \tilde{t} \Rightarrow \tilde{r}$, $1 \leqslant i \leqslant n$ succeeds.

Note that **map**, when realized as an extra inference rule for ρLog, can be used to perform transformations in parallel. It can also be specified within ρLog as a clause, doing transformations sequentially. Such an "internalization" of **map** is, in fact, pretty simple:

$\mathbf{map}(z) :: \epsilon \Rightarrow \epsilon.$
$\mathbf{map}(z) :: (x, \overline{x}) \Rightarrow (y, \overline{y})$ if $z :: x \Rightarrow y$, $\mathbf{map}(z) :: \overline{x} \Rightarrow \overline{y}$.

Similarly, it is also rather straightforward to specify the **choice** strategy as ρLog clauses:

$\mathbf{choice}(z, \overline{z}) :: \overline{x} \Rightarrow \overline{y}$ if $z :: \overline{x} \Rightarrow \overline{y}$.
$\mathbf{choice}(z, \overline{z}) :: \overline{x} \Rightarrow \overline{y}$ if $\mathbf{choice}(\overline{z}) :: \overline{x} \Rightarrow \overline{y}$.

The semantics of ρLog can be defined in the same way as it is done in logic programming, see, e.g., [11].

4 From Rewriting Logic to ρLog: Mapping Entailment Systems

The goal of this section is to illustrate that, via an appropriate mapping, deduction in rewriting logic can be modeled by deduction in ρLog. In other words, our goal is to define a mapping between what is called *entailment systems* [17] of rewriting logic and ρLog. By an entailment system of a logic one understands a triple consisting of the signature, set of sentences, and the entailment relation $\vdash$ that satisfies certain properties (reflexivity, monotonicity, transitivity, and $\vdash$-translation). The inference systems of both rewriting logic and the definite fragment of ρLog we consider here provide the entailment relation that satisfies those properties.

Hence, the goal is to define an *entailment system mapping* Φ from rewriting logic to ρLog such that the conservativity theorem holds. This theorem, formulated at the end of this section, states that a sequent is provable in rewriting logic with respect to a rewrite theory iff the image of the sequent under Φ is provable in ρLog with respect to a program obtained from the rewrite theory by Φ.

Hence, we start defining Φ for a rewrite theory $\mathcal{R} = (\mathsf{F}, \mathsf{E}, \mathsf{L}, \mathsf{R})$. We assume that $\mathcal{F}$ is split into five disjoint countable sets of symbols $\mathcal{F}_{\mathsf{F}}$, $\mathcal{F}_{\mathsf{V}}$, $\mathcal{F}_{\mathsf{S}}$, $\mathcal{F}_{\mathsf{L}}$, and $\mathcal{F}_\rho$, such that Φ for rewriting logic function symbols, variables, basic sorts, and labels is defined as follows:

- For each $\mathsf{f} \in \mathsf{F}$ we have a symbol $f \in \mathcal{F}_{\mathsf{F}}$, and

  $$\Phi(\mathsf{f}) = f.$$

- For each $\mathsf{x} \in \mathsf{V}$ there is a symbol $c_{\mathsf{x}} \in \mathcal{F}_{\mathsf{V}}$, and

  $$\Phi(\mathsf{x}) = \mathit{rlv}(c_{\mathsf{x}}),$$

  where rlv is a function symbol from $\mathcal{F}_\rho$. Hence, rewriting logic variables are mapped to ρLog ground terms tagged by the function symbol rlv.
- For each rewriting logic basic sort a there is a symbol $a \in \mathcal{F}_{\mathsf{S}}$, and

  $$\Phi(\mathsf{a}) = \mathit{rls}(a),$$

  where rls is a function symbol from $\mathcal{F}_\rho$. Hence, rewriting logic sort symbols are mapped to ρLog ground terms tagged by the function symbol rls.
- For each $\mathsf{l} \in \mathsf{L}$ there is a symbol $l \in \mathcal{F}_{\mathsf{L}}$, and

  $$\Phi(\mathsf{l}) = l.$$

The symbols from $\mathcal{F}_\rho$ will be also used in ρLog programs below.

Since ρLog is unsorted, we need to encode sort definitions and the subsort relation explicitly as clauses. This is done in the following way:

- For each pair of basic sorts a, b, related by the subsort relation $\leqslant$, Φ gives a clause

  $$\mathit{subsort\_basic} :: \mathit{rls}(a) \Rightarrow \mathit{rls}(b).$$

  Then the subsort relation is defined as follows:

  $$\begin{array}{l}\mathit{subsort} :: \mathit{rls}(x) \Rightarrow \mathit{rls}(x).\\ \mathit{subsort} :: \mathit{rls}(x) \Rightarrow \mathit{rls}(y) \text{ if}\\ \quad \mathit{subsort\_basic} :: \mathit{rls}(x) \Rightarrow \mathit{rls}(z),\ \mathit{subsort} :: \mathit{rls}(z) \Rightarrow \mathit{rls}(y).\end{array}$$

- For each function symbol $\mathsf{f} : \mathsf{a}_1 \cdots \mathsf{a}_n \to \mathsf{b}$, Φ gives a clause

  $$\begin{array}{l}\mathit{sort\_def} :: f(x_1, \ldots, x_n) \Rightarrow \mathit{rls}(b) \text{ if}\\ \quad \mathit{sort} :: x_1 \Rightarrow \mathit{rls}(a_1), \ldots, \mathit{sort} :: x_n \Rightarrow \mathit{rls}(a_n),\end{array}$$

  where the strategy sort is defined as

  $$\begin{array}{l}\mathit{sort} :: x \Rightarrow \mathit{rls}(y) \text{ if}\\ \quad \mathit{sort\_def} :: x \Rightarrow \mathit{rls}(z),\ \mathit{subsort} :: \mathit{rls}(z) \Rightarrow \mathit{rls}(y).\end{array}$$

- We define a strategy *is_sorted* for terms:

  $$is\_sorted :: x \Rightarrow true \text{ if } sort :: x \Rightarrow rls(y),$$

  where *true* is a function symbol. This clause says that a term is sorted if it has a sort.

Further, Φ is extended in a straightforward way to a mapping from $\mathcal{T}(\mathsf{F},\mathsf{V})$-terms, equations, and substitutions into $\mathcal{T}(\mathcal{F},\mathcal{V})$-terms, equations, and individual variable substitutions, respectively We define $\Phi([\mathsf{t}]) = \Phi(\mathsf{t})$.

For the set of rewrite rules R, the mapping Φ is defined as follows:

$$\begin{aligned}\Phi(\mathsf{R}) := \{&\Phi(rule) \mid rule \in \mathsf{R}\} \cup \\ &\{rwl\_choice :: \overline{x} \Rightarrow \overline{y} \text{ if } \mathbf{choice}(\Phi(\mathsf{l}_1),\ldots,\Phi(\mathsf{l}_m)) :: \overline{x} \Rightarrow \overline{y}\}.\end{aligned}$$

where $\mathsf{l}_1,\ldots,\mathsf{l}_m$ are all the labels of the rules in R, and Φ for the rules is defined below.

Before saying what the image of a rule $\mathsf{l} : [\mathsf{t}_0] \to [\mathsf{r}_0]$ if $[\mathsf{t}_1] \to [\mathsf{r}_1] \wedge \cdots \wedge [\mathsf{t}_n] \to [\mathsf{r}_n]$ under Φ is, we assume that $\Phi(\mathsf{l}) = l$, $\Phi([\mathsf{t}_i]) = t_i$, and $\Phi([\mathsf{r}_i]) = r_i$ for $0 \leqslant i \leqslant n$. Besides, let $\mathsf{x}_1,\ldots,\mathsf{x}_m$ be all rewriting logic variables in r_0 and for each $1 \leqslant i \leqslant m$, let c_{x_i} be the corresponding symbol from $\mathcal{F}_{\mathsf{V}}$. Then, by the definition of Φ, r_0 contains $rlv(c_{\mathsf{x}_i})$ in place of x_i. Let $c_1,\ldots,c_m$ be symbols from $\mathcal{F}_\rho$ that are fresh in the context, and denote by r'_0 the term obtained from r_0 by replacing each $rlv(c_{\mathsf{x}_i})$, $1 \leqslant i \leqslant m$, by c_i. Then:

$$\begin{aligned}&\Phi(\mathsf{l} : [\mathsf{t}_0] \to [\mathsf{r}_0] \text{ if } [\mathsf{t}_1] \to [\mathsf{r}_1] \wedge \cdots \wedge [\mathsf{t}_n] \to [\mathsf{r}_n]) := \\ &\quad l :: y_0 \Rightarrow (c_1 \hookrightarrow s_1,\ldots,c_m \hookrightarrow s_m,\ r'_0) \text{ if} \\ &\qquad match :: t_0 \ll y_0 \Rightarrow \overline{z}^{\sigma}_0, \\ &\qquad apply\_subst(\overline{z}^{\sigma}_0) :: t_1 \Rightarrow t'_1,\ apply\_subst(\overline{z}^{\sigma}_0) :: r_1 \Rightarrow r'_1, \\ &\qquad rwl\_inf :: t'_1 \Rightarrow y_1, \\ &\qquad match :: r'_1 \ll y_1 \Rightarrow \overline{z}^{\sigma}_1, \\ &\qquad apply\_subst(\overline{z}^{\sigma}_0,\overline{z}^{\sigma}_1) :: t_2 \Rightarrow t'_2,\ apply\_subst(\overline{z}^{\sigma}_0,\overline{z}^{\sigma}_1) :: r_2 \Rightarrow r'_2, \\ &\qquad \ldots, \\ &\qquad rwl\_inf :: t'_n \Rightarrow y_n, \\ &\qquad match :: r'_n \ll y_n \Rightarrow \overline{z}^{\sigma}_n, \\ &\qquad apply\_subst(\overline{z}^{\sigma}_0,\ldots,\overline{z}^{\sigma}_n) :: rlv(c_{\mathsf{x}_1}) \Rightarrow s_1, \\ &\qquad \ldots, \\ &\qquad apply\_subst(\overline{z}^{\sigma}_0,\ldots,\overline{z}^{\sigma}_n) :: rlv(c_{\mathsf{x}_m}) \Rightarrow s_m,\end{aligned}$$

where $\hookrightarrow$ is a function symbol from $\mathcal{F}_\rho$, used to model replacement pairs. Obviously, if r_0 is a ground term, then the sequence $c_1 \hookrightarrow rlv(c_{\mathsf{x}_1}),\ldots,c_n \hookrightarrow rlv(c_{\mathsf{x}_n})$ is empty and $r'_0 = r_0$. *rwl_inf* is the strategy that corresponds to the inference in rewriting logic and is defined below.

The translation of r_0 into the sequence $(c_1 \hookrightarrow rlv(c_{\mathsf{x}_1}),\ldots,c_n \hookrightarrow rlv(c_{\mathsf{x}_n}),r'_0)$ is a trick that will play its role with the Replacement inference rule, where the instances of variables in the right hand side of a rule are reduced. The intuition behind this

sequence is similar to the *let* construct in programming, and below (in the definition of the Replacement Rule) we will define a strategy that has a similar effect.

So far we have not taken into account the equational part of rewrite theories, i.e., the set E. Its translation can be dealt with in various ways. For instance, we can assume that it is incorporated into the matching mechanism of ρLog as a matching algorithm modulo E. This approach is feasible when matching modulo E is decidable and finitary. Or, if E induces a convergent rewrite system, then its image under Φ can be a set of ρLog rules, obtained in the similar way as $\Phi(\mathsf{R})$, but grouped under the name of one strategy (e.g., *reduce_by_equalities*) and in the inference step (i.e., in the strategy *rwl_inf* below) the terms before and after reduction by inference rules are brought to the normal form with respect to the strategy *reduce_by_equalities*. One can also consider a mixed variant (which is implemented in Maude), where matching modulo some equational theories are built-in, and the remaining set of equalities is convergent.

Basically, with this we have Φ defined for rewrite theories. In whatever way one deals with the rewriting logic equalities, we always need syntactic matching for the expressions translated in ρLog. Since the rewriting logic variables are mapped to ρLog ground terms, matching should be implemented explicitly:

$$
\begin{array}{l}
\mathit{match} :: (x \ll y) \Rightarrow \overline{x} \ \mathsf{if} \\
\quad \mathit{change\_tag}(\mathit{rlv} \mapsto \mathit{temp\_tag}) :: y \Rightarrow z, \\
\quad \mathbf{nf}(\mathbf{first\_one}(\mathit{finish}, \mathit{solved\_equation}, \mathit{variable\_elim}, \mathit{decomposition})) :: \\
\qquad (\mathit{mp}(x \ll z), \mathit{subst}) \Rightarrow \mathit{subst}(\overline{z}), \\
\quad \mathit{change\_tag}(\mathit{temp\_tag} \mapsto \mathit{rlv}) :: \mathit{subst}(\overline{z}) \Rightarrow \mathit{subst}(\overline{x}).
\end{array}
$$

The code above corresponds to the variant of matching algorithm when variables in the right hand side of the matching problem are replaced by temporary constants, then the matching rules are fired (first applicable, as long as possible), and in the computed matcher the introduced constants are mapped back to the original variables they replaced. The constructor function symbol for the matching problem is *mp*, and for the substitution *subst*.

The strategy *finish* below says that if the matching problem is empty, then the computed substitution should be returned. This corresponds to the success of matching. The other three strategies implement the standard matching rules.

$$
\begin{array}{l}
\mathit{finish} :: (\mathit{mp}, x) \Rightarrow x. \\
\mathit{solved\_equation} :: (\mathit{mp}(x \ll x, \overline{x}), \mathit{subst}(\overline{y})) \Rightarrow (\mathit{mp}(\overline{x}), \mathit{subst}(\overline{y})). \\
\mathit{variable\_elim} :: (\mathit{mp}(\mathit{rlv}(x) \ll y, \overline{x}), \mathit{subst}(\overline{y})) \Rightarrow \\
\quad (\mathit{mp}(\overline{x}_1), \mathit{subst}(\mathit{rlv}(x) \mapsto y, \overline{y}_1)) \ \mathsf{if} \\
\qquad \mathit{apply\_subst}(\mathit{rlv}(x) \mapsto y) :: \mathit{mp}(\overline{x}) \Rightarrow \mathit{mp}(\overline{x}_1), \\
\qquad \mathit{apply\_subst}(\mathit{rlv}(x) \mapsto y) :: \mathit{subst}(\overline{y}) \Rightarrow \mathit{subst}(\overline{y}_1). \\
\mathit{decomposition} :: (\mathit{mp}(F(\overline{x}_1) \ll F(\overline{x}_2), \overline{y}), x) \Rightarrow (\mathit{mp}(\overline{z}, \overline{y}), x) \ \mathsf{if} \\
\qquad \mathit{zip} :: (F(\overline{x}_1), F(\overline{x}_2)) \Rightarrow \overline{z}.
\end{array}
$$

The remaining strategies are auxiliary ones used in the rules or in the algorithm

control above:

$$
\begin{array}{l}
\mathit{zip} := \mathbf{first\_one}(\mathit{zip\_nonempty}, \mathit{zip\_empty}).\\
\mathit{zip\_nonempty} :: (F(x_1, \overline{x}_1), F(x_2, \overline{x}_2)) \Rightarrow (x_1 \ll x_2, \overline{y}) \text{ if}\\
\quad \mathit{zip} :: (F(\overline{x}_1), F(\overline{x}_2)) \Rightarrow \overline{y}.\\
\mathit{zip\_empty} :: (F, F) \Rightarrow \epsilon.
\end{array}
$$

$$
\begin{array}{l}
\mathit{apply\_subst}(\overline{x}) := \mathbf{first\_one}(\mathit{apply\_subst\_basic}(\overline{x}), \mathit{apply\_subst\_rec}(\overline{x})).\\
\mathit{apply\_subst\_basic}(\overline{x}, \mathit{rlv}(x) \mapsto y, \overline{y}) :: \mathit{rlv}(x) \Rightarrow y.\\
\mathit{apply\_subst\_rec}(\overline{x}) :: F(\overline{y}) \Rightarrow F(\overline{z}) \text{ if } \mathbf{map}(\mathit{apply\_subst}(\overline{x})) :: \overline{y} \Rightarrow \overline{z}.
\end{array}
$$

$$
\begin{array}{l}
\mathit{change\_tag}(F_1 \mapsto F_2) :=\\
\quad \mathbf{first\_one}(\mathit{change\_tag\_basic}(F_1 \mapsto F_2), \mathit{change\_tag\_rec}(F_1 \mapsto F_2)).\\
\mathit{change\_tag\_basic}((F_1 \mapsto F_2)) :: F_1(x) \Rightarrow F_2(x).\\
\mathit{change\_tag\_rec}(F_1 \mapsto F_2) :: F(\overline{y}) \Rightarrow F(\overline{z}) \text{ if}\\
\quad \mathbf{map}(\mathit{change\_tag}(F_1 \mapsto F_2)) :: \overline{y} \Rightarrow \overline{z}.
\end{array}
$$

One could easily extend this algorithm to work, for instance, with commutative matching symbols. We would need to add only one rule, called commutative decomposition:

$$
\begin{array}{l}
\mathit{commutative\_decomposition} :: (\mathit{mp}(F(x_1, y_1) \ll F(x_2, y_2), \overline{y}), x) \Rightarrow\\
\quad (\mathit{mp}(x_1 \ll y_2, x_2 \ll y_1, \overline{y}), x) \text{ if}\\
\quad\quad \mathit{is\_commutative} :: F \Rightarrow \mathit{true}.
\end{array}
$$

(The strategy *is_commutative* is assumed to be defined for each commutative function symbol, and it is a part of the translation of commutativity equations from E.) To make this rule work in the matching algorithm, we will need to replace the occurrence of *decomposition* in *match* above by the choice between *commutative_decomposition* and *decomposition*.

In a similar way, one could easily incorporate into ρLog equational matching algorithms in some other common theories, such as associativity, associativity-commutativity or their combinations with the unit element. (These are theories for which Maude also provides built-in matching algorithms.)

The final step in the construction of the mapping Φ is to define it for the inference rules of rewriting logic. They are translated into a set of ρLog as follows:

Reflexivity Rule in ρLog:

$$\mathit{rwl\_refl} :: x \Rightarrow x \text{ if } \mathit{is\_sorted} :: x \Rightarrow \mathit{true}.$$

Congruence Rule in ρLog:

$$\mathit{rwl\_cong} :: X(\overline{x}) \Rightarrow X(\overline{y}) \text{ if } \mathbf{map}(\mathit{rwl\_inf}) :: \overline{x} \Rightarrow \overline{y}.$$

Replacement Rule in ρLog:

$$\mathit{rwl\_repl} :: x \Rightarrow y \text{ if}$$

$$rwl\_choice :: x \Rightarrow (\overline{y}, z),$$
$$\mathbf{map}(reduce_{\hookrightarrow}) :: \overline{y} \Rightarrow \overline{z},$$
$$let(\overline{z}) :: z \Rightarrow y.$$

The strategy $reduce_{\hookrightarrow}$ is defined as

$$reduce_{\hookrightarrow} :: x \hookrightarrow y \Rightarrow x \hookrightarrow z \text{ if } rwl\_inf :: y \Rightarrow z.$$

The strategy *let* is defined as

$$let(\overline{z}) :: x \Rightarrow y \text{ if } \mathbf{first\_one}(replace(\overline{z}), \mathbf{id}) :: x \Rightarrow y.$$
$$replace(\overline{z}_1, x \hookrightarrow y, \overline{z}_2) :: x \Rightarrow y.$$
$$replace(\overline{z}) :: X(\overline{x}) \Rightarrow X(\overline{y}) \text{ if } \mathbf{map}(let(\overline{z})) :: \overline{x} \Rightarrow \overline{y}.$$

Transitivity Rule in ρLog:

$$rwl\_trans :: x \Rightarrow y \text{ if}$$
$$rwl\_inf :: x \Rightarrow z, rwl\_inf :: z \Rightarrow y.$$

The main strategy is *rwl_inf*, which encodes the fact that an inference step in rewriting logic is made by the above mentioned inference rules (and guaranteeing well-sortedness of the involved terms):

Inference:

$$rwl\_inf :: x \Rightarrow y \text{ if}$$
$$is\_sorted :: x \Rightarrow true,$$
$$\mathbf{choice}(rwl\_refl, rwl\_cong, rwl\_repl, rwl\_trans) :: x \Rightarrow y,$$
$$is\_sorted :: y \Rightarrow true.$$

Hence, we constructed the translation mapping Φ from a rewriting logic theory $\mathcal{R}$ to the ρLog set of definite clauses $\Phi(\mathcal{R})$, and translated the inference rules of rewriting logic into ρLog definite clauses as well.

While the clauses for *rwl_refl*, *rwl_cong*, *rwl_trans* directly imitate the behavior of the corresponding inference rules of rewriting logic (and vice versa), the *rwl_repl* rule needs more explanation. For this purpose, we read the Replacement inference on page 3 bottom-up and see how proving the ρLog atom $rwl\_repl :: t_0\Phi(\sigma) \Rightarrow r_0\Phi(\vartheta)$ corresponds exactly to proving the rewriting logic sequent $[\mathsf{t}_0\sigma] \longrightarrow [\mathsf{r}_0\vartheta]$ by the Replacement inference, where σ and ϑ are substitutions from that rule.

Proving $rwl\_repl :: t_0\sigma \Rightarrow r_0\vartheta$ requires proving atoms in the body of the clause that defines *rwl_repl*. The first of them (call it **A1**), with the strategy *rwl_choice*, corresponds to finding a rule for the rewrite theory: It should be the one that has t_0 (or a term that equals t_0 modulo the set of equalities $\Phi(\mathsf{E})$) in its left hand side, and has the construction that corresponds to r_0 (or a term that is $\Phi(\mathsf{E})$-equal to r_0) in its right hand side. The construction consists of (i) a sequence of correspondences between fresh function symbols and $\Phi(\sigma)$-*instances* of variables of r_0 (this sequence is consumed by $\overline{y}$ in the clause and consists of terms of the form $c_i^x \hookrightarrow s_i$), and (ii) the term r_0' which is obtained from r_0 by replacing variables by those fresh atoms.

Note that proving **A1**, if the body of its clause is not empty, requires proving the $\Phi(\sigma)$-instance of that body, that is nothing else than the task of proving the sequents obtained from the σ-instance of the condition of the selected rule with the label l, i.e., those $[\mathsf{t}_i\sigma] \longrightarrow [\mathsf{r}_i\sigma]$ sequents in the upper part of the Replacement inference.

Next atom maps *rwl_inf* on the sequence $\overline{y}$, which means that $rwl\_inf :: c_i^x \hookrightarrow s_i \Rightarrow rhs_i$ should be proved for all elements $c_i^x \hookrightarrow s_i$ in the sequence. However, due to the fact that $\hookrightarrow$ does not appear in the left hand side of rules and c_i^x's were fresh, the rhs_i's should have a form $c_i^x \hookrightarrow u_i$ for some u_i's. But this corresponds to the proof of the sequent $[\mathsf{s}_i] \longrightarrow [\mathsf{u}_i]$ in the Replacement rule.

Finally, the strategy *let* puts each u_i in place of c_i^x in r_0', obtaining $r_0\Phi(\vartheta)$. It corresponds to the application of the substitution ϑ to r_0 in the Replacement rule.

The following result, called the conservativity theorem, connects deductions in rewriting logic to those in ρLog:

Theorem 4.1 *Given a rewrite theory $\mathcal{R}$, a sequent $[\mathsf{t}] \longrightarrow [\mathsf{r}]$ is provable in rewriting logic from $\mathcal{R}$ iff the atom $rwl\_inf :: \Phi(\mathsf{t}) \Rightarrow \Phi(\mathsf{r})$ is provable in ρLog from $\Phi(\mathcal{R})$.*

Proof. (Sketch) First, assume that the equational part of $\mathcal{R}$ is empty, i.e., we have the syntactic equality. We need to show that terms, sorts, subsort relation, equalities, rules, and inferences of rewriting logic are adequately represented in ρLog, but it follows directly from the construction of Φ. For instance, it can be immediately seen that b is a sort of a term t of rewriting logic iff $sort :: \Phi(\mathsf{t}) \Rightarrow \Phi(\mathsf{b})$ is proved in ρLog. The specification of matching in ρLog directly follows the rules of the algorithm. Based on the adequacy of sortedness and matching, we can see that adequacy is straightforward for the reflexivity, congruence, and transitivity inference rules. For the replacement rule, the proof is based on the reasoning above for *rwl_repl*.

As for non-empty equational theories, the result holds when equational matching can be effectively represented in ρLog. Essentially, it means that a terminating finitary algorithm should be available. In this case, we can reason similarly to the case when matching is syntactic. □

Example 4.2 At the end of this section, we see how the rewriting logic theory from Example 2.1 is translated into ρLog clauses. (The general part such as commutative matching and inference rules are the same as above.)

$$
\begin{aligned}
&l_1 :: z \Rightarrow (a \hookrightarrow s, a) \text{ if}\\
&\quad match :: rlv(a_{\mathsf{x}}) + 0 \ll z \Rightarrow \overline{z}^{\sigma},\\
&\quad apply\_subst(\overline{z}^{\sigma}) :: rlv(a_{\mathsf{x}}) \Rightarrow s.\\
&l_2 :: z \Rightarrow (a_1 \hookrightarrow s_1, a_2 \hookrightarrow s_2, suc(suc(a_1 + a_2))) \text{ if}\\
&\quad match :: suc(rlv(a_x)) + suc(rlv(a_y)) \ll z \Rightarrow \overline{z}^{\sigma},\\
&\quad apply\_subst(\overline{z}^{\sigma}) :: rlv(a_x) \Rightarrow s_1,\\
&\quad apply\_subst(\overline{z}^{\sigma}) :: rlv(a_y) \Rightarrow s_2.\\
&l_3 :: z \Rightarrow (a \hookrightarrow s, a) \text{ if}\\
&\quad match :: rev(rlv(a_{\mathsf{x}})) \ll z \Rightarrow \overline{z}^{\sigma},
\end{aligned}
$$

$apply\_subst(\overline{z}^{\sigma}) :: rlv(a_{\times}) \Rightarrow s.$

$l_4 :: z \Rightarrow (a_1 \hookrightarrow s_1, a_2 \hookrightarrow s_2, rev(a_2) \bowtie rev(a_1))$ if
$match :: rev(rlv(a_x) \bowtie rlv(a_y)) \ll z \Rightarrow \overline{z}^{\sigma},$
$apply\_subst(\overline{z}^{\sigma}) :: rlv(a_x) \Rightarrow s_1,$
$apply\_subst(\overline{z}^{\sigma}) :: rlv(a_y) \Rightarrow s_2.$

$subsort\_basic :: rls(a_N) \Rightarrow rls(a_T).$
$sort\_def :: x_1 + x_2 \Rightarrow rls(a_N)$ if
$sort :: x_1 \Rightarrow rls(a_N),$
$sort :: x_2 \Rightarrow rls(a_N).$
$sort\_def :: x_1 \bowtie x_2 \Rightarrow rls(a_T)$ if
$sort :: x_1 \Rightarrow rls(a_T),$
$sort :: x_2 \Rightarrow rls(a_T).$
$sort\_def :: suc(x) \Rightarrow rls(a_N)$ if
$sort :: x \Rightarrow rls(a_N).$
$sort\_def :: rev(x) \Rightarrow rls(a_T)$ if
$sort :: x \Rightarrow a_T.$

5 Conclusion

We showed how rewriting logic (RWL) and ρLog calculus can be related, defining a fragment of ρLog, into which rewriting logic can be encoded by a provability preserving mapping. The mapping, denoted by Φ, actually, relates entailment systems of these two formalisms. Given a theory of rewriting logic, consisting of an alphabet, equations, labels and rewrite rule, Φ maps

- each RWL constant to a constant in the language of ρLog,
- each RWL variable to a ground ρLog term, whose head indicates that it is an encoding of an RWL variable and the argument is a constant corresponding to the variable,
- each RWL basic sort to a ground ρLog term, whose head indicates that it is an encoding of an RWL sort and the argument is a constant corresponding to the sort,
- each RWL rule label into a ρLog constant,
- RWL sort definitions, subsort relation, rewrite rules, inference rules, inference control, and the matching mechanism into ρLog clauses.
- RWL sequents are mapped into ρLog atoms.

RWL equations are either considered to be represented in the implementation of equational matching of ρLog, or they are translated as rules if they induce a convergent rewrite system. Those rules then serve for normalization of terms before and after reduction. A mixed approach is also possible.

The range of Φ is the definite (negation-free) fragment of ρLog, but the rich strategy language of this formalism helps to imitate some kind of behavior which is

usually modeled with the help of negation-as-failure (or the cut) in logic programming. An example of such a strategy is **first_one**, which stops evaluation after one answer of the first applicable strategy is computed.

The important property of the mapping Φ is that it is conservative: provability of a rewriting logic sequent from a rewrite theory is equivalent to the provability of the Φ-image of the sequent from an Φ-image of the theory in ρLog. It shows the expressive power of ρLog: This formalism, like rewriting logic, can be used as a logical and semantic framework.

Acknowledgments

This research has been partially supported by the Brazilian National Council for Scientific and Technological Development CNPq under grant CsF/BJT 401319/2014-8, by the Rustaveli National Science Foundation under the grants FR/508/4-120/14 and YS15 2.1.2 70, and by the Austrian Science Fund (FWF) under the project P 28789-N32.

References

[1] Franz Baader and Wayne Snyder. Unification theory. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier BV., 2001.

[2] Walid Belkhir, Alain Giorgetti, and Michel Lenczner. A symbolic transformation language and its application to a multiscale method. *J. Symb. Comput.*, 65:49–78, 2014.

[3] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[4] Jorge Coelho, Besik Dundua, Mário Florido, and Temur Kutsia. A rule-based approach to XML processing and Web reasoning. In Pascal Hitzler and Thomas Lukasiewicz, editors, *RR 2010*, volume 6333 of *LNCS*, pages 164–172. Springer, 2010.

[5] Besik Dundua. *Programming with Sequence and Context Variables: Foundations and Applications*. PhD thesis, Department of Computer Science, University of Porto, 2014.

[6] Besik Dundua, Temur Kutsia, and Mircea Marin. Strategies in PρLog. In Maribel Fernández, editor, *9th Int. Workshop on Reduction Strategies in Rewriting and Programming, WRS 2009*, volume 15 of *EPTCS*, pages 32–43, 2009.

[7] Besik Dundua, Temur Kutsia, and Klaus Reisenberger-Hagmayer. An overview of PρLog. In Yuliya Lierler and Walid Taha, editors, *Practical Aspects of Declarative Languages - 19th International Symposium, PADL 2017, Paris, France, January 16-17, 2017, Proceedings*, volume 10137 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2017.

[8] Joseph A. Goguen and José Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theor. Comput. Sci.*, 105(2):217–273, 1992.

[9] Temur Kutsia. Solving and Proving in Equational Theories with Sequence Variables and Flexible Arity Symbols. RISC Report Series 02-09, Research Institute for Symbolic Computation (RISC), University of Linz, Schloss Hagenberg, 4232 Hagenberg, Austria, May 2002. PhD Thesis.

[10] Temur Kutsia and Mircea Marin. Matching with regular constraints. In Geoff Sutcliffe and Andrei Voronkov, editors, *LPAR*, volume 3835 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2005.

[11] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[12] Mircea Marin and Temur Kutsia. On the implementation of a rule-based programming system and some of its applications. In Boris Konev and Renate Schmidt, editors, *Proceedings of the 4th International Workshop on the Implementation of Logics (WIL'03)*, pages 55–68, Almaty, Kazakhstan, 2003.

[13] Mircea Marin and Temur Kutsia. Foundations of the rule-based system ρLog. *Journal of Applied Non-Classical Logics*, 16(1-2):151–168, 2006.

[14] Mircea Marin and Florina Piroi. Deduction and Presentation in ρLog. *ENTCS*, 93:161–182, 2004.

[15] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In Dov M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 9, pages 1–87. Kluwer Academic Publishers, 2002.

[16] Narciso Marti-Oliet, Jose Meseguer, and Alberto Verdejo. A Rewriting Semantics for Maude Strategies. *ENTCS*, 238(3):1–18, 2009.

[17] José Meseguer. General logics. *Studies in Logic and the Foundations of Mathematics*, 129:275–329, 1989.

[18] José Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

[19] Jose Meseguer. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming*, 81(7):721 – 781, 2012.

[20] Phong Nguyen. *Meta-mining: a meta-learning framework to support the recommendation, planning and optimization of data mining workflows*. PhD thesis, Department of Computer Science, University of Geneva, 2015.