Relating Meseguer's Rewriting Logic with the Constructor-Based Rewriting Logic

Miguel Palomino Tarjuelo

Departamento de Sistemas Informáticos y Programación
Facultad de Matemáticas, Universidad Complutense de Madrid

A Master's Thesis Directed by
Dr. Narciso Martí Oliet

May 2001

Contents

§1. Introduction

The aim of this research paper is to study in close detail, and to clarify in some extent, the relationships between two well known approaches to rewriting as logical deduction, namely, José Meseguer's rewriting logic [21], and the constructor-based rewriting logic developed by Mario Rodríguez Artalejo's research group in Madrid [14].

The first of these was proposed as a logical framework wherein to represent other logics, and also as a semantic framework for the specification of languages and systems. The experience accumulated throughout the last years has come to support that original intention, having been shown that rewriting logic is a very flexible framework in which many other logics, including first-order logic, intuitionistic logic, linear logic, Horn logic with equality, as well as any other logic with a sequent calculus, can be represented. An important characteristic of these representations that should be stressed is that they are usually quite simple and natural, so that their mathematical properties are often straightforward to derive.

On the other hand, the goal of the constructor-based rewriting logic is to serve as a logical basis for declarative programming languages involving lazy evaluation, offering support, in addition, to nondeterministic functions.

A suitable framework in which to carry out our study is the theory of *general logics* developed by Meseguer. There, a logic is described in a very abstract manner and two separated components are distinguished in it: an *entailment system* and an *institution*, corresponding with the syntantic and the semantic parts of the logic, respectively. Ideally, we would like to find entailment systems and institutions associated to the logics under our consideration so as to be able to define suitable mappings between them; unfortunately, this will prove to be not possible in all cases, and in those occasions we will be forced to leave this formal framework.

Our interest throughout this work will be mainly focused on the simplest versions of both logics, corresponding to an untyped situation. Nevertheless we will also introduce some of their possible extensions in order to point out how our methods could be adapted to deal with more general cases. (Actually, rewriting logic is parameterized by an underlying equational logic, which can vary from the very simple unsorted and unconditional one with which we will be mostly concerned, to the very expressive membership equational logic.)

The final part of this paper tries to contribute with some practical work to the theoretical developments of previous sections. The idea is to define abstract datatypes for the finitely presentable theories in both Meseguer's and Rodríguez-Artalejo's logics, as well as a function mapping theories in this last logic to theories in rewriting logic simulating their behaviour. This function can be defined equationally, and so can be specified in the Maude language associated to rewriting logic, which will allow us to prototype and execute any constructor-based rewriting logic theory.

We have tried this work to be as self-contained as possible, reviewing all notions that are needed, specially the more exotic ones. We assume, however, a basic knowledge of category theory as can be found in [4, 17]; whenever a more advanced topic is used it is first discussed in the text.

In a work like this, notation itself sometimes becomes a problem. We have stuck to standard usage, but tried not to be excessively formalist in order to lighten the notation as much as possible. For example, the turnstile $\vdash$ is used for the derivability relation; an accurate use of it would require to point the logic we are deriving in as subscript, but also the concrete signature we are working with. However, except perhaps when presenting a definition or when we are interested in emphasizing some point, this extra information will not be written and we will let the context to disambiguate any possible conflict.

Acknowledgements. I would like to express my most sincere gratitude to my advisor, Narciso Martí Oliet, responsible to a great extent of my current position and reseach interests, and who has been always willing to help me, and not only throughout the development of this work. I specially thank José Meseguer, who pointed to us the right way to extend the notion of satisfaction in rewriting logic, and Mario Rodríguez Artalejo, who read a preliminary version of this paper and made some valuable comments. Finally, I also thank David de Frutos for kindly accepting to take the place of my advisor in the commitee entrusted with judging this work. Financial support through a predoctoral grant by the Spanish Ministry for Education, Culture and Sports, is gratefully acknowledged.

§2. General Logics

A general axiomatic theory of logics should cover all the key ingredients of a logic. These include: a *syntax*, a notion of *entailment* of a sentence from a set of axioms, a notion of *model*, and a notion of *satisfaction* of a sentence by a model. A flexible axiomatic notion of a *proof calculus*, in which proofs of entailments, not just the entailments themselves, are first class citizens should also be included (we will not be concerned about this issue in this work, though). The theory of *general logics* [19] is a study of these different ingredients and their interrelations, and the present section is a brief review of the main definitions that will be used later on, borrowing heavily from Martí-Oliet and Meseguer's paper [18].

2.1. Syntax

Syntax is typically given by a *signature* Σ providing a grammar on which *sentences* are built. For first order logic, a typical signature consists of a list of function symbols and a list of predicate symbols, each with a certain number of arguments, which are used to build up sentences by means of the usual logical connectives. For our purposes, and in order to allow maximum freedom and generality, it will be assumed that for each logic there is a category **Sign** of possible signatures for it, and a functor *sen* assigning to each signature Σ the set $sen(\Sigma)$ of all its sentences.

2.2. Entailment systems

For a given signature Σ in **Sign**, *entailment* (also called *provability*) of a sentence $\varphi \in sen(\Sigma)$ from a set of axioms $\Gamma \subseteq sen(\Sigma)$ is a relation $\Gamma \vdash \varphi$ which holds if and only if we can prove φ from the axioms Γ using the rules of the logic. We make this relation relative to a signature. In the rest of the paper, let $|\mathcal{C}|$ denote the collection of objects of a category $\mathcal{C}$.

An *entailment system* is a triple $\mathcal{E} = (\mathbf{Sign}, sen, \vdash)$ such that

- **Sign** is a category whose objects are called *signatures*,
- $sen : \mathbf{Sign} \to \mathbf{Set}$ is a functor associating to each signature Σ a corresponding set of Σ-*sentences*, and
- $\vdash$ is a function which associates to each $\Sigma \in |\mathbf{Sign}|$ a binary relation $\vdash_\Sigma \subseteq \mathcal{P}(sen(\Sigma)) \times sen(\Sigma)$ called Σ-*entailment* such that the following properties are satisfied:
    1. *reflexivity*: for any $\varphi \in sen(\Sigma)$, $\{\varphi\} \vdash_\Sigma \varphi$,
    2. *monotonicity*: if $\Gamma \vdash_\Sigma \varphi$ and $\Gamma' \supseteq \Gamma$ then $\Gamma' \vdash_\Sigma \varphi$,
    3. *transitivity*: if $\Gamma \vdash_\Sigma \varphi_i$, for all $i \in I$, and $\Gamma \cup \{\varphi_i \mid i \in I\} \vdash_\Sigma \psi$, then $\Gamma \vdash_\Sigma \psi$,
    4. $\vdash$-*translation*: if $\Gamma \vdash_\Sigma \varphi$, then for any $H : \Sigma \to \Sigma'$ in **Sign**, $sen(H)(\Gamma) \vdash_{\Sigma'} sen(H)(\varphi)$.

The entailment relation induces a function mapping each set of sentences Γ to the set $\Gamma^\bullet = \{\varphi \mid \Gamma \vdash \varphi\}$. We call $\Gamma^\bullet$ the set of *theorems* provable from Γ.

Given an entailment system $\mathcal{E}$, its category **Th** of *theories* has as objects pairs $T = (\Sigma, \Gamma)$, with Σ a signature and $\Gamma \subseteq sen(\Sigma)$. A *theory morphism* $H : (\Sigma, \Gamma) \to (\Sigma', \Gamma')$

is a signature morphism $H : \Sigma \to \Sigma'$ such that if $\varphi \in \Gamma$, then $\Gamma' \vdash_{\Sigma'} sen(H)(\varphi)$. A theory morphism is *axiom-preserving* if, in addition, it satisfies the condition $sen(H)(\Gamma) \subseteq \Gamma'$. This defines a subcategory $\mathbf{Th}_0$ with the same objects as **Th** but with morphisms restricted to be axiom-preserving theory morphisms, that does not depend on the entailment relation. By projecting on the first component we get a functor $sign : \mathbf{Th} \to \mathbf{Sign}$. Note also that the functor *sen* can be extended to a functor on theories by taking $sen(\Sigma, \Gamma) = sen(\Sigma)$.

2.3. Institutions

The notion of model is based on Goguen and Burstall's pioneering work on institutions (see [12]). An *institution* is a 4-tuple $\mathcal{I} = (\mathbf{Sign}, sen, \mathbf{Mod}, \models)$ such that

- **Sign** is a category whose objects are called *signatures*,
- $sen : \mathbf{Sign} \to \mathbf{Set}$ is a functor associating to each signature Σ a set of Σ-*sentences*,
- $\mathbf{Mod} : \mathbf{Sign}^{\mathrm{op}} \to \mathbf{Cat}$ is a functor that gives for each signature Σ a category whose objects are called Σ-*models*, and
- $\models$ is a function associating to each $\Sigma \in |\mathbf{Sign}|$ a binary relation $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times sen(\Sigma)$ called Σ-*satisfaction*, in such a way that the following property holds for any $H : \Sigma \to \Sigma'$, $M' \in |\mathbf{Mod}(\Sigma')|$ and all $\varphi \in sen(\Sigma)$:

$$M' \models_{\Sigma'} sen(H)(\varphi) \iff \mathbf{Mod}(H)(M') \models_\Sigma \varphi.$$

Given a set of Σ-sentences Γ, the category $\mathbf{Mod}(\Sigma, \Gamma)$ is defined as the full subcategory of $\mathbf{Mod}(\Sigma)$ determined by those models $M \in \mathbf{Mod}(\Sigma)$ that satisfy all the sentences in Γ. A relation between sets of sentences and sentences, also denoted as $\models$, can be defined by

$$\Gamma \models_\Sigma \varphi \iff M \models_\Sigma \varphi \text{ for each } M \in |\mathbf{Mod}(\Sigma, \Gamma)|.$$

We can then associate an entailment system to each institution $\mathcal{I} = (\mathbf{Sign}, sen, \mathbf{Mod}, \models)$ in a natural way by means of the triple $\mathcal{I}^+ = (\mathbf{Sign}, sen, \models)$, where $\models$ now denotes the previously defined relation between sets of sentences and sentences; $\mathcal{I}^+$ is easily seen to satisfy the conditions to be an entailment system.

Given an institution $\mathcal{I}$, its category **Th** of theories is defined as the category of theories associated to the entailment system $\mathcal{I}^+$. If $H : (\Sigma, \Gamma) \to (\Sigma', \Gamma')$ is a theory morphism and $M' \in \mathbf{Mod}(\Sigma', \Gamma')$, it is not difficult to check that $\mathbf{Mod}(H)(M') \in$

$\mathbf{Mod}(\Sigma, \Gamma)$. The model functor **Mod** can then be extended to a functor $\mathbf{Mod} : \mathbf{Th}^{\mathrm{op}} \to \mathbf{Cat}$.

Liberality is an important property expressing the possibility of free constructions generalizing the principle of "initial algebra semantics". An institution is *liberal* if all the forgetful functors $\mathbf{Mod}(H) : \mathbf{Mod}(\Sigma', \Gamma') \to \mathbf{Mod}(\Sigma, \Gamma)$ induced by the theory morphisms $H : (\Sigma, \Gamma) \to (\Sigma', \Gamma')$ have left adjoints.

Another property expressing the possibility of "putting theories together" by colimits is the exactness of an institution. An institution is *exact* if its category of signatures is cocomplete and the model functor $\mathbf{Mod} : \mathbf{Sign}^{\mathrm{op}} \to \mathbf{Cat}$ preserves colimits, and is *semiexact* if **Sign** has pushouts and **Mod** preserves pushouts.

2.4. Logics

Defining a *logic* is now immediate. A *logic* is a 5-tuple $\mathcal{L} = (\mathbf{Sign}, sen, \mathbf{Mod}, \vdash, \models)$ such that:

- $(\mathbf{Sign}, sen, \vdash)$ is an entailment system,
- $(\mathbf{Sign}, sen, \mathbf{Mod}, \models)$ is an institution, and
- the following *soundness condition* is satisfied: for any $\Sigma \in |\mathbf{Sign}|$, $\Gamma \subseteq sen(\Sigma)$, and $\varphi \in sen(\Sigma)$,
$$\Gamma \vdash_\Sigma \varphi \implies \Gamma \models_\Sigma \varphi.$$

The logic is called *complete* if the above implication is in fact an equivalence.

2.5. Mapping logics

The advantage of having an axiomatic theory of logics is that the "space" of all logics (or that of all entailment systems, institutions, etc.) becomes well understood. This space is not just a collection of objects bearing no relationship to each other; in fact, the most interesting outcome of the theory of general logics is that it gives us a method for *relating* logics in a general and systematic way, and to exploit such relations in many applications, by considering *maps* that interpret one logic into another.

Let us first discuss in some detail *maps of entailment systems.* Basically, a map of entailment systems $\mathcal{E} \to \mathcal{E}'$ maps the language of $\mathcal{E}$ to that of $\mathcal{E}'$ in a way that respects the entailment relation. This means that signatures of $\mathcal{E}$ are functorially mapped to signatures of $\mathcal{E}'$, and that sentences of $\mathcal{E}$ are mapped to sentences of $\mathcal{E}'$ in a way that is coherent with the mapping of their corresponding signatures. In addition, such a mapping α must respect the entailment relations $\vdash$ of $\mathcal{E}$ and $\vdash'$ of $\mathcal{E}'$. It turns out that for many interesting applications one wants to be more general and allow maps that send a signature of $\mathcal{E}$ to a *theory* of $\mathcal{E}'$. These maps extend to maps between theories, and in this context the coherence with the mapping at the level of signatures is expressed by the notion of *sensible functor.* In what follows, let us denote by (Σ', Γ') the image obtained by applying to a theory (Σ, Γ) a functor $\Phi : \mathbf{Th}_0 \to \mathbf{Th}_0'$ preserving signatures; in particular, the theory $\Phi(\Sigma, \emptyset)$ will be denoted by $(\Sigma', \emptyset')$.

Given entailment systems $\mathcal{E} = (\mathbf{Sign}, sen, \vdash)$ and $\mathcal{E}' = (\mathbf{Sign}', sen', \vdash')$, a functor $\Phi : \mathbf{Th}_0 \to \mathbf{Th}_0'$ and a natural transformation $\alpha : sen \Rightarrow sen' \circ \Phi$, we say that Φ is *α-sensible* if the following conditions are satisfied:

1. There is a functor $\Phi^\diamond : \mathbf{Sign} \to \mathbf{Sign}'$ such that $sign' \circ \Phi = \Phi^\diamond \circ sign$.
2. $(\Gamma')^\bullet = (\emptyset' \cup \alpha_\Sigma(\Gamma))^\bullet$.

Essentially, this means that Φ is completely determined by its restriction to empty theories and α. It is proved in [19] that the natural transformation α only depends on the signatures, not on the theories.

Now, the definition sketched above can be formally expressed as follows. Given entailment systems $\mathcal{E} = (\mathbf{Sign}, sen, \vdash)$ and $\mathcal{E}' = (\mathbf{Sign}', sen', \vdash')$, a *map of entailment systems* $(\Phi, \alpha) : \mathcal{E} \to \mathcal{E}'$ consists of a natural transformation $\alpha : sen \Rightarrow sen' \circ \Phi$ and an α-sensible functor $\Phi : \mathbf{Th}_0 \to \mathbf{Th}_0'$ satisfying the following property:

$$\Gamma \vdash_\Sigma \varphi \implies \Gamma' \vdash'_{\Sigma'} \alpha_\Sigma(\varphi).$$

We call (Φ, α) *conservative* when the above implication is an equivalence.

The ideas behind *maps of institutions* are similar. Now, in addition, we will have a natural transformation $\beta : \mathbf{Mod}' \circ \Phi^{\mathrm{op}} \Rightarrow \mathbf{Mod}$ which, like model functors, maps models "backwards" and is subject to a certain satisfaction condition.

Given institutions $\mathcal{I} = (\mathbf{Sign}, sen, \mathbf{Mod}, \models)$ and $\mathcal{I}' = (\mathbf{Sign}', sen', \mathbf{Mod}', \models')$, a *map of institutions* $(\Phi, \alpha, \beta) : \mathcal{I} \to \mathcal{I}'$ consists of a natural transformation $\alpha : sen \Rightarrow sen' \circ \Phi$, an α-sensible functor[1] $\Phi : \mathbf{Th}_0 \to \mathbf{Th}_0'$, and a natural transformation $\beta : \mathbf{Mod}' \circ \Phi^{\mathrm{op}} \Rightarrow \mathbf{Mod}$ such that for each $\Sigma \in |\mathbf{Sign}|$, $\varphi \in sen(\Sigma)$, and $M' \in |\mathbf{Mod}'(\Phi(\Sigma, \emptyset))|$ the following property is satisfied:

$$M' \models'_{\Sigma'} \alpha_\Sigma(\varphi) \iff \beta_{(\Sigma,\emptyset)}(M') \models_\Sigma \varphi.$$

---

[1] The functor Φ is α-sensible for the entailment systems $(\mathbf{Sign}, sen, \models)$ and $(\mathbf{Sign}', sen', \models')$ associated to $\mathcal{I}$ and $\mathcal{I}'$.

The definition of *map of logics* follows now immediately: it is given by two maps, one for its underlying entailment system and another for its underlying institution, which agree on the translation of signatures and sentences. Thus, given logics $\mathcal{L} = (\mathbf{Sign}, \mathit{sen}, \mathbf{Mod}, \vdash, \models)$ and $\mathcal{L}' = (\mathbf{Sign}', \mathit{sen}', \mathbf{Mod}', \vdash', \models')$, a *map of logics* $(\Phi, \alpha, \beta) : \mathcal{L} \to \mathcal{L}'$ consists of a functor $\Phi : \mathbf{Th}_0 \to \mathbf{Th}'_0$, and natural transformations $\alpha : \mathit{sen} \Rightarrow \mathit{sen}' \circ \Phi$ and $\beta : \mathbf{Mod}' \circ \Phi^{\mathrm{op}} \Rightarrow \mathbf{Mod}$ such that:

- $(\Phi, \alpha) : (\mathbf{Sign}, \mathit{sen}, \vdash) \to (\mathbf{Sign}', \mathit{sen}', \vdash')$ is a map of entailment systems, and
- $(\Phi, \alpha, \beta) : (\mathbf{Sign}, \mathit{sen}, \mathbf{Mod}, \models) \to (\mathbf{Sign}', \mathit{sen}', \mathbf{Mod}', \models')$ is a map of institutions.

We call (Φ, α, β) *conservative* if and only if (Φ, α) is so as a map of entailment systems.

§3. The Logic CRWL

Constructor-based ReWriting Logic (CRWL) is a logic developed at the Declarative Programming Group in the Universidad Complutense de Madrid. This (untyped) logic has been described in [14], where it is proposed as a framework in which “to model the evaluation of terms in a constructor-based language including non-strict and possibly non-deterministic functions, so that it can serve as a logical basis for declarative programming languages involving lazy evaluation”. The present section reviews the basic definitions presented in that paper.

3.1. Signatures

CRWL uses *signatures with constructors*, which are countable sets $\Sigma = C_\Sigma \cup F_\Sigma$, where $C_\Sigma = \bigcup_{n \in \mathbb{N}} C_\Sigma^n$ and $F_\Sigma = \bigcup_{n \in \mathbb{N}} F_\Sigma^n$ are disjoint sets of *constructor* and *defined function symbols* respectively, each of them with an associated arity. Given a countable set $\mathcal{X}$ of variables, we will write $\mathit{Expr}(\Sigma, \mathcal{X})$ for the set of expressions which can be built with Σ and $\mathcal{X}$, and we will distinguish the subset $\mathit{Term}(\Sigma, \mathcal{X})$ of those terms which only make use of C_Σ and $\mathcal{X}$. We will also use $\Sigma_\perp$ to refer to the signature which is obtained from Σ by adding to it a new constructor $\perp$ of arity 0, and $\mathit{Expr}_\perp(\Sigma, \mathcal{X})$ and $\mathit{Term}_\perp(\Sigma, \mathcal{X})$ for the associated (partial) expressions and terms.

In Juan Miguel Molina’s thesis [24], *signature morphisms* are also defined. A signature morphism $\sigma : \Sigma \to \Sigma'$ from a signature $\Sigma = C_\Sigma \cup F_\Sigma$ to another $\Sigma' = C_{\Sigma'} \cup F_{\Sigma'}$ is a pair of functions (denoted with the same σ)

$$\sigma : C_\Sigma \to C_{\Sigma'} \quad \text{and} \quad \sigma : F_\Sigma \to F_{\Sigma'},$$

mapping n-ary symbols to n-ary symbols.

A signature morphism $\sigma : \Sigma \to \Sigma'$ can be extended to a map between the corresponding expressions as follows:

$$\begin{array}{rcll}
\sigma(\bot) & = & \bot; & \\
\sigma(x) & = & x, & \forall x \in \mathcal{X}; \\
\sigma(c) & = & \sigma(c), & \forall c \in C^0_\Sigma \cup F^0_\Sigma; \\
\sigma(h(t_1,\ldots,t_n)) & = & \sigma(h)(\sigma(t_1),\ldots,\sigma(t_n)), & \forall h \in C^n_\Sigma \cup F^n_\Sigma, n > 0.
\end{array}$$

3.2. Sentences, theories and provability

Given a signature Σ, a *CRWL-program* is defined as a set Γ of conditional rewrite rules of the form:

$$f(t_1,\ldots,t_n) \to r \Leftarrow a_1 \bowtie b_1,\ldots,a_m \bowtie b_m \quad (m \geq 0)$$

with $f \in F^n_\Sigma$, $t_1,\ldots,t_n \in \mathit{Term}(\Sigma,\mathcal{X})$, $a_i, b_i \in \mathit{Expr}(\Sigma,\mathcal{X}), i = 1,\ldots,m$, and each variable occurring in $t_1,\ldots,t_n$ having a single ocurrence. In [14] CRWL-programs are also called CRWL-theories. Here, we will reserve that name for pairs $T = (\Sigma,\Gamma)$, as it is usually done. From a given theory (Σ,Γ) we are interested in deriving sentences of two kinds:

- *Reduction statements*: $a \to b$, with $a, b \in \mathit{Expr}_\bot(\Sigma,\mathcal{X})$.
- *Joinability statements*: $a \bowtie b$, with $a, b \in \mathit{Expr}_\bot(\Sigma,\mathcal{X})$.

Among reduction statements, those with $b \in \mathit{Term}_\bot(\Sigma,\mathcal{X})$ are called *approximation statements.*

Given a signature morphism σ, the extension to rewrite rules is immediate by defining $\sigma(a \bowtie b) = \sigma(a) \bowtie \sigma(b)$, and

$$\sigma(l \to r \Leftarrow C) = \sigma(l) \to \sigma(r) \Leftarrow \sigma(C).$$

To define formal derivability from a theory $T = (\Sigma,\Gamma)$ two rewriting calculus are introduced: a Basic Rewriting Calculus (BRC) and a Goal-Oriented Rewriting Calculus (GORC). BRC is more natural than GORC, but GORC is used to derive the operational semantics of CRWL. Besides, GORC does not allow to derive reduction statements in its full generality, but only approximation statements; however, for them it is equivalent to BRC. In the following, only BRC will be used, whose rules are:

1. **Bottom (B).** For each $e \in \mathit{Expr}_\perp(\Sigma, \mathcal{X})$,
$$\frac{}{e \rightarrow \perp}.$$
2. **Monotonicity (MN).** For each $h \in C^n_\Sigma \cup F^n_\Sigma$, $n \in \mathbb{N}$,
$$\frac{e_1 \rightarrow e_1' \quad \dots \quad e_n \rightarrow e_n'}{h(e_1, \dots, e_n) \rightarrow h(e_1', \dots, e_n')}.$$
3. **Reflexivity (RF).** For each $e \in \mathit{Expr}_\perp(\Sigma, \mathcal{X})$,
$$\frac{}{e \rightarrow e}.$$
4. **Reduction (R).** For each $(l \rightarrow r \Leftarrow C) \in [\Gamma]_\perp$,
$$\frac{C}{l \rightarrow r},$$
where $[\Gamma]_\perp$ is the set of all instances of rewrite rules in Γ, obtained by substituting elements in $\mathit{Term}_\perp(\Sigma, \mathcal{X})$ for variables.
5. **Transitivity (TR).**
$$\frac{e \rightarrow e' \qquad e' \rightarrow e''}{e \rightarrow e''}$$
6. **Join (J).**
$$\frac{a \rightarrow t \qquad b \rightarrow t}{a \bowtie b},$$
if $t \in \mathit{Term}(\Sigma, \mathcal{X})$.

We say that T *entails* $e \rightarrow e'$ and write $T \vdash_{\mathrm{CRWL}} e \rightarrow e'$ if $e \rightarrow e'$ can be obtained by finite application of the above rules. By an abuse of notation we will sometimes simply write $\Gamma \vdash_{\mathrm{CRWL}} e \rightarrow e'$.

3.3. Models

Before defining models we review some definitions. A *partially ordered set* (in short, poset) with bottom $\perp$ is a set S equipped with a partial order $\sqsubseteq$ and a least element $\perp$. We say that an element $x \in S$ is *totally defined* if x is maximal with respect to $\sqsubseteq$. The set of all totally defined elements of S will be denoted $\mathit{Def}(S)$. $D \subseteq S$ is a *directed* set if for all $x, y \in D$ there exists $z \in D$ with $x \sqsubseteq z$, $y \sqsubseteq z$. A subset $A \subseteq S$ is a *cone* if $\perp \in D$ and, for all $x \in A$, $y \in S$, if $y \sqsubseteq x$ then $y \in A$. An *ideal* $I \subseteq S$ is a directed cone. For $x \in S$, the principal ideal generated by x is $\langle x \rangle = \{y \in S \mid y \sqsubseteq x\}$. We write $\mathcal{C}(S)$, $\mathcal{I}(S)$ for the sets of cones and ideals of S respectively.

Given a signature Σ, a *CRWL-algebra* over Σ is a triple

$$\mathcal{A} = (D^{\mathcal{A}}, \{c^{\mathcal{A}}\}_{c \in C_\Sigma}, \{f^{\mathcal{A}}\}_{f \in F_\Sigma}),$$

where $D^{\mathcal{A}}$ is a poset with bottom, and $c^{\mathcal{A}}$ and $f^{\mathcal{A}}$ are monotone mappings from $(D^{\mathcal{A}})^n$ to $\mathcal{C}(D^{\mathcal{A}})$, with n the corresponding arity. In addition, for $c \in C^n_\Sigma$ and for all $u_1, \ldots, u_n \in D^{\mathcal{A}}$, there exists a $v \in D^{\mathcal{A}}$ such that $c^{\mathcal{A}}(u_1, \ldots, u_n) = \langle v \rangle$. Moreover, $v \in Def(D^{\mathcal{A}})$ in case that all $u_i \in Def(D^{\mathcal{A}})$.

Note that any $h : S \to \mathcal{C}(S')$ can be extended to a function $\hat{h} : \mathcal{C}(S) \to \mathcal{C}(S')$ defined by $\hat{h}(x) = \bigcup_{x \in S} h(x)$. By an abuse of notation, we will write $\hat{h}$ also as h in the sequel.

A *valuation* over $\mathcal{A}$ is any mapping $\eta : \mathcal{X} \to D^{\mathcal{A}}$, and we say that η is *totally defined* if $\eta(x) \in Def(D^{\mathcal{A}})$ for all $x \in \mathcal{X}$. The *evaluation* of an expression $e \in Expr_\perp(\Sigma, \mathcal{X})$ in $\mathcal{A}$ under η yields $[\![e]\!]^{\mathcal{A}}\eta \in \mathcal{C}(D^{\mathcal{A}})$ which is defined recursively as follows:

- $[\![\perp]\!]^{\mathcal{A}}\eta = \langle \perp_{\mathcal{A}} \rangle$.
- $[\![x]\!]^{\mathcal{A}}\eta = \langle \eta(x) \rangle$, for $x \in \mathcal{X}$.
- $[\![h(e_1, \ldots, e_n)]\!]^{\mathcal{A}}\eta = h^{\mathcal{A}}([\![e_1]\!]^{\mathcal{A}}\eta, \ldots, [\![e_1]\!]^{\mathcal{A}}\eta)$, for all $h \in C^n_\Sigma \cup F^n_\Sigma$.

We are now prepared to define models. Assume a program Γ and a CRWL-algebra $\mathcal{A}$. We define:

- $\mathcal{A}$ satisfies a reduction statement $a \to b$ under a valuation η, $(\mathcal{A}, \eta) \models a \to b$, if $[\![a]\!]^{\mathcal{A}}\eta \supseteq [\![b]\!]^{\mathcal{A}}\eta$.
- $\mathcal{A}$ satisfies a joinability statement $a \bowtie b$ under η, $(\mathcal{A}, \eta) \models a \bowtie b$, if $[\![a]\!]^{\mathcal{A}}\eta \cap [\![b]\!]^{\mathcal{A}}\eta \cap Def(D^{\mathcal{A}}) \neq \emptyset$.
- $\mathcal{A}$ satisfies a rule $l \to r \Leftarrow C$ if every valuation η such that $(\mathcal{A}, \eta) \models C$ verifies $(\mathcal{A}, \eta) \models l \to r$.
- $\mathcal{A}$ is a *model* of Γ, $\mathcal{A} \models \Gamma$ if $\mathcal{A}$ satisfies all the rules is Γ.

Finally, we can also define homomorphisms between CRWL-algebras. Let $\mathcal{A}$, $\mathcal{B}$ be two CRWL-algebras over a signature Σ. A *CRWL-homomorphism* $H : \mathcal{A} \to \mathcal{B}$ is a monotone function $H : D^{\mathcal{A}} \to \mathcal{C}(D^{\mathcal{B}})$ which satisfies the following conditions:

1. H is element-valued: for all $u \in D^{\mathcal{A}}$ there exists $v \in D^{\mathcal{B}}$ such that $H(u) = \langle v \rangle$.
2. H is strict: $H(\bot_{\mathcal{A}}) = \langle \bot_{\mathcal{B}} \rangle$.
3. H preserves constructors: for all $c \in C^n_\Sigma$, $u_i \in D^{\mathcal{A}}$, is $H(c^{\mathcal{A}}(u_1, \ldots, u_n)) = c^{\mathcal{B}}(H(u_1), \ldots, H(u_n))$.
4. H loosely preserves defined functions: that is, for all $f \in F^n_\Sigma$, $u_i \in D^{\mathcal{A}}$, $H(f^{\mathcal{A}}(u_1, \ldots, u_n)) \subseteq f^{\mathcal{B}}(H(u_1), \ldots, H(u_n))$.

CRWL-algebras as objects with CRWL-homomorphisms as arrows form a category.

§4. Rewriting Logic

Rewriting logic (RL for short) is a logic that was developed by José Meseguer in the early nineties [20, 21], aiming at the unification of different approaches to concurrency. During this last decade, RL has proved to be a powerful and flexible tool, very suitable as both a logical and a semantic framework [18]. In this section we review its rules of deduction as well as its semantics, borrowing heavily from [21].

4.1. Basic universal algebra

RL is parameterized with respect to the version of the underlying equational logic, which can be unsorted, many-sorted, order-sorted, or the recently developed membership equational logic [22]. For the sake of simplifying the exposition, we treat here the *unsorted* case.

A set Σ of function symbols is a ranked alphabet $\Sigma = \{\Sigma_n \mid n \in \mathbb{N}\}$. A Σ-algebra is then a set A together with an assignment of a function $f_A : A^n \to A$ for each $f \in \Sigma_n$ with $n \in \mathbb{N}$. We denote by T_Σ the Σ-algebra of ground Σ-terms, and by $T_\Sigma(\mathcal{X})$ the Σ-algebra of Σ-terms with variables in a set $\mathcal{X}$. Similarly, given a set E of Σ-equations, $T_{\Sigma,E}$ denotes the Σ-algebra of equivalence classes of ground Σ-terms modulo the equations E; in the same way, $T_{\Sigma,E}(\mathcal{X})$ denotes the Σ-algebra of equivalence classes of Σ-terms with variables in $\mathcal{X}$ modulo the equations E. Let $[t]_E$ or just $[t]$ denote the E-equivalence class of t.

Given a term $t \in T_\Sigma(\{x_1, \ldots, x_n\})$, and terms $u_1, \ldots, u_n$, $t(u_1/x_1, \ldots, u_n/x_n)$ denotes the term obtained from t by *simultaneously substituting* u_i for x_i, $i = 1, \ldots, n$. To simplify notation, we denote a sequence of objects $a_1, \ldots, a_n$ by $\overline{a}$; with this notation, $t(u_1/x_1, \ldots, u_n/x_n)$ can be abbreviated to $t(\overline{u}/\overline{x})$.

4.2. The rules of RL

A *signature* in RL is a pair (Σ, E) with Σ a ranked alphabet of function symbols and E a set of Σ-equations. Rewriting will operate on equivalence classes of terms modulo the set of equations E. In this way, we free rewriting from the syntactic constraints of a term representation and gain a much greater flexibility in deciding what counts as a *data structure*; for example, string rewriting is obtained by imposing an associativity axiom, and multiset rewriting by imposing associativity and commutativity. Of course, standard term rewriting is obtained as the particular case in which the set E of equations is empty.

Given a signature (Σ, E), *sentences* of RL are "sequents" of the form $[t]_E \to [t']_E$, where t and t' are Σ-terms possibly involving some variables from the countably infinite set $\mathcal{X} = \{x_1, \ldots, x_n, \ldots\}$. A *theory* in this logic, called an RL-theory, is a slight generalization of the usual notion of theory in that we allow the axioms—in this case the sequents $[t]_E \to [t']_E$—to be labelled. This is very natural for many applications, and customary for automata—viewed as labelled transition systems—and for Petri nets, which are both particular instances of our definition.

A *RL-theory* $\mathcal{R}$ is a 4-tuple $\mathcal{R} = (\Sigma, E, L, \Gamma)$ where Σ is a ranked alphabet of function symbols, E is a set of Σ-equations, L is a set of *labels*, and Γ is a set of pairs $\Gamma \subseteq L \times (T_{\Sigma,E}(\mathcal{X})^2)^+$ whose first component is a label and whose second component is a non-empty sequence of pairs of E-equivalence classes of terms, with $\mathcal{X} = \{x_1, \ldots, x_n, \ldots\}$ a countably infinite set of variables. Elements of Γ are called *rewrite rules*. For a rewrite rule $(r, ([t], [t'])([a_1], [b_1]) \ldots ([a_m], [b_m]))$ we use the notation

$$r : [t] \to [t'] \text{ \textit{if} } [a_1] \to [b_1] \wedge \ldots \wedge [a_m] \to [b_m].$$

To indicate that $\{x_1, \ldots, x_n\}$ is the set of variables occurring in either t, t' or C, we write $r : [t(x_1, \ldots, x_n)] \to [t'(x_1, \ldots, x_n)]$ *if* $C(x_1, \ldots, x_n)$, or in abbreviated notation $r : [t(\overline{x})] \to [t'(\overline{x})]$ *if* $C(\overline{x})$.

Given an RL-theory $\mathcal{R}$, we say that $\mathcal{R}$ *entails* a sequent $[t] \to [t']$ and write $\mathcal{R} \vdash_{\mathrm{RL}} [t] \to [t']$ if and only if $[t] \to [t']$ can be obtained by finite application of the following *rules of deduction*:

1. **Reflexivity (RF)**. For each $[t] \in T_{\Sigma,E}(\mathcal{X})$,

$$\frac{}{[t] \to [t]}.$$

2. **Congruence (CG)**. For each $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$\frac{[t_1] \to [t'_1] \quad \ldots \quad [t_n] \to [t'_n]}{[f(t_1, \ldots, t_n)] \to [f(t'_1, \ldots, t'_n)]}.$$

3. **Replacement (RP).** For each rewrite rule $r : [t(\overline{x})] \to [t'(\overline{x})]$ *if* $[a_1(\overline{x})] \to [b_1(\overline{x})] \wedge \ldots \wedge [a_m(\overline{x})] \to [b_m(\overline{x})]$ in Γ,

$$\frac{\begin{array}{c}[w_1] \to [w_1'] \quad \ldots \quad [w_n] \to [w_n'] \\ [a_1(\overline{w}/\overline{x})] \to [b_1(\overline{w}/\overline{x})] \quad \ldots \quad [a_m(\overline{w}/\overline{x})] \to [b_m(\overline{w}/\overline{x})]\end{array}}{[t(\overline{w}/\overline{x})] \to [t'(\overline{w'}/\overline{x})]}.$$

4. **Transitivity (TR).**

$$\frac{[t_1] \to [t_2] \qquad [t_2] \to [t_3]}{[t_1] \to [t_3]}.$$

A nice consequence of having defined rewriting logic is that concurrent rewriting, rather than emerging as an operational notion, actually coincides with deduction in such a logic. Given an RL-theory $\mathcal{R} = (\Sigma, E, L, \Gamma)$, a (Σ, E)-sequent $[t] \to [t']$ is called a *concurrent $\mathcal{R}$-rewrite* (or just a *rewrite*) if and only if it can be derived from $\mathcal{R}$ by finite application of the rules 1-4, i.e., $\mathcal{R} \vdash_{\mathrm{RL}} [t] \to [t']$.

4.3. The models of RL

Before proceeding to *$\mathcal{R}$-systems*, the models of RL, we need the categorical notion of *subequalizer* (see [16]), a notion generalizing that of equalizer of two functors[2].

Given a family of pairs of functors $\{F_i, G_i : \mathcal{A} \to \mathcal{B}_i \mid i \in I\}$, the (simultaneous) *subequalizer* of this family is a category $\mathit{Subeq}((F_i, G_i)_{i \in I})$ together with a functor

$$J : \mathit{Subeq}((F_i, G_i)_{i \in I}) \to \mathcal{A}$$

and a family of natural transformations $\{\alpha_i : F_i \circ J \Rightarrow G_i \circ J \mid i \in I\}$ satisfying the following universal property: Given a functor $H : \mathcal{C} \to \mathcal{A}$ and a family of natural transformations $\{\beta_i : F_i \circ H \Rightarrow G_i \circ H \mid i \in I\}$, there exists a unique functor $(H, \{\beta_i\}_{i \in I}) : \mathcal{C} \to \mathit{Subeq}((F_i, G_i)_{i \in I})$ such that

$$J \circ (H, \{\beta_i\}_{i \in I}) = H \quad \text{and} \quad \alpha_i \circ (H, \{\beta_i\}_{i \in I}) = \beta_i \quad (i \in I).$$

---

[2] In [23], subequalizers are shown to coincide with *inserters*, a special kind of weighted limit, in the 2-category **Cat**. This allows the author to generalize the models of RL, building them over arbitrary 2-categories and even enriched categories.

Schematically, the following diagram must be commutative:

$$\begin{tikzcd}
 & & \mathcal{A} \arrow[rr, "F_i"] \arrow[dd, "id_{\mathcal{A}}"] \arrow[dr, phantom, "\overset{\alpha_i}{\Rightarrow}"] & & \mathcal{B}_i \arrow[dd, "id_{\mathcal{B}_i}"] \\
\mathit{Subeq}(F_j, G_j) \arrow[urr, "J"] \arrow[rrr, "J"] & & & \mathcal{A} \arrow[ur, "G_i"'] \arrow[dd, "id_{\mathcal{A}}"] & \\
 & & \mathcal{A} \arrow[rr, "F_i"] \arrow[dr, phantom, "\overset{\beta_i}{\Rightarrow}"] & & \mathcal{B}_i \\
\mathcal{C} \arrow[uu, "{(H, \{\beta_j\})}"] \arrow[urr, "H"] \arrow[rrr, "H"'] & & & \mathcal{A} \arrow[ur, "G_i"'] &
\end{tikzcd}$$

The construction of $\mathit{Subeq}((F_i, G_i)_{i \in I})$ is quite simple. Its objects are pairs $(A, \{b_i\}_{i \in I})$ with A an object in $\mathcal{A}$ and $b_i : F_i(A) \to G_i(A)$ a morphism in $\mathcal{B}_i$. Morphisms

$$a : (A, \{b_i\}_{i \in I}) \to (A', \{b'_i\}_{i \in I})$$

are morphisms $a : A \to A'$ in $\mathcal{A}$ such that for each $i \in I$, $G_i(a) \circ b_i = b'_i \circ F_i(a)$. The functor J is just projection into the first component. The natural transformations α_j are defined by

$$\alpha_j(A, \{b_i\}_{i \in I}) = b_j \quad (j \in I).$$

Then, given an RL-theory $\mathcal{R} = (\Sigma, E, L, \Gamma)$, an *$\mathcal{R}$-system* $\mathcal{S}$ is a category $\mathcal{S}$ together with:

- a (Σ, E)-algebra structure given by a family of functors

  $$\{f_{\mathcal{S}} : \mathcal{S}^n \to \mathcal{S} \mid f \in \Sigma_n, n \in \mathbb{N}\}$$

  satisfying the equations E, i.e., for any $t(x_1, \dots, x_n) = t'(x_1, \dots, x_n)$ in E we have an identity of functors $t_{\mathcal{S}} = t'_{\mathcal{S}}$, where the functor $t_{\mathcal{S}}$ is defined inductively from the functors $f_{\mathcal{S}}$ in the obvious way.

- for each rewrite rule

  $$r : [t(\overline{x})] \to [t'(\overline{x})] \;\mathit{if}\; [a_1(\overline{x})] \to [b_1(\overline{x})] \wedge \ldots \wedge [a_m(\overline{x})] \to [b_m(\overline{x})]$$

  in Γ, a natural transformation

  $$r_{\mathcal{S}} : t_{\mathcal{S}} \circ J_{\mathcal{S}} \Rightarrow t'_{\mathcal{S}} \circ J_{\mathcal{S}},$$

  where $J_{\mathcal{S}} : \mathit{Subeq}((a_{j\mathcal{S}}, b_{j\mathcal{S}})_{1 \le j \le m}) \to \mathcal{S}^n$ is the subequalizer functor.

An $\mathcal{R}$-*homomorphism* $F : \mathcal{S} \to \mathcal{S}'$ between two $\mathcal{R}$-systems is then a functor $F : \mathcal{S} \to \mathcal{S}'$ such that

- it is a Σ-algebra homomorphism, i.e., $F \circ f_{\mathcal{S}} = f_{\mathcal{S}'} \circ F^n$, for each f in Σ_n, $n \in \mathbb{N}$, and
- "F preserves Γ", i.e., for each rewrite rule

  $$r : [t(\overline{x})] \to [t'(\overline{x})] \textit{ if } C$$

  in Γ we have the identity of natural transformations

  $$F \circ r_{\mathcal{S}} = r_{\mathcal{S}'} \circ F^{\bullet},$$

  where $F^{\bullet} : \mathit{Subeq}(C_{\mathcal{S}}) \to \mathit{Subeq}(C_{\mathcal{S}'})$ is the unique functor induced by the universal property of $\mathit{Subeq}(C_{\mathcal{S}'})$ by the composition functor

  $$\mathit{Subeq}(C_{\mathcal{S}}) \xrightarrow{J_{\mathcal{S}}} \mathcal{S}^n \xrightarrow{F^n} \mathcal{S}'^n$$

  and the natural transformations $F \circ \alpha_j$, $1 \leq j \leq m$, where the condition C has m rewrites $[a_j] \to [b_j]$, and α_j is the jth natural transformation associated to the subequalizer $\mathit{Subeq}(C_{\mathcal{S}})$. That is, the following diagram must commute:

  $$\begin{tikzcd}
  & \mathcal{S}^n \arrow[rr, "t_{\mathcal{S}}"] \arrow[dd, "F^n"] & & \mathcal{S} \arrow[dd, "F"] \\
  \mathit{Subeq}(a_{j\mathcal{S}}, b_{j\mathcal{S}}) \arrow[ur, "J_{\mathcal{S}}"] \arrow[rr, "J_{\mathcal{S}}"] \arrow[dd, "F^{\bullet}"'] & & \mathcal{S}^n \arrow[ur, "t'_{\mathcal{S}}"'] \arrow[dd, "F^n"] & \\
  & \mathcal{S}'^n \arrow[rr, "t_{\mathcal{S}'}"] & & \mathcal{S}' \\
  \mathit{Subeq}(a_{j\mathcal{S}'}, b_{j\mathcal{S}'}) \arrow[ur, "J_{\mathcal{S}'}"] \arrow[rr, "J'_{\mathcal{S}}"'] & & \mathcal{S}' \arrow[ur, "t'_{\mathcal{S}'}"'] &
  \arrow[from=1-2, to=2-3, phantom, "\overset{r_{\mathcal{S}}}{\Rightarrow}"]
  \arrow[from=3-2, to=4-3, phantom, "\overset{r_{\mathcal{S}'}}{\Rightarrow}"]
  \end{tikzcd}$$

  Despite the somewhat complicated definition of $F^{\bullet}$, its behaviour on objects is quite simple; it is given by the equation

  $$F^{\bullet}(\overline{C}^n, \overline{c}^m) = (F^n(\overline{C}^n), F^m(\overline{c}^m)).$$

This defines a category $\mathcal{R}$-**Sys** in the obvious way.

A sequent $[t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$ is satisfied by a $\mathcal{R}$-system $\mathcal{S}$ if there exists a natural transformation

$$\alpha : t_{\mathcal{S}} \Rightarrow t'_{\mathcal{S}}$$

between the functors $t_{\mathcal{S}}, t'_{\mathcal{S}} : \mathcal{S}^n \to \mathcal{S}$. We use the notation

$$\mathcal{S} \models [t(x_1,\ldots,t_n)] \to [t'(x_1,\ldots,x_n)].$$

With respect to this definition of satisfaction, the proof calculus is sound and complete [21]. Completeness is obtained by means of an initial model construction.

§5. An Entailment System for CRWL?

According to the data presented in Section 3 (theories, derivability, rewriting calculus) we can try to associate an entailment system to CRWL. The category of signatures is immediately obtained, as it is not difficult to check that composition of signature morphisms is associative. Nevertheless, it is not so clear how to define the functor *sen* : **Sign** $\to$ **Set**. Following the definitions in Section 3 we have three natural possibilities (together with their combinations): assigning to each signature the set of all its conditional rewrite rules, or the set of all its reduction statements, or the set of its approximation statements. The last two options are too limited because they would only allow unconditional theories; the first one poses (at least at first sight) the problem of defining how to derive a conditional rule within the calculus.

For the time being, let us associate to each signature the corresponding set of conditional rewrite rules (it is also Juan Miguel Molina's choice in his thesis [24], in order to define an institution associated to CRWL). For a reduction statement φ let us define $\Gamma \vdash \varphi$ if and only if φ can be derived from Γ using BRC. Then, no matter what the definition of $\Gamma \vdash \varphi$ is when φ is conditional, we won't have an entailment system. The reason is that the relation $\vdash$ is not transitive (even for regular rules, i.e., those which do not introduce new variables in the righthand side of the rewriting relation). For example, let Σ be a signature with $C_\Sigma = \emptyset$, $c, d \in F^0_\Sigma$, $h \in F^1_\Sigma$. Then, bearing in mind that $h(c) \to h(\bot)$ is always provable (with rules **B** and **MN** on page 11),

$$\begin{array}{rcl} \{\, c \to h(c), h(x) \to h(d), h(x) \to h(d) \Leftarrow x \bowtie x, \,\} & \vdash & c \to h(d) \\ \{\, c \to h(c), h(x) \to h(d) \Leftarrow x \bowtie x \,\} & \vdash & h(x) \to h(d), \end{array}$$

but

$$\{\, c \to h(c), h(x) \to h(d) \Leftarrow x \bowtie x \,\} \not\vdash c \to h(d).$$

The first statement is proved by instantiating $h(x) \to h(d)$ with $\perp$ and applying transitivity (note that c cannot be used to instantiate this rule, as it is not a term); for the second, simply instantiate $h(x) \to h(d) \Leftarrow x \bowtie x$ with x. The third statement is formally proved by induction on derivations; let us just note that the crucial point is that the rule $h(x) \to h(d) \Leftarrow x \bowtie x$ cannot be instantiated with $\perp$. This proves that we are not going to be able to build an entailment system based on the CRWL rewriting calculus, as any sensible one should contain, at least, the conditional rewrite rules among its sentences.

We will have, however, an entailment system corresponding to the institution that will be associated to CRWL in Section 13. The previous example is no longer a counterexample due to the peculiar relationship between derivability ($\vdash$) and satisfaction ($\models$) in CRWL, reflected in the following results which are proved in [14]:

- If φ is a reduction or a joinability statement, $\Gamma \vdash_{\mathrm{CRWL}} \varphi$ implies that $(A, \eta) \models \varphi$, for every $A \models \Gamma$ and every *totally defined* valuation η.
- If φ is an approximation or a joinability statement, the previous implication becomes an equivalence.

This way, the soundness and completeness results are partial since they limit the type of the sentences (for completeness) as well as the type of the valuations (for both of them). Restricting the notion of satisfaction to totally defined valuations so as to strengthen the previous theorems does not seem to work, as every model of Γ must satisfy all the rules under *any* valuation.

In fact, defining $\Gamma \models \varphi$ in the usual way, that is, $\Gamma \models \varphi$ if and only if for every model $\mathcal{A}$ of Γ we have $\mathcal{A} \models \varphi$ (this definition does not appear in [14]) it turns out that the CRWL rewriting calculus is neither sound nor complete, as the following examples, in which $c, h \in F_\Sigma$, show:

- Regarding soundness,

  $$\{\, h(x) \to c \Leftarrow x \bowtie x \,\} \vdash h(x) \to c \quad \text{but} \quad \{\, h(x) \to c \Leftarrow x \bowtie x \,\} \not\models h(x) \to c.$$

  The first statement is immediate; for the second just consider the CRWL-algebra $\mathcal{A}$ with $D^{\mathcal{A}} = \{\perp, a_1, a_2\}$ and ordering $\perp \sqsubseteq a_1 \sqsubseteq a_2$, $c^{\mathcal{A}} = \langle a_2 \rangle$ and $h^{\mathcal{A}}(x) = \langle x \rangle$.
- Regarding completeness,

  $$\{\, h(x) \to x \,\} \models h(c) \to c \quad \text{but} \quad \{\, h(x) \to x \,\} \not\vdash h(c) \to c.$$

  Given a model $\mathcal{A}$ of $\{\, h(x) \to x \,\}$ and $a \in c^{\mathcal{A}}$, we have $a \in h^{\mathcal{A}}(a) \subseteq h^{\mathcal{A}}(c^{\mathcal{A}})$, and so $c^{\mathcal{A}} \subseteq h^{\mathcal{A}}(c^{\mathcal{A}})$ and $\mathcal{A} \models h(c) \to c$. On the other hand, it can be proved by induction on the derivation that if $\{\, h(x) \to x \,\} \vdash h(c) \to e$, then either $e = \perp$, or $e = h(\perp)$, or $e = h(c)$.

§6. An Entailment System for RL

We treat here the version of RL which uses unconditional, unsorted equational logic as its underlying equational logic. We can try to associate to it an entailment system and the obvious choice would be to take as **Sign** the category of equational theories; then, *sen* would assign to a signature the set of its associated conditional rewrite rules, and $\vdash$ would be given by derivation in the RL-calculus. Unfortunately, the fact that conditional rewrite rules are not derivable in the calculus of Section 4.2 causes that the reflexivity property (see page 6) that any entailment system must verify, does not hold. A way out of this problem would be to change the definition of *sen* so that only unconditional rewrite rules were associated to a signature, and this actually produces an entailment system $\mathcal{E}_{\mathrm{RL}}^{\mathrm{unc}}$. Another possibility would be to extend the original proposal, defining derivability for conditional rules. The latter will be our main task for the rest of this section.

Note that not only is derivability undefined for conditional rules, but also is satisfaction. However, we would like to rest on a natural definition of satisfaction to support the claim that our extended notion of derivability is a suitable one. Taking the definition of $\mathcal{R}$-system as guide, we say that a conditional rewrite rule

$$[t(\overline{x})] \to [t'(\overline{x})] \textit{ if } [a_1(\overline{x})] \to [b_1(\overline{x})] \wedge \ldots \wedge [a_m(\overline{x})] \to [b_m(\overline{x})],$$

is satisfied by an $\mathcal{R}$-system $\mathcal{S}$ if there exists a natural transformation

$$\alpha : t_{\mathcal{S}} \circ J_{\mathcal{S}} \Rightarrow t'_{\mathcal{S}} \circ J_{\mathcal{S}},$$

where $J_{\mathcal{S}} : \mathit{Subeq}((a_{j\mathcal{S}}, b_{j\mathcal{S}})_{1\leq j\leq m}) \to \mathcal{S}^n$. Much more about this topic will be discussed in Section 14. For the moment we will just use it to justify our definition of derivability of conditional rules.

Given an RL-theory $\mathcal{R} = (\Sigma, E, L, \Gamma)$ and a set of variables X, we define $\mathcal{R}(X) = (\Sigma(X), E, L, \Gamma')$ where $\Sigma(X)$ is the set of function symbols obtained by adding the elements of X as constants to Σ, and Γ' is obtained from Γ by renaming with fresh variables. We then have the following

6.1 Proposition *Let $\mathcal{R}$ be an RL-theory and $[t(\overline{x})] \to [t'(\overline{x})]$ if $[a_1(\overline{x})] \to [b_1(\overline{x})] \wedge \ldots \wedge [a_m(\overline{x})] \to [b_m(\overline{x})]$ a conditional rewrite rule; then, the following statements are equivalent:*

1. $\mathcal{R} \models [t(\overline{x})] \to [t'(\overline{x})] \textit{ if } [a_1(\overline{x})] \to [b_1(\overline{x})] \wedge \ldots \wedge [a_m(\overline{x})] \to [b_m(\overline{x})]$;
2. $\mathcal{R}(\overline{x}) \cup \{[a_1(\overline{x})] \to [b_1(\overline{x})], \ldots, [a_m(\overline{x})] \to [b_m(\overline{x})]\} \models [t(\overline{x})] \to [t'(\overline{x})]$;
3. $\mathcal{R}(\overline{x}) \cup \{[a_1(\overline{x})] \to [b_1(\overline{x})], \ldots, [a_m(\overline{x})] \to [b_m(\overline{x})]\} \vdash [t(\overline{x})] \to [t'(\overline{x})]$.

Proof. Statements 2 and 3 are equivalent by the soundness and completeness of the RL-calculus [21]. We will now prove that 1 implies 2 and then, that 3 implies 1.

To see that 1 ⇒ 2, let $\mathcal{S}$ be an $\mathcal{R}(\overline{x}) \cup \{[a_1(\overline{x})] \to [b_1(\overline{x})], \dots, [a_m(\overline{x})] \to [b_m(\overline{x})]\}$-system. There exist, therefore, morphisms

$$h_j : a_j(\overline{x})_{\mathcal{S}} \to b_j(\overline{x})_{\mathcal{S}} \qquad j = 1, \dots, m.$$

Since in this context $t(\overline{x})$ and $t'(\overline{x})$ (as well as all the $a_j(\overline{x})$ and $b_j(\overline{x})$) are ground terms, we only need to find a morphism $t(\overline{x})_{\mathcal{S}} \to t'(\overline{x})_{\mathcal{S}}$ to prove that $\mathcal{S} \models [t(\overline{x})] \to [t'(\overline{x})]$. For that, $\mathcal{S}$ can also be considered an $\mathcal{R}$-system and so, by hypothesis, there exists a natural transformation

$$\alpha : t_{\mathcal{S}} \circ J_{\mathcal{S}} \Rightarrow t'_{\mathcal{S}} \circ J_{\mathcal{S}},$$

where $J_{\mathcal{S}} : \mathit{Subeq}((a_{j\mathcal{S}}, b_{j\mathcal{S}})_{1\le j\le m}) \to \mathcal{S}^n$. Because of the h_j, $1 \le j \le m$, and noting that $a_j(\overline{x})_{\mathcal{S}} = a_{j\mathcal{S}}(\overline{x}_{\mathcal{S}})$ (and analogously for b_j), the interpretation $\overline{x}_{\mathcal{S}}$ of the variables $\overline{x}$ in $\mathcal{S}$ belongs to the subequalizer: $(\overline{x}_{\mathcal{S}}, \overline{h}) \in \mathit{Subeq}((a_{j\mathcal{S}}, b_{j\mathcal{S}})_{1\le j\le m})$. But then $\alpha(\overline{x}_{\mathcal{S}}, \overline{h})$ is the required morphism.

For the implication 3 ⇒ 1, given an $\mathcal{R}$-system $\mathcal{S}$ we will prove by induction on the derivation that

$$\mathcal{S} \models [t(\overline{x})] \to [t'(\overline{x})] \ \mathit{if}\ [a_1(\overline{x})] \to [b_1(\overline{x})] \wedge \dots \wedge [a_m(\overline{x})] \to [b_m(\overline{x})].$$

According to the last rule of deduction employed:

- **RF**. It must be $[t] = [t']$ and the result is immediate.
- **CG**. If
  $$\frac{[t_1] \to [t'_1] \quad \dots \quad [t_p] \to [t'_p]}{[f(t_1, \dots, t_p)] \to [f(t'_1, \dots, t'_p)]},$$
  we have, by the induction hypothesis,
  $$\mathcal{S} \models [t_i] \to [t'_i] \ \mathit{if}\ [a_1] \to [b_1] \wedge \dots \wedge [a_m] \to [b_m] \qquad 1 \le i \le p,$$
  and there exist natural transformations
  $$\alpha_i : t_{i\mathcal{S}} \circ J_{\mathcal{S}} \Rightarrow t'_{i\mathcal{S}} \circ J_{\mathcal{S}} \qquad 1 \le i \le p,$$
  where $J_{\mathcal{S}} : \mathit{Subeq}((a_{j\mathcal{S}}, b_{j\mathcal{S}})_{1\le j\le m}) \to \mathcal{S}^n$. Let $(\overline{s}, \overline{m}) \in \mathit{Subeq}((a_{j\mathcal{S}}, b_{j\mathcal{S}})_{1\le j\le m})$; if we define
  $$\alpha(\overline{s}, \overline{m}) = f_{\mathcal{S}}(\alpha_1(\overline{s}, \overline{m}), \dots, \alpha_p(\overline{s}, \overline{m})),$$
  we obtain a natural transformation $\alpha : f(t_1, \dots, t_p) \circ J_{\mathcal{S}} \Rightarrow f(t'_1, \dots, t'_n) \circ J_{\mathcal{S}}$ and the result is proved. Some warning words are in order here. In $J_{\mathcal{S}}$:

$\mathit{Subeq}((a_{j\mathcal{S}}, b_{j\mathcal{S}})_{1\leq j\leq m}) \to \mathcal{S}^n$, the n appearing as superscript depends on the actual number of variables in the sentence $[t_i] \to [t'_i]$ *if* $[a_1] \to [b_1] \wedge \ldots \wedge [a_m] \to [b_m]$ and, although the a_j and b_j are fixed, this is not the case for t_i and t'_i and so the n may vary with each i. This would imply that also the category $\mathit{Subeq}((a_{j\mathcal{S}}, b_{j\mathcal{S}})_{1\leq j\leq m})$ could vary, as its objects are pairs whose first component is an object of $\mathcal{S}^n$, and then the definition of α given above would no longer be valid. However, this is only a technical nuisance because the extra variables that t_i and t'_i may add are simply ignored by the functors $a_{j\mathcal{S}}$ and $b_{j\mathcal{S}}$, and everything could be made to fit properly by using projection functors that would preserve the natural transformations. This same remark applies to the remaining cases, too.

- **TR**. If we have
$$\frac{[t] \to [t'] \qquad [t'] \to [t'']}{[t] \to [t'']}$$
then, by induction hypothesis, there exist natural transformations
$$\alpha_1 : t_{\mathcal{S}} \circ J_{\mathcal{S}} \Rightarrow t'_{\mathcal{S}} \circ J_{\mathcal{S}} \quad \text{and} \quad \alpha_2 : t'_{\mathcal{S}} \circ J_{\mathcal{S}} \Rightarrow t''_{\mathcal{S}} \circ J_{\mathcal{S}},$$
where $J_{\mathcal{S}} : \mathit{Subeq}((a_{j\mathcal{S}}, b_{j\mathcal{S}})_{1\leq j\leq m}) \to \mathcal{S}^n$. Then, the composition $\alpha_2 \circ \alpha_1$ gives the result.

- **RP**. We will distinguish two cases:
    1. The rule employed is one of the $[a_j(\overline{x})] \to [b_j(\overline{x})]$. Since the terms are ground we must have
    $$\frac{}{[a_j(\overline{x})] \to [b_j(\overline{x})]}.$$
    But in this case, $\mathcal{S} \models [a_i] \to [b_i]$ *if* $[a_1] \to [b_1] \wedge \ldots \wedge [a_m] \to [b_m]$ follows because the own construction of the subequalizer produces a natural transformation $\alpha_j : a_{j\mathcal{S}} \circ J_{\mathcal{S}} \Rightarrow b_{j\mathcal{S}} \circ J_{\mathcal{S}}$.
    2. For some rule $[l(\overline{y})] \to [r(\overline{y})]$ *if* $[u_1(\overline{y})] \to [v_1(\overline{y})] \wedge \ldots \wedge [u_q(\overline{y})] \to [v_q(\overline{y})]$ in $\mathcal{R}$, we have
    $$\frac{\begin{array}{c}[w_1] \to [w'_1] \quad \ldots \quad [w_p] \to [w'_p] \\ [u_1(\overline{w}/\overline{y})] \to [v_1(\overline{w}/\overline{y})] \quad \ldots \quad [u_q(\overline{w}/\overline{y})] \to [v_q(\overline{w}/\overline{y})]\end{array}}{[l(\overline{w}/\overline{y})] \to [r(\overline{w'}/\overline{y})]}.$$
    By the induction hypothesis there exist natural transformations
    $$\alpha_i : w_{i\mathcal{S}} \circ J_{\mathcal{S}} \Rightarrow w'_{i\mathcal{S}} \circ J_{\mathcal{S}} \qquad 1 \leq i \leq p,$$
    and
    $$\beta_i : u_i(\overline{w})_{\mathcal{S}} \circ J_{\mathcal{S}} \Rightarrow v_i(\overline{w})_{\mathcal{S}} \circ J_{\mathcal{S}} \qquad 1 \leq i \leq q,$$

where $J_{\mathcal{S}} : \mathit{Subeq}((a_{j\mathcal{S}}, b_{j\mathcal{S}})_{1\le j\le m}) \to \mathcal{S}^n$. Since $\mathcal{S}$ is an $\mathcal{R}$-system, there also exists a natural transformation

$$\gamma : l_{\mathcal{S}} \circ J'_{\mathcal{S}} \Rightarrow r_{\mathcal{S}} \circ J'_{\mathcal{S}}$$

where $J'_{\mathcal{S}} : \mathit{Subeq}((u_{j\mathcal{S}}, v_{j\mathcal{S}})_{1\le j\le q}) \to \mathcal{S}^p$. We now need to find a natural transformation $\alpha : l(\overline{w})_{\mathcal{S}} \circ J_{\mathcal{S}} \Rightarrow r(\overline{w'})_{\mathcal{S}} \circ J_{\mathcal{S}}$. For that, let $(\overline{s}, \overline{m}) \in \mathit{Subeq}((a_{j\mathcal{S}}, b_{j\mathcal{S}})_{1\le j\le m})$; due to the morphisms $\beta_i(\overline{s}, \overline{m})$ it turns out that $(\overline{w_{\mathcal{S}}(\overline{s}), \beta(\overline{s}, \overline{m})})$ belongs to $\mathit{Subeq}((u_{j\mathcal{S}}, v_{j\mathcal{S}})_{1\le j\le q})$ and we can define

$$\alpha(\overline{s}, \overline{m}) = r_{\mathcal{S}}(\overline{\alpha(\overline{s}, \overline{m})}) \circ \gamma(\overline{w_{\mathcal{S}}(\overline{s})}, \overline{\beta(\overline{s}, \overline{m})}),$$

which finishes the proof. □

A straightforward consequence of the previous proposition is a sound and complete extension of the RL-calculus with the following rule of deduction:

5. **Implication introduction (II).**

$$\frac{\mathcal{R}(\overline{x}) \cup \{[a_1(\overline{x})] \to [b_1(\overline{x})], \ldots, [a_m(\overline{x})] \to [b_m(\overline{x})]\} \vdash [t(\overline{x})] \to [t'(\overline{x})]}{\mathcal{R} \vdash [t(\overline{x})] \to [t'(\overline{x})] \ \mathit{if}\ [a_1(\overline{x})] \to [b_1(\overline{x})] \wedge \ldots \wedge [a_m(\overline{x})] \to [b_m(\overline{x})]}.$$

We can now focus again on the main purpose of this section. For that, we associate to RL the entailment sytem $\mathcal{E}^{\mathrm{c}}_{\mathrm{RL}} = (\mathbf{Sign}, \mathit{sen}, \vdash)$ given by:

- **Sign**: the category of equational theories and theory morphisms;
- *sen*: the functor assigning to an equational theory the set of all its associated conditional rewrite rules, and mapping a theory morphism to its natural extension to rewrite rules;
- $\vdash$ is defined, for a set Γ of (Σ, E) sentences, as derivation for the corresponding RL-theory in the extended RL-calculus.

6.2 Proposition $\mathcal{E}^{\mathrm{c}}_{\mathrm{RL}} = (\mathbf{Sign}, \mathit{sen}, \vdash)$ *is an entailment system.*

Proof. The fact that composition of signature morphisms is associative (for equational logics in general, and for our unsorted and unconditional case in particular) is all that is needed to check that **Sign** is a category and *sen* a functor. Regarding the properties that $\vdash$ must satisfy:

1. reflexivity: Simply use the rule **RP** (combined with **II** for conditional rules).

2. monotonicity: Immediate by the definition of the entailment relation.

3. transitivity: Assume $\Gamma \vdash \varphi_i$ for all $i \in I$ and $\Gamma \cup \{\varphi_i \mid i \in I\} \vdash \psi$. The easiest way to prove $\Gamma \vdash \psi$ is by resorting to the soundness and completeness of the RL-calculus. Let $\mathcal{S}$ be a Γ-system, so $\mathcal{S} \models \varphi_i$ for all $i \in I$. Therefore Γ can also be considered a $\Gamma \cup \{\varphi_i \mid i \in I\}$-system and then $\mathcal{S} \models \psi$.

4. $\vdash$-translation: Suppose $\Gamma \vdash \varphi$. Given a theory morphism H, it is proved by induction on the derivation that $sen(H)(\Gamma) \vdash sen(H)(\varphi)$. The only nontrivial case is the one corresponding to **RP** and we illustrate it with an unconditional rule. If, for some $[t(\overline{x})] \to [t'(\overline{x})] \in \Gamma$, we have

$$\frac{\Gamma \vdash [w_1] \to [w_1'] \quad \dots \quad \Gamma \vdash [w_n] \to [w_n']}{\Gamma \vdash [t(\overline{w}/\overline{x})] \to [t'(\overline{w'}/\overline{x})]}$$

   then, by the induction hypothesis, $sen(H)(\Gamma) \vdash sen(H)([w_i] \to [w_i'])$ for $i = 1, \dots, n$, and, as $sen(H)([t] \to [t']) = [H(t)] \to [H(t')]$ belongs to $sen(H)(\Gamma)$, the results follows by applying **RP**. $\square$

Before finishing, it should be emphasized that throughout this section no mention at all has been made of the labels in an RL-theory. They could have been safely included within the signature part; however, they do not play any role as far as the entailment relation is concerned and, if only for ease of exposition, we have preferred to omit then. This situation will change drastically when we shift to models and try to assign an institution to RL; then, we will be forced to distinguish between labelled and unlabelled sentences, as described in Section 14.

§7. Simulating CRWL in RL

As there does not exist an entailment system associated to CRWL, it is clear that we are not going to be able to define an entailment system map between CRWL and RL. However, in this section, we show a method to simulate the behaviour of CRWL in RL. As pointed out in the last paragraph of the previous section, the set of labels of an RL-theory does not take part in the entailment process, and so it is omitted; the same convention will also be adopted in Sections 8 and 11.

Since rules in CRWL can only be instantiated with terms and not expressions, we will need to be able to distinguish between them in RL in order to apply the rules correctly. For this, variables in CRWL need to be represented in RL and we will use constants for that. Let us assume then that variables in CRWL are given by a countably infinite set $\mathcal{V} = \{v_0, v_1, \dots\}$ whereas variables in RL come from another set $\mathcal{X} = \{x_0, x_1, \ldots\}$, with $\mathcal{V} \cap \mathcal{X} = \emptyset$. Occasionally, we will also use $v \in \mathcal{V}$ and $x, y, z \in \mathcal{X}$.

A rewrite in CRWL will be simulated through a binary relation R (technically, as only functions exist, a binary function), so that $e \to e'$ in CRWL if and only if $R(e, e') \to \mathit{true}$ in RL. In a similar way, $\bowtie$ will be used to simulate joinability statements.

More precisely, let Σ be a signature with constructors; we associate to it the following RL-theory (Σ', E', Γ'):

- $\Sigma' = C_\Sigma \cup F_\Sigma \cup \{\mathit{tterm}, \mathit{pterm}, \mathit{pexpr}, \mathit{true}, \bot, \bowtie, R\} \cup \mathcal{V}$. The function symbols *tterm*, *pterm* and *pexpr* are unary and check whether their argument is a total or a partial term, or an expression; $\bot$ and the elements of $\mathcal{V}$ are constants for the bottom and the variables in CRWL; R and $\bowtie$ are binary; for $\bowtie$, infix notation will be used.
- $E' = \emptyset$.
- In Γ' we have:

$$\begin{array}{l}
\mathit{tterm}(v_i) \to \mathit{true} \quad (\forall v_i \in \mathcal{V}) \\
\mathit{tterm}(h(x_1, \ldots, x_n)) \to \mathit{true} \\
\qquad \mathit{if}\ \ \mathit{tterm}(x_1) \to \mathit{true} \wedge \ldots \wedge \mathit{tterm}(x_n) \to \mathit{true} \quad (\forall h \in C^n_\Sigma, n \in \mathbb{N}) \\
\mathit{pterm}(\bot) \to \mathit{true} \\
\mathit{pterm}(v_i) \to \mathit{true} \quad (\forall v_i \in \mathcal{V}) \\
\mathit{pterm}(h(x_1, \ldots, x_n)) \to \mathit{true} \\
\qquad \mathit{if}\ \ \mathit{pterm}(x_1) \to \mathit{true} \wedge \ldots \wedge \mathit{pterm}(x_n) \to \mathit{true} \quad (\forall h \in C^n_\Sigma, n \in \mathbb{N}) \\
\mathit{pexpr}(\bot) \to \mathit{true} \\
\mathit{pexpr}(v_i) \to \mathit{true} \quad (\forall v_i \in \mathcal{V}) \\
\mathit{pexpr}(h(x_1, \ldots, x_n)) \to \mathit{true} \\
\qquad \mathit{if}\ \ \mathit{pexpr}(x_1) \to \mathit{true} \wedge \ldots \wedge \mathit{pexpr}(x_n) \to \mathit{true} \quad (\forall h \in C^n_\Sigma \cup F^n_\Sigma, n \in \mathbb{N})
\end{array}$$

  to select terms, and

$$\begin{array}{l}
x \bowtie y \to \mathit{true}\ \mathit{if}\ \ R(x, z) \to \mathit{true} \wedge R(y, z) \to \mathit{true} \wedge \mathit{tterm}(z) \to \mathit{true} \\
R(x, \bot) \to \mathit{true}\ \mathit{if}\ \ \mathit{pexpr}(x) \to \mathit{true} \\
R(x, x) \to \mathit{true}\ \mathit{if}\ \ \mathit{pexpr}(x) \to \mathit{true} \\
R(x, y) \to \mathit{true}\ \mathit{if}\ \ R(x, z) \to \mathit{true} \wedge R(z, y) \to \mathit{true} \\
R(h(x_1, \ldots, x_n), h(y_1, \ldots, y_n)) \to \mathit{true} \\
\qquad \mathit{if}\ \ R(x_1, y_1) \to \mathit{true} \wedge \ldots \wedge R(x_n, y_n) \to \mathit{true} \quad (\forall h \in C^n_\Sigma \cup F^n_\Sigma, n \in \mathbb{N})
\end{array}$$

  corresponding to rules **J**, **B**, **RF**, **TR**, **MN** in BRC, respectively (check page 11). It is not necessary to include *pexpr* explicitly in all the rules because these conditions can be derived as logical consequences.

To every rule $l(\overline{v}) \to r(\overline{v}) \Leftarrow a_1(\overline{v}) \bowtie b_1(\overline{v}), \ldots, a_m(\overline{v}) \bowtie b_m(\overline{v})$ over Σ, we associate the following rule in RL over (Σ', E')

$$\begin{aligned} &R(l(\overline{x}), r(\overline{x})) \to \mathit{true} \\ &\quad \mathit{if}\ a_1(\overline{x}) \bowtie b_1(\overline{x}) \to \mathit{true} \wedge \ldots \wedge a_m(\overline{x}) \bowtie b_m(\overline{x}) \to \mathit{true} \wedge \\ &\qquad \mathit{pterm}(x_1) \to \mathit{true} \wedge \ldots \wedge \mathit{pterm}(x_n) \to \mathit{true}, \end{aligned}$$

where each CRWL variable v_i (a constant in the RL-theory) has been substituted by the variable x_i.

This last rule corresponds with the reduction rule **R** on page 11, and the condition that (program) rules in CRWL can only be instantiated with terms is taken care of by demanding $\mathit{pterm}(x) \to \mathit{true}$ for all the variables appearing in it. Similarly, for the joinability rule it is now *tterm* that is entrusted with picking only those expressions representing total terms.

Given a CRWL-theory $T = (\Sigma, \Gamma)$, we write $\alpha(T)$ for the RL-theory obtained by adding the translation of every rule in Γ to the theory associated to Σ. The following proposition ensures us that the translation is correct.

7.1 Proposition *Given a CRWL-theory $T = (\Sigma, \Gamma)$ such that $\alpha(T) = (\Sigma', E', \Gamma')$, then, if $l, r, a, b \in T_{\Sigma'}(\mathcal{X})$:*

$$\begin{array}{lcl} l, r \in \mathit{Expr}_\perp(\Sigma, \mathcal{V}) \text{ and } T \vdash_{\mathrm{CRWL}} l \to r & \Longleftrightarrow & \alpha(T) \vdash_{\mathrm{RL}} R(l, r) \to \mathit{true} \\ a, b \in \mathit{Expr}_\perp(\Sigma, \mathcal{V}) \text{ and } T \vdash_{\mathrm{CRWL}} a \bowtie b & \Longleftrightarrow & \alpha(T) \vdash_{\mathrm{RL}} a \bowtie b \to \mathit{true}. \end{array}$$

To prove the previous proposition we will need the following lemmas. The first one states some simple facts.

7.2 Lemma *Let $T = (\Sigma, \Gamma)$ be a CRWL-theory, $\alpha(T) = (\Sigma', E', \Gamma')$ and $e, e' \in T_{\Sigma'}(\mathcal{X})$.*

1. *If $\alpha(T) \vdash_{\mathrm{RL}} e \to e'$ and $e \in \mathit{Expr}_\perp(\Sigma, \mathcal{V})$ or $e' \in \mathit{Expr}_\perp(\Sigma, \mathcal{V})$, then $e = e'$.*
2. *If $\alpha(T) \vdash_{\mathrm{RL}} \mathit{tterm}(e) \to e'$, then either e' is true or $\mathit{tterm}(e'')$ for some e'' such that $\alpha(T) \vdash_{\mathrm{RL}} e \to e''$.*

Proof. Both facts are proved by structural induction on the derivation.

In the first case, if the last rule applied is, for example, **CG**, we have

$$\frac{e_1 \to e_1' \quad \ldots \quad e_n \to e_n'}{f(e_1, \ldots, e_n) \to f(e_1', \ldots, e_n')},$$

and by the induction hypothesis $e_i = e'_i$, $1 \leq n \leq n$, and so $f(e_1, \ldots, e_n) = f(e'_1, \ldots, e'_n)$. Note that **RP** cannot be the last rule employed.

In the second case it is necessary to prove first, also by induction, that *true* only rewrites to itself. Then, for example, if the last rule employed is **TR**,

$$\frac{\mathit{tterm}(e) \to e'' \qquad e'' \to e'}{\mathit{tterm}(e) \to e'},$$

we have by induction hypothesis that either $e'' = \mathit{true}$, in which case e' must be also *true*, or $e'' = \mathit{tterm}(e''')$ with $\alpha(T) \vdash_{\mathrm{RL}} e \to e'''$, and then the results follows by the induction hypothesis (and the transitivity of RL). □

7.3 Lemma *If* $T = (\Sigma, \Gamma)$ *is a CRWL-theory,* $\alpha(T) = (\Sigma', E', \Gamma')$ *and* $e \in T_{\Sigma'}(\mathcal{X})$*, then:*

1. $e \in \mathit{Term}(\Sigma, \mathcal{V}) \iff \alpha(T) \vdash_{\mathrm{RL}} \mathit{tterm}(e) \to \mathit{true}$,
2. $e \in \mathit{Term}_\perp(\Sigma, \mathcal{V}) \iff \alpha(T) \vdash_{\mathrm{RL}} \mathit{pterm}(e) \to \mathit{true}$,
3. $e \in \mathit{Expr}_\perp(\Sigma, \mathcal{V}) \iff \alpha(T) \vdash_{\mathrm{RL}} \mathit{pexpr}(e) \to \mathit{true}$.

Proof.

1. The $\Rightarrow$ part is immediate, by structural induction on e. For the $\Leftarrow$ part we use induction on the derivation; we have to study the last rule applied:
   - **RF**, **CG** are not possible.
   - **RP**. The only conditional rules that can be applied are the ones associated to *tterm*. For the first one there is nothing to prove, as $\mathcal{V} \subseteq \mathit{Term}(\Sigma, \mathcal{V})$; for the second, the result follows from the induction hypothesis and the definition of $\mathit{Term}(\Sigma, \mathcal{V})$.
   - **TR**. In this case we have, for some $e' \in T_{\Sigma'}(\mathcal{X})$,

     $$\frac{\mathit{tterm}(e) \to e' \qquad e' \to \mathit{true}}{\mathit{tterm}(e) \to \mathit{true}}.$$

     By Lemma 7.2, e' must be either *true* or $\mathit{tterm}(e'')$, with $\alpha(T) \vdash_{\mathrm{RL}} e \to e''$. The first case is immediate. For the second one, the induction hypothesis applied to $\alpha(T) \vdash_{\mathrm{RL}} \mathit{tterm}(e'') \to \mathit{true}$ gives us $e'' \in \mathit{Term}(\Sigma, \mathcal{V})$ and, again by Lemma 7.2, $e = e''$.
2. Facts 2 and 3 are proved in an analogous way. □

Proof of Proposition 7.1. Both directions are proved by induction on the derivation, studying the last rule applied.

For the $\Rightarrow$ part,

- **B**. We have $T \vdash_{\text{CRWL}} l \to \perp$. As $l \in \mathit{Expr}_\perp(\Sigma, \mathcal{V})$, by the previous lemma it is $\alpha(T) \vdash_{\text{RL}} \mathit{pexpr}(l) \to \mathit{true}$ so, by the first rule associated to R, we have $\alpha(T) \vdash_{\text{RL}} R(l, \perp) \to \mathit{true}$.

- **RF**. By the previous lemma, $\alpha(T) \vdash_{\text{RL}} \mathit{pexpr}(l) \to \mathit{true}$, so the result follows by applying the second rule associated to R.

- **TR**. We have
$$\frac{T \vdash_{\text{CRWL}} l \to t \qquad T \vdash_{\text{CRWL}} t \to r}{T \vdash_{\text{CRWL}} l \to r}.$$
By induction hypothesis, $\alpha(T) \vdash_{\text{RL}} R(l,t) \to \mathit{true}$ and $\alpha(T) \vdash_{\text{RL}} R(t,r) \to \mathit{true}$, and by the third rule associated to R we can derive $\alpha(T) \vdash_{\text{RL}} R(l,r) \to \mathit{true}$.

- **MN**. Similarly to **TR**.

- **J**. From
$$\frac{T \vdash_{\text{CRWL}} a \to t \qquad T \vdash_{\text{CRWL}} b \to t}{T \vdash_{\text{CRWL}} a \bowtie b} \quad t \in \mathit{Term}(\Sigma, \mathcal{V}),$$
we get, by induction hypothesis, $\alpha(T) \vdash_{\text{RL}} R(a,t) \to \mathit{true}$ and $\alpha(T) \vdash_{\text{RL}} R(b,t) \to \mathit{true}$, and by the previous lemma $\alpha(T) \vdash_{\text{RL}} \mathit{tterm}(t) \to \mathit{true}$, so we can apply the rule associated to $\bowtie$ to reach the result.

- **R**. As the rules in CRWL can only be instantiated with partial terms, it is the case that the conditions of the form $\mathit{pterm}(x_i) \to \mathit{true}$ in the translation of the CRWL-rule are satisfied; by induction hyphotesis, the others are also satisfied, so we get the result.

For the converse $\Leftarrow$,

- **RF**, **CG** are not possible.

- **TR**. Suppose
$$\frac{\alpha(T) \vdash_{\text{RL}} R(l,r) \to e \qquad \alpha(T) \vdash_{\text{RL}} e \to \mathit{true}}{\alpha(T) \vdash_{\text{RL}} R(l,r) \to \mathit{true}}.$$
(The case for $a \bowtie b$ is analogous.) By induction on the derivation of $\alpha(T) \vdash_{\text{RL}} R(l,r) \to e$ and using the fact that *true* only rewrites to itself, it follows

easily that that e must be either *true*, or $R(l',r')$ with $\alpha(T) \vdash_{\mathrm{RL}} l \to l'$ and $\alpha(T) \vdash_{\mathrm{RL}} r \to r'$. In the first case the result follows immediately; in the second, by induction hypothesis, $l',r' \in \mathit{Expr}_{\perp}(\Sigma,\mathcal{V})$ and $T \vdash_{\mathrm{CRWL}} l' \to r'$, and by Lemma 7.2 we have $l = l'$ and $r = r'$.

- **RP**. The result follows because the rules associated to R reflect faithfully the behaviour of BRC. For example, if

$$\frac{\begin{array}{c}\alpha(T) \vdash_{\mathrm{RL}} a \to a' \qquad \alpha(T) \vdash_{\mathrm{RL}} b \to b' \qquad \alpha(T) \vdash_{\mathrm{RL}} c \to c' \\ \alpha(T) \vdash_{\mathrm{RL}} R(a,c) \to \mathit{true} \qquad \alpha(T) \vdash_{\mathrm{RL}} R(b,c) \to \mathit{true} \\ \alpha(T) \vdash_{\mathrm{RL}} \mathit{tterm}(c) \to \mathit{true}\end{array}}{\alpha(T) \vdash_{\mathrm{RL}} a \bowtie b \to \mathit{true}},$$

then $c \in \mathit{Term}(\Sigma,\mathcal{V})$ by the Lemma 7.3, and $T \vdash_{\mathrm{CRWL}} a \to c$ and $T \vdash_{\mathrm{CRWL}} b \to c$ by the induction hypothesis so, applying **J**, $T \vdash_{\mathrm{CRWL}} a \bowtie b$. □

§8. Simulating RL in CRWL

We could try to simulate RL in CRWL and, for that, function symbols in RL would be mapped to constructor symbols in CRWL and unconditional rules would suffer no transformation at all. However, conditions in rewrite rules would pose a problem since conditions in CRWL are somewhat more restricted than conditions in RL. To overcome this difficulty we will represent, as in Section 7, the rewriting relation in RL through a binary relation (function) R in CRWL, so that $t \to t'$ in RL if and only if $R(t,t') \to \mathit{true}$ in CRWL. It still remains the question of rewriting modulo a set of equations. This situation will be treated by transforming each equation $t = t'$ into the rewrites $t \to t'$ and $t' \to t$.

More precisely, given a signature (Σ, E) in RL we associate to it a CRWL-theory over the signature Σ' with $C_{\Sigma'} = \Sigma \cup \{\mathit{true}\}$ and $F_{\Sigma'} = \{R\}$, with *true* and R of arities 0 and 2, respectively. Here it is not necessary to represent variables in RL as constants in CRWL, and we will use the same set $\mathcal{X}$ of variables for both logics. The rules in the theory are

$$\begin{array}{rcl} R(x_1,x_2) \to \mathit{true} & \Leftarrow & x_1 \bowtie x_2, \\ R(x,y) \to \mathit{true} & \Leftarrow & R(x,z) \bowtie \mathit{true}, R(z,y) \bowtie \mathit{true}, \end{array}$$

and, for each $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$R(f(x_1,\ldots,x_n), f(y_1,\ldots,y_n)) \to \mathit{true} \Leftarrow R(x_1,y_1) \bowtie \mathit{true}, \ldots, R(x_n,y_n) \bowtie \mathit{true},$$

mimicking the reflexivity, transitivity and congruence rules in the RL-calculus, together with

$$\begin{array}{l} R(t,t') \to \mathit{true}, \\ R(t',t) \to \mathit{true}, \end{array}$$

for every $t = t' \in E$. The goal of the condition in the rule corresponding to reflexivity is to avoid instantiating it with terms containing $\perp$, which have no meaning in RL.

A conditional rewrite rule

$$[l] \to [r] \textit{ if } [a_1] \to [b_1] \wedge \ldots \wedge [a_m] \to [b_m]$$

over (Σ, E) in RL is then translated to

$$R(l, r) \to \mathit{true} \Leftarrow R(a_1, b_1) \bowtie \mathit{true}, \ldots, R(a_m, b_m) \bowtie \mathit{true},$$

where l, r, a_i, b_i are arbitrary members of $[l]$, $[r]$, $[a_i]$ and $[b_i]$, respectively.

In fact, the previous definitions must be slightly modified due to some technical details. In a conditional rewrite rule $l \to r \Leftarrow C$ in CRWL, l must be linear, and it is obvious that with the above definitions this property is not ensured for the translation of equations and rewrite rules; therefore, those rules must be "linearised" (see [2, 1]). The linearised version of a conditional rewrite rule $l \to r \Leftarrow C$ is given by $l' \to r \Leftarrow C, C_l$, where l' and C_l are calculated as follows: For every variable x appearing $k > 1$ times in l, its j-th occurrence, $2 \leq j \leq k$ is substituted by a new variable y_j and $x \bowtie y_j$ is added to C_l. Moreover (and this is simply a characteristic of our translation), even when a variable x appears only once, $x \bowtie x$ will be added to the conditional part so that x cannot be instantiated with a partial term. The treatment of linearised rules in the rest of the section, though rigorous, will not be too formal.

In what follows, we write $\beta(T)$ for the CRWL-theory associated to a RL-theory T.

8.1 Proposition *If* (Σ', E') *is the CRWL theory corresponding to a signature* (Σ, E) *in RL and if* $t, t' \in T_\Sigma(\mathcal{X})$*, then*

$$E \vdash t = t' \quad \Longrightarrow \quad \Big(E' \vdash_{\mathrm{CRWL}} R(t, t') \to \mathit{true} \quad \textit{and} \quad E' \vdash_{\mathrm{CRWL}} R(t', t) \to \mathit{true} \Big).$$

Proof. By induction on the derivation of $E \vdash t = t'$. The rules of a deduction system for equational logic typically include rules of reflexivity, symmetry, transitivity, congruence, and substitutivity, similar to those appearing in Section 9.1. Let us just consider the case of the substitutivity rule. We have

$$\frac{}{\theta(t_1) = \theta(t_2)} \qquad (t_1 = t_2) \in E,$$

for some assignment $\theta : \mathcal{X} \to T_\Sigma(\mathcal{X})$. Associated to $t_1 = t_2$ we have the linearised versions of the two rules $R(t_1, t_2) \to \mathit{true}$ and $R(t_2, t_1) \to \mathit{true}$ in E' and, as $T_\Sigma(\mathcal{X}) \subseteq \mathit{Term}_\perp(\Sigma', \mathcal{X})$, we can instantiate them with θ (mapping those x which arose in the linearization process to the same term as the original variable) to obtain the result. □

With this in hand we are ready to prove the first half of the main proposition.

8.2 Proposition *Given any RL-theory* $T = (\Sigma, E, \Gamma)$*, and* $l, r \in T_\Sigma(\mathcal{X})$*:*

$$\begin{array}{lll} T \vdash_{\mathrm{RL}} [l] \to [r] & \Longrightarrow & (\exists l' \in [l], \exists r' \in [r])\ \ \beta(T) \vdash_{\mathrm{CRWL}} R(l', r') \to \mathit{true} \\ & \Longleftrightarrow & (\forall l' \in [l], \forall r' \in [r])\ \ \beta(T) \vdash_{\mathrm{CRWL}} R(l', r') \to \mathit{true}. \end{array}$$

Proof. Let us first prove the equivalence. There is nothing to prove from right to left; in the opposite direction the result is a consequence of the previous proposition and the $\beta(T)$-rule $R(x, y) \to \mathit{true} \Leftarrow R(x, z) \bowtie \mathit{true}, R(z, y) \bowtie \mathit{true}$. Now we prove the first implication by induction on the derivation, according to the last rule used:

- **RF**. $T \vdash_{\mathrm{RL}} [l] \to [l]$, and the result follows by instantiating $R(x_1, x_2) \to \mathit{true} \Leftarrow x_1 \bowtie x_2$ with l and l.
- **CG**. From
$$\frac{T \vdash_{\mathrm{RL}} [l_1] \to [r_1] \quad \ldots \quad T \vdash_{\mathrm{RL}} [l_n] \to [r_n]}{T \vdash_{\mathrm{RL}} [f(l_1, \ldots, l_n)] \to [f(r_1, \ldots, r_n)]}$$
and the induction hypothesis, $\beta(T) \vdash_{\mathrm{CRWL}} R(l_i', r_i') \to \mathit{true}$ for some $l_i' \in [l_i]$, $r_i' \in [r_i]$, $1 \le i \le n$. Then, using the rule $R(f(x_1, \ldots, x_n), f(y_1, \ldots, y_n)) \to \mathit{true} \Leftarrow R(x_1, y_1) \bowtie \mathit{true}, \ldots, R(x_n, y_n) \bowtie \mathit{true}$, we get $\beta(T) \vdash_{\mathrm{CRWL}} f(l_1', \ldots, l_n') \to f(r_1', \ldots, r_n')$, verifying $f(l_1', \ldots, l_n') \in [f(l_1, \ldots, l_n)]$ and $f(r_1', \ldots, r_n') \in [f(r_1, \ldots, r_n)]$.
- **TR**. From
$$\frac{T \vdash_{\mathrm{RL}} [l] \to [t] \qquad T \vdash_{\mathrm{RL}} [t] \to [r]}{T \vdash_{\mathrm{RL}} [l] \to [r]}$$
and the induction hypothesis, $\beta(T) \vdash_{\mathrm{CRWL}} l' \to t'$ and $\beta(T) \vdash_{\mathrm{CRWL}} t'' \to r'$, with $l' \in [l]$, $t', t'' \in [t]$ and $r' \in [r]$. Then, due to the equivalence just proved, $\beta(T) \vdash_{\mathrm{CRWL}} l \to t$ and $\beta(T) \vdash_{\mathrm{CRWL}} t \to r$ and we get the result using the translation of the transitivity rule.
- **RP**. We have, for some $[l(\overline{x})] \to [r(\overline{x})]$ *if* $[a_1(\overline{x})] \to [b_1(\overline{x})] \wedge \ldots \wedge [a_m(\overline{x})] \to [b_m(\overline{x})]$ in Γ,
$$\frac{\begin{array}{c} T \vdash_{\mathrm{RL}} [w_1] \to [w_1'] \quad \ldots \quad T \vdash_{\mathrm{RL}} [w_n] \to [w_n'] \\ T \vdash_{\mathrm{RL}} [a_1(\overline{w}/\overline{x})] \to [b_1(\overline{w}/\overline{x})] \quad \ldots \quad T \vdash_{\mathrm{RL}} [a_m(\overline{w}/\overline{x})] \to [b_m(\overline{w}/\overline{x})] \end{array}}{T \vdash_{\mathrm{RL}} [l(\overline{w}/\overline{x})] \to [r(\overline{w'}/\overline{x})]}.$$
By induction hypothesis, there exists $a_i' \in [a_i(\overline{w}/\overline{x})], b_i' \in [b_i(\overline{w}/\overline{x})]$ such that $\beta(T) \vdash_{\mathrm{CRWL}} R(a_i', b_i') \to \mathit{true}$ for $i = 1, \ldots, m$. Again by the equivalence, $\beta(T) \vdash_{\mathrm{CRWL}} R(a_i(\overline{w}/\overline{x}), b_i(\overline{w}/\overline{x})) \to \mathit{true}$, for $i = 1, \ldots, m$. We can then use the linearised version of $R(l, r) \to \mathit{true} \Leftarrow R(a_1, b_1) \bowtie \mathit{true}, \ldots, R(a_m, b_m) \bowtie \mathit{true}$, substituting all variables which arose from the same one during the linearisation process with the same w_i (so that the conditions $x \bowtie x$, $x \bowtie y_j$

are trivially verified), to arrive at $\beta(T) \vdash_{\mathrm{CRWL}} R(l(\overline{w}/\overline{x}), r(\overline{w}/\overline{x})) \to \mathit{true}$. In a similar way, $\beta(T) \vdash_{\mathrm{CRWL}} R(w_i, w_i') \to \mathit{true}$, $i = 1, \ldots, n$, is also obtained, and repeated aplication of the translation of the transitivity rule would show, first, that $\beta(T) \vdash_{\mathrm{CRWL}} R(r(\overline{w}/\overline{x}), r(\overline{w'}/\overline{x})) \to \mathit{true}$, and then $\beta(T) \vdash_{\mathrm{CRWL}} R(l(\overline{w}/\overline{x}), r(\overline{w'}/\overline{x})) \to \mathit{true}$, as desired. □

Our next goal will be to prove the converse of the last proposition. However, more care is needed here since, for example, an equation of the form $x * 0 = 0$ will allow us to derive $R(\mathit{true} * 0, 0) \to \mathit{true}$. Even more bizarre derivations are possible by repeated application of transitivity, like, for example, $R(R(\mathit{true}, \mathit{true}), \mathit{true}) \to \mathit{true}$. To prove that these sequents, however, do not allow us to derive anything new in CRWL, we concentrate first on some preliminary results. The next one is proved by an easy induction for each fact in the statement:

8.3 Lemma *Let $T = (\Sigma, E, \Gamma)$ be a RL-theory, $\beta(T) = (\Sigma', \Gamma')$, and $e, e_1, \ldots, e_n \in \mathit{Expr}_\perp(\Sigma', \mathcal{X})$ expressions in CRWL. Then:*

1. *If $\beta(T) \vdash_{\mathrm{CRWL}} \perp \to e$, then $e = \perp$;*
2. *For all $x \in \mathcal{X}$, if $\beta(T) \vdash_{\mathrm{CRWL}} x \to e$, then either $e = \perp$ or $e = x$;*
3. *If $\beta(T) \vdash_{\mathrm{CRWL}} R(e_1, e_2) \to e$, then either $e = \perp$, or $e = \mathit{true}$, or $e = R(e_1', e_2')$ with $\beta(T) \vdash_{\mathrm{CRWL}} e_i \to e_i'$, $i = 1, 2$;*
4. *For every $f \in C_{\Sigma'}$, if $\beta(T) \vdash_{\mathrm{CRWL}} f(e_1, \ldots, e_n) \to e$ then either $e = \perp$, or $e = f(e_1', \ldots, e_n')$ with $\beta(T) \vdash_{\mathrm{CRWL}} e_i \to e_i'$ for some $e_i' \in \mathit{Expr}_\perp(\Sigma', \mathcal{X})$, $i = 1, \ldots, n$.* □

In what follows this lemma will be used mostly without explicit reference to it: for example, when deducing $\beta(T) \vdash_{\mathrm{CRWL}} R(t, t') \to \mathit{true}$ from $\beta(T) \vdash_{\mathrm{CRWL}} R(t, t') \bowtie \mathit{true}$.

8.4 Lemma *If T is an RL-theory and $\beta(T) = (\Sigma', \Gamma')$, then:*

1. *For all $e, e' \in \mathit{Expr}_\perp(\Sigma', \mathcal{X})$, if $\beta(T) \vdash_{\mathrm{CRWL}} e \to e'$ and e' is total, then e is total;*
2. *For all $t \in \mathit{Term}_\perp(\Sigma', \mathcal{X})$, $e' \in \mathit{Expr}_\perp(\Sigma, \mathcal{X})$, if $\beta(T) \vdash_{\mathrm{CRWL}} t \to e'$ and e' is total, then $t = e'$;*
3. *For all $t, t' \in \mathit{Term}_\perp(\Sigma', \mathcal{X})$, if $\beta(T) \vdash_{\mathrm{CRWL}} t \bowtie t'$, then t is total and $t' = t$.*

Proof.

1. Induction on the last rule of the derivation. **B** and **J** are not possible; **RF** is immediate; for **TR**, if
$$\frac{e \to e'' \qquad e'' \to e'}{e \to e'}$$
then, by induction hypothesis, e'' is total, and again by induction hypothesis e is total. **MN** similarly. For **R** we have to distinguish all these cases:
   - If $R(x_1, x_2) \to \mathit{true} \Leftarrow x_1 \bowtie x_2$ has been used, then $e = R(e_1, e_2)$ and $\beta(T) \vdash_{\mathrm{CRWL}} e_1 \to t$, $\beta(T) \vdash_{\mathrm{CRWL}} e_2 \to t$ for some t, total, have been previously obtained in the derivation. By induction hypothesis, both e_1 and e_2 are total and so is e.
   - If a rule of the form $R(f(x_1, \ldots, x_n), f(y_1, \ldots, y_n)) \to \mathit{true} \Leftarrow R(x_1, y_1) \bowtie \mathit{true}, \ldots, R(x_n, y_n) \bowtie \mathit{true}$ or $R(x, y) \to \mathit{true} \Leftarrow R(x, z) \bowtie \mathit{true}, R(z, y) \bowtie \mathit{true}$ has been used, the result follows by induction hypothesis.
   - If the last rule applied has been one of those corresponding to equations or rewrite rules then $e = R(l, r)$ and a condition of the form $x \bowtie x$ or $x \bowtie y_j$ for every variable appearing in it must have been satisfied. If x has been instantiated with t, then those conditions imply that $\beta(T) \vdash_{\mathrm{CRWL}} t \to t'$ for some total t', so by induction hypothesis t is total, and so will be the expression e.
2. By 1, $t \in \mathit{Term}(\Sigma', \mathcal{X})$. By structural induction on t:
   - $t = x$, then $e' = \perp$ (absurd) or $e' = x$ and the result holds;
   - $t = f(t_1, \ldots, t_n)$, then either $e' = \perp$ (absurd) or $e' = f(e'_1, \ldots, e'_n)$ with $\beta(T) \vdash_{\mathrm{CRWL}} t_i \to e'_i$. In this last case, by induction hypothesis, $t_i = e'_i$ for $i = 1, \ldots, n$ and so $t = e'$.
3. There exists $t'' \in \mathit{Term}(\Sigma', \mathcal{X})$ with $\beta(T) \vdash_{\mathrm{CRWL}} t \to t''$ and $\beta(T) \vdash_{\mathrm{CRWL}} t' \to t''$, and by 2, $t = t'' = t'$. □

8.5 Proposition *Let* $T = (\Sigma, E, \Gamma)$ *be an RL-theory,* $\beta(T) = (\Sigma', \Gamma')$*, and let* $l, r \in \mathit{Term}_\perp(\Sigma', \mathcal{X})$*. If* $\beta(T) \vdash_{\mathrm{CRWL}} R(l, r) \to \mathit{true}$ *then* $l, r \in \mathit{Term}(\Sigma', \mathcal{X})$ *(equivalently,* $l, r \in T_{\Sigma \cup \{\mathit{true}\}}(\mathcal{X})$*) and* $(\Sigma \cup \{\mathit{true}\}, E, \Gamma) \vdash_{\mathrm{RL}} [l] \to [r]$.

Proof. By Lemma 8.4(1), $l, r \in \mathit{Term}(\Sigma', \mathcal{X})$. For the second part, we proceed by induction on the proof of $\beta(T) \vdash_{\mathrm{CRWL}} R(l, r) \to \mathit{true}$. The last rule applied must have been **TR** or **R**.

- For **TR** we have

$$\frac{\beta(T) \vdash_{\mathrm{CRWL}} R(l,r) \to e \qquad \beta(T) \vdash_{\mathrm{CRWL}} e \to \mathit{true}}{\beta(T) \vdash_{\mathrm{CRWL}} R(l,r) \to \mathit{true}}.$$

  If $e = \mathit{true}$ the result follows by induction hypothesis. Otherwise it must be $e = R(l',r')$, total by Lemma 8.4(1), with $\beta(T) \vdash_{\mathrm{CRWL}} l \to l'$ and $\beta(T) \vdash_{\mathrm{CRWL}} r \to r'$, so by Lemma 8.4(2) $l = l'$, $r = r'$ and the result holds by induction hypothesis.

- For **R** there are five different cases. Recall that rules in CRWL are instantiated with members of $\mathit{Term}_{\perp}(\Sigma', \mathcal{X})$.

  1. If

  $$\frac{\beta(T) \vdash_{\mathrm{CRWL}} l \bowtie r}{\beta(T) \vdash_{\mathrm{CRWL}} R(l,r) \to \mathit{true}},$$

     then by Lemma 8.4(3) $l = r$ so $(\Sigma \cup \{\mathit{true}\}, E, \Gamma) \vdash_{\mathrm{RL}} [l] \to [r]$.

  2. If

  $$\frac{\beta(T) \vdash_{\mathrm{CRWL}} R(l,t) \bowtie \mathit{true} \qquad \beta(T) \vdash_{\mathrm{CRWL}} R(t,r) \bowtie \mathit{true}}{\beta(T) \vdash_{\mathrm{CRWL}} R(l,r) \to \mathit{true}},$$

     we have by induction hypothesis $(\Sigma \cup \{\mathit{true}\}, E, \Gamma) \vdash_{\mathrm{RL}} [l] \to [t]$ and $(\Sigma \cup \{\mathit{true}\}, E, \Gamma) \vdash_{\mathrm{RL}} [t] \to [r]$, so $(\Sigma \cup \{\mathit{true}\}, E, \Gamma) \vdash_{\mathrm{RL}} [l] \to [r]$ by transitivity of RL.

  3. For the translation of the congruence rule the result also follows immediately by the induction hypothesis.

  4. The result is obtained using a (linearised) rule associated to an equation $t = t' \in E$. The conditions of the form $x \bowtie x$ and $x \bowtie y_j$ in the rule together with Lemma 8.4(3) imply that all the variables which arose from the same one must have been instantiated with the same element of $\mathit{Term}(\Sigma', \mathcal{X})$. This way $E \vdash l = r$, so $[l] = [r]$ and $(\Sigma \cup \{\mathit{true}\}, E, \Gamma) \vdash_{\mathrm{RL}} [l] \to [r]$ by reflexivity of RL.

  5. If the last rule applied is one of those associated to an element of Γ then, as in the previous case, all variables have been instantiated properly and the result is proved applying the induction hypothesis. □

8.6 Proposition *Given any RL-theory $T = (\Sigma, E, \Gamma)$, and $l, r \in T_{\Sigma}(\mathcal{X})$:*

$$\begin{aligned} T \vdash_{\mathrm{RL}} [l] \to [r] \quad &\iff \quad (\exists l' \in [l], \exists r' \in [r])\ \ \beta(T) \vdash_{\mathrm{CRWL}} R(l',r') \to \mathit{true} \\ &\iff \quad (\forall l' \in [l], \forall r' \in [r])\ \ \beta(T) \vdash_{\mathrm{CRWL}} R(l',r') \to \mathit{true}. \end{aligned}$$

Proof. By Propositions 8.2 and 8.5, it is enough to see that if $(\Sigma \cup \{true\}, E, \Gamma) \vdash_{\mathrm{RL}} [l] \to [r]$ then $(\Sigma, E, \Gamma) \vdash_{\mathrm{RL}} [l] \to [r]$. The easiest way of proving this implication is by using the completeness of the RL calculus.

Let $\mathcal{S}$ be a (Σ, E, Γ)-system. If $\mathcal{S}$ is the empty category, then $\mathcal{S} \models_\Sigma [l] \to [r]$ trivially. Otherwise, let us define $true_{\mathcal{S}}$ to be any object in the category. With this definition $\mathcal{S}$ is clearly a $(\Sigma \cup \{true\}, E, \Gamma)$-system, so $\mathcal{S} \models_{\Sigma\cup\{true\}} [l] \to [r]$. But then $\mathcal{S} \models_\Sigma [l] \to [r]$ also under Σ, as desired. □

§9. Introducing Types

We will now examine the versions of CRWL and RL which make use of types. For CRWL we will consider the extension presented in [2, 1], while for RL we will use membership equational logic as underlying equational logic, and refer to it as MERL from now on. Paralleling what happened in the unsorted case, an entailment system can be associated to MERL, but not to the extension of CRWL. Let us briefly review the basic characteristics of these new logics.

9.1. Membership equational logic

Membership equational logic is an expressive version of equational logic. A full treatment of its syntax and semantics can be found in [22, 5]. Here we define only those notions that will be needed later on, borrowing from [22].

A signature in membership equational logic is a triple $\Omega = (K, \Sigma, S)$ with K a set of *kinds*, Σ a K-kinded signature and $S = \{S_k\}_{k\in K}$ a pairwise disjoint K-kinded family of sets. We call S_k the set of *sorts* of kind k. The pair (K, Σ) is what is usually called a many-sorted signature of function symbols; however, we call the elements of K kinds because each kind k now has a set S_k of associated sorts. Also, we denote by $T_\Sigma(\mathcal{X})_k$ the set of terms of kind k with variables in a set $\mathcal{X}$.

The atomic formulae of membership equational logic are either equations $t = t'$, where t and t' are Σ-terms of the same kind, or *membership assertions* of the form $t : s$, where the term t has kind k and $s \in S_k$. Sentences are Horn clauses on these atomic formulas, that is, sentences of the form

i. $(\forall \mathcal{X})\ t = t' \Leftarrow a_1 = b_1 \wedge \ldots \wedge a_m = b_m \wedge w_1 : s_1 \wedge \ldots \wedge w_p : s_p$,

ii. $(\forall \mathcal{X})\ t : s \Leftarrow a_1 = b_1 \wedge \ldots \wedge a_m = b_m \wedge w_1 : s_1 \wedge \ldots \wedge w_p : s_p$, where $\mathcal{X}$ is a K-kinded set containing all the variables appearing in t, t' (resp. t) and the a_i, b_i and w_i.

The semantics of membership equational logic can be defined in a similar way to that of many-sorted logic; now, in addition, the sorts will be interpreted in the models as subsets of the carrier for the kind. As far as we are concerned, however, we only need to know that given a theory (Ω, E), the following set of rules of deduction is sound and complete (see [22]).

1. **Reflexivity**. For each $t \in T_\Sigma(\mathcal{X})$,

$$\frac{}{E \vdash (\forall \mathcal{X})\ t = t}.$$

2. **Symmetry**.

$$\frac{E \vdash (\forall \mathcal{X})\ t = t'}{E \vdash (\forall \mathcal{X})\ t' = t}.$$

3. **Transitivity**.

$$\frac{E \vdash (\forall \mathcal{X})\ t = t' \qquad E \vdash (\forall \mathcal{X})\ t' = t''}{E \vdash (\forall \mathcal{X})\ t = t''}.$$

4. **Congruence**. For each $f : k_1 \dots k_n \to k$ in Σ and terms $t_i, t'_i \in T_\Sigma(\mathcal{X})_{k_i}$, $1 \leq i \leq n$,

$$\frac{E \vdash (\forall \mathcal{X})\ t_1 = t'_1 \quad \dots \quad E \vdash (\forall \mathcal{X})\ t_n = t'_n}{E \vdash (\forall \mathcal{X})\ f(t_1, \dots, t_n) = f(t'_1, \dots, t'_n)}.$$

5. **Membership**.

$$\frac{E \vdash (\forall \mathcal{X})\ t = t' \qquad E \vdash (\forall \mathcal{X})\ t : s}{E \vdash (\forall \mathcal{X})\ t' : s}.$$

6. **Modus ponens**. Given a sentence

$$(\forall \mathcal{X})\ t = t' \Leftarrow a_1 = b_1 \wedge \dots \wedge a_m = b_m \wedge w_1 : s_1 \wedge \dots \wedge w_p : s_p$$
$$(\text{resp. } (\forall \mathcal{X})\ t : s \Leftarrow a_1 = b_1 \wedge \dots \wedge a_m = b_m \wedge w_1 : s_1 \wedge \dots \wedge w_p : s_p)$$

in the set E of axioms, and given a K-kinded assignment $\theta : \mathcal{X} \to T_\Sigma(\mathcal{Y})$, then

$$\frac{E \vdash (\forall \mathcal{Y})\ \theta(a_i) = \theta(b_i) \quad 1 \leq i \leq m \qquad E \vdash (\forall \mathcal{Y})\ \theta(w_j) : s_j \quad 1 \leq j \leq p}{E \vdash (\forall \mathcal{Y})\ \theta(t) = \theta(t') \quad (\text{resp. } (\forall \mathcal{Y})\ \theta(t) : s)}.$$

9.2. Algebraic CRWL

The extension of CRWL defined in [2, 1] adds polymorphic types to the original version, as well as algebraic datatypes; it is called Algebraic Constructor-based ReWriting Logic and denoted by ACRWL. (Note that there also exists an extension of CRWL with higher-order functional types, described in [15], which is not considered here.) For the following presentation we borrow from [3, 2].

Let $TC = \bigcup_{n \geq 0} TC^n$ be a countable ranked alphabet of *type constructors*, and TV a countable set of *type variables* $\alpha, \beta, \ldots$ *Polymorphic types* $\tau \in T_{TC}(TV)$ are built as $\tau ::= \alpha \mid T(\tau, \ldots, \tau)$, where $T \in TC^n$ and $\alpha \in TV$. A *polymorphic signature* Σ is a triple $\langle TC_\Sigma, C_\Sigma, F_\Sigma \rangle$, where TC_Σ is a set of type constructors, and C_Σ and F_Σ are sets of type declarations $h : (\tau_1, \ldots, \tau_n) \rightarrow \tau_0$ for *data constructors* and *defined function symbols*, respectively. In addition, no overloading is allowed in $C_\Sigma \cup F_\Sigma$, and elements of C_Σ satisfy the following *transparency* property: all type variables in $\tau_1, \ldots, \tau_n$ also appear in τ_0. The types given by the declarations in $C_\Sigma \cup F_\Sigma$ are called *principal types.* We write $\Sigma_\perp$ for the signature obtained from Σ by adding to it the constructor $\perp :\rightarrow \alpha$.

Assuming another countably infinite set of *data variables* $\mathcal{X}$ we build *partial expressions* $e \in Expr_\perp(\Sigma, \mathcal{X})$ as $e ::= x \mid h(e, \ldots, e)$, where $h : (\tau_1, \ldots, \tau_n) \rightarrow \tau_0 \in C_{\Sigma_\perp} \cup F_\Sigma$ and $x \in \mathcal{X}$. The set of *partial terms* $Term_\perp(\Sigma, \mathcal{X})$ is obtained by using only symbols in $C_{\Sigma_\perp}$. Analogously, removing the constructor $\perp :\rightarrow \alpha$ the sets of total expressions and terms are obtained. A *data substitution* is a function mapping data variables to partial terms. A data substitution δ is *safe* for a term $s \in Term(\Sigma, \mathcal{X})$ if $\delta(x)$ is total for every variable x appearing more than once in s.

An *environment* is defined as a set V of type-annotated data variables $x : \tau$, such that V does not include two different annotations for the same variable. The following rules of inference are used to determine whether an expression has a type with respect to V:

$$\frac{}{V \vdash x : \tau} \quad \text{if } x : \tau \in V \qquad \text{and} \qquad \frac{V \vdash e_1 : \tau_1 \quad \ldots \quad V \vdash e_n : \tau_n}{V \vdash h(e_1, \ldots, e_n) : \tau_0},$$

where $h : (\tau_1, \ldots, \tau_n) \rightarrow \tau_0$ is an instance of the principal type of h in the signature. A partial expression e is said to have type τ with respect to V if $V \vdash e : \tau$.

An *equational axiom* is a sentence of the form $s \approx t$, where s and t are total terms. If s and t share the same variables, $s \approx t$ is said to be *regular*; if, in addition, $s, t \notin \mathcal{X}$, the equational axiom is *strongly regular.* A strongly regular equational axiom $s \approx t$ is *well-typed* if there exists an environment V such that $V \vdash s : \tau$ and $V \vdash t : \tau$ for some type τ.

Program rules are constructor-based rewrite rules for defined functions. More precisely, assuming a principal type declaration $f : (\tau_1, \ldots, \tau_n) \to \tau_0 \in F_\Sigma$, a *defining rule* for f must have the form: $f(t_1, \ldots, t_n) \to r \Leftarrow a_1 \bowtie b_1, \ldots, a_m \bowtie b_m$, where the left-hand side is *linear* (i.e., without multiple occurrences of variables), $t_i \in \mathit{Term}(\Sigma, \mathcal{X})$, $1 \le i \le n$, $r, a_i, b_i \in \mathit{Expr}(\Sigma, \mathcal{X})$, $1 \le i \le m$. The rule is said to be *regular* if all variables in r also appear in $f(t_1, \ldots, t_n)$. Furthermore, the rule is *well-typed* if there is some environment V such that $V \vdash t_i : \tau_i$, $1 \le i \le n$, $V \vdash r : \tau_0$ and, for all $a_i \bowtie b_i$, $1 \le i \le m$, there exists some τ_i' such that $V \vdash a_i : \tau_i'$ and $V \vdash b_i : \tau_i'$.

An *ACRWL-program* (or theory) $\mathcal{P}$ is a triple $\mathcal{P} = \langle \Sigma, C, \Gamma \rangle$, where Σ is a polymorphic signature, C a finite set of equational axioms, and Γ a finite set of program rules. A program is said to be *regular* if all equational axioms in C are strongly regular and all the rules in Γ are regular. A regular program is *well-typed* if all the equational axioms in C and all the rules in Γ are so.

From a given program $\mathcal{P}$ two kinds of sentences will be derived: *reduction statements* $e \to e'$, and *joinability statements* $e \bowtie e'$, with $e, e' \in \mathit{Expr}_\perp(\Sigma, \mathcal{X})$. For that, the two calculi defined for CRWL, BRC and GORC, are extended with a new rule dealing with equational axioms. The BRC in this new setting consists of the six rules of Section 3 plus the following one:

7. **Mutation (MUT).** For each $s \sqsupseteq t \in [C]_\sqsupseteq$,

$$\frac{}{s \to t},$$

where $[C]_\sqsupseteq = \{s\delta \sqsupseteq t\delta,\ t\delta' \sqsupseteq s\delta' \mid s \approx t \in C, \delta, \delta'$ safe substitutions for s and t, respectively$\}$.

We say that $\mathcal{P}$ entails $e \to e'$ and write $\mathcal{P} \vdash_{\mathrm{ACRWL}} e \to e'$ if $e \to e'$ can be obtained by finite application of the BRC-rules.

A *Polymorphically Typed algebra* (PT-algebra) $\mathcal{A}$ is given by a 5-tuple $(D^\mathcal{A}, U^\mathcal{A}, :^\mathcal{A}, \{T^\mathcal{A}\}_{T \in TC_\Sigma}, \{c^\mathcal{A}\}_{c \in C_\Sigma}, \{f^\mathcal{A}\}_{f \in F_\Sigma})$ where $D^\mathcal{A}$ is a poset, $U^\mathcal{A}$ is a set (called the universe of types), $:^\mathcal{A} \subseteq D^\mathcal{A} \times U^\mathcal{A}$ is a relation such that $\mathcal{E}(l)^\mathcal{A} = \{d \in D^\mathcal{A} \mid d :^\mathcal{A} l\}$ is a cone for every $l \in U^\mathcal{A}$, $T^\mathcal{A} : (U^\mathcal{A})^n \to U^\mathcal{A}$ is a function for each $T \in TC$ of arity n, and $c^\mathcal{A}$ and $f^\mathcal{A}$ are monotone mappings from $(D^\mathcal{A})^n$ to $\mathcal{C}(D^\mathcal{A})$, with n the arity of the corresponding symbol. In addition, for $c^\mathcal{A}$ it is also required that for any $d_1, \ldots, d_n \in D^\mathcal{A}$ there exists $d \in D^\mathcal{A}$ such that $c^\mathcal{A}(d_1, \ldots, d_n) = \langle d \rangle$; moreover, $d \in \mathit{Def}(D^\mathcal{A})$ in case all $d_i \in \mathit{Def}(D^\mathcal{A})$.

A *type valuation* μ is a function from TV to $U^\mathcal{A}$. A *data valuation* η is a function from $\mathcal{X}$ to $D^\mathcal{A}$; η is *safe* for $t \in \mathit{Term}_\perp(\Sigma, \mathcal{X})$ if $\eta(x) \in \mathit{Def}(D^\mathcal{A})$ for every x appearing more than once in t. A *valuation* is a pair (μ, η) of type and data valuations. With these definitions, denotations of types and expressions are obtained as usual.

A PT-algebra $\mathcal{A}$ is *well-typed* if it holds that $h^{\mathcal{A}}(d_1, \ldots, d_n) \subseteq \mathcal{E}^{\mathcal{A}}([\![\tau_0]\!]^{\mathcal{A}}\mu)$ for all $h : (\tau_1, \ldots, \tau_n) \to \tau_0 \in \Sigma$, every type valuation μ, and any $d_i \in \mathcal{E}^{\mathcal{A}}([\![\tau_i]\!]^{\mathcal{A}}\mu)$, $1 \leq i \leq n$.

Satisfaction of conditional rewrite rules is defined in the same way as in CRWL. For equational axioms, a PT-algebra $\mathcal{A}$ satisfies $s \approx t$ if, for every valuation $\xi = (\mu, \eta)$, $[\![s]\!]^{\mathcal{A}}\eta \supseteq [\![t]\!]^{\mathcal{A}}\eta$ if η is safe for s, and $[\![t]\!]^{\mathcal{A}}\eta \supseteq [\![s]\!]^{\mathcal{A}}\eta$ if η is safe for t. $\mathcal{A}$ is a model of an ACRWL-program if it satisfies all its equational axioms and program rules.

A *PT-homomorphism* $F : \mathcal{A} \to \mathcal{B}$ is given by a pair of mappings (F_t, F_d), where $F_t : U^{\mathcal{A}} \to U^{\mathcal{B}}$, and $F_d : D^{\mathcal{A}} \to \mathcal{C}(D^{\mathcal{B}})$ is monotone and verifies the same conditions as CRWL-homomorphisms. In addition, $F_t(T^{\mathcal{A}}(l_1, \ldots, l_n)) = T^{\mathcal{B}}(F_t(l_1), \ldots, F_t(l_n))$ for all $T \in TC_{\Sigma}$ of arity n, and $l_1, \ldots, l_n \in U^{\mathcal{A}}$. A PT-homomorphism $F : \mathcal{A} \to \mathcal{B}$ is *well-typed* if $F_d(\mathcal{E}^{\mathcal{A}}(l)) \subseteq \mathcal{E}^{\mathcal{B}}(F_t(l))$ for all $l \in U^{\mathcal{A}}$.

§10. Simulating ACRWL

In this section we look back at the results of Sections 7 and 8 and study whether they continue holding when ACRWL is substituted for CRWL.

First, we try to simulate the behaviour of ACRWL within RL. Now that we have got types in CRWL the first idea would be to try to use MERL in order to carry out the translation. However, a close look at its rules reveals that types play no role in the BRC. It is proved in [2] that, for a regular and well-typed ACRWL-program $\mathcal{P}$, if $\mathcal{P} \vdash e \to e'$ and $V \vdash e : \tau$ for some environment V and type τ, then $V \vdash e' : \tau$. But this result is not guaranteed for programs not verifying these conditions and, even for those which do, there is nothing to prevent us from using BRC to derive a statement $e \to e'$ with e an ill-typed expression. Therefore, the simulation will still take place within the basic framework of unsorted RL.

Reflecting the fact that the new rewriting calculus is obtained from the original one by simply adding a new rule of inference, the translation remains *exactly* the same as the one we defined in Section 7 (and the proof of its soundness would go along the same lines) except that now, for each equational axiom $s \approx t$, two new conditional rewrite rules, modelling the behaviour of **MUT**, have to be added:

$$\begin{array}{l} R(s(\overline{x}, \overline{y}), t(\overline{x}, \overline{y})) \to \mathit{true} \ \mathit{if} \ \ \mathit{pterm}(\overline{x}) \to \mathit{true} \wedge \mathit{tterm}(\overline{y}) \to \mathit{true}, \\ R(t(\overline{x'}, \overline{y'}), s(\overline{x'}, \overline{y'})) \to \mathit{true} \ \mathit{if} \ \ \mathit{pterm}(\overline{x'}) \to \mathit{true} \wedge \mathit{tterm}(\overline{y'}) \to \mathit{true}, \end{array}$$

where $\overline{y}$ and $\overline{y'}$ are the variables appearing more than once in s and in t, respectively. (So that safe substitutions are taken into account.)

What about the other way around, that is, simulating RL within ACRWL? Obviously, we can simply ignore the new characteristics and employ the pattern described in Section 8 to obtain a correct translation. But one could also think of taking advantage of equational axioms in order to simulate the equational part of an RL-theory. More precisely, an equation $t = t'$ would be transformed, instead of into two rewrite rules $R(t,t') \to \mathit{true}$ and $R(t',t) \to \mathit{true}$, into a single equational axiom $t \approx t'$. This way, given an equational theory E, and E' the ACRWL-program for its associated set of equational axioms, we would have

$$E \vdash t = t' \quad \Longrightarrow \quad E' \vdash t \to t' \text{ and } E' \vdash t' \to t,$$

and this result could be used to prove, in much the same way as in Section 8, that for T an RL-theory and T' its associated ACRWL-program,

$$T \vdash [t] \to [t'] \quad \Longrightarrow \quad T' \vdash R(t,t') \to \mathit{true}.$$

We conjecture that the converse implication also holds, but have not been able to prove it.

§11. Simulating MERL in CRWL

As in Section 8, a binary function symbol R will be used to simulate the rewriting relation in CRWL: the goal in mind is having $t \to t'$ in MERL if and only if $R(t,t') \to \mathit{true}$ in CRWL. Now, in addition, a binary function symbol I will be needed to represent equality in membership equational logic ($t = t' \iff I(t,t') \to \mathit{true}$), as well as another one M to simulate memberships ($t : s \iff M(t,s) \to \mathit{true}$).

A further distinction concerns the treatment of variables. CRWL has no types so, given the function symbols of a membership equational signature, we could use them to build more terms than those that we would have in the original logic; e.g., $\mathit{nil} + 0$, assuming $0 : \mathit{nat}$ and $\mathit{nil} : \mathit{list}$ in the signature. But the use of such terms would enlarge the set of provable statements in a theory. For example, assume a given signature with two kinds k_1, k_2, an operation $f : k_1 \to k_1$ and two constants $a, b : k_2$ (sorts play no role): Given the equations $(\forall x, y)\ f(x) = f(y)$ and $(\forall x, y)\ x = y \Leftarrow f(x) = f(y)$, where x, y are k_1-kinded variables, disregarding well-typed terms we could deduce $a = b$. So it will be necessary to distinguish somehow the well-typed terms and, for that, we are forced again to represent variables in MERL as constants in CRWL. The task of recognising well-typed terms will be carried out by a binary function symbol wtterm in such a way that $\mathit{wtterm}(t,k) \to \mathit{true}$ if and only if the term t has kind k. For the rest of the section we will use $\mathcal{V}$ for the set of variables in MERL (it will be a K-kinded set for some set of kinds K) and $\mathcal{X}$ for the variables in CRWL.

Let then $((K,\Sigma,S),E)$ be a MERL signature, i.e., a membership equational logic theory; without loss of generality we can assume that the signature is not ambiguous, so that each term has a unique kind. We associate to it a CRWL-theory with signature Σ' such that $C_{\Sigma'} = \Sigma \cup \{\mathit{true}\} \cup K \cup S \cup \mathcal{V}$ and $F_{\Sigma'} = \{\mathit{wtterm}, R, I, M\}$. For the sake of conceptual clarity and in order to avoid excessive repetition we will write $\overline{\mathit{wtterm}(x,k) \bowtie \mathit{true}}$ for $\mathit{wtterm}(x_1,k_1) \bowtie \mathit{true}, \ldots, \mathit{wtterm}(x_n,k_n) \bowtie \mathit{true}$, and analogously using R and I instead of *wtterm*. The first set of rules are used to select the well-typed terms:

$$\begin{array}{rcll}
\mathit{wtterm}(v,k) \to \mathit{true} & & (\forall v \in \mathcal{V} \text{ of kind } k \in K), & \\
\mathit{wtterm}(c,k) \to \mathit{true} & & (\forall c : k \in \Sigma), & \\
\mathit{wtterm}(f(\overline{x}),k) \to \mathit{true} & \Leftarrow & \overline{\mathit{wtterm}(x,k) \bowtie \mathit{true}} & (\forall f : k_1 \ldots k_n \to k \in \Sigma).
\end{array}$$

The next set of rules are used to simulate the equational calculus:

$$\begin{array}{rcl}
I(x_1,x_2) \to \mathit{true} & \Leftarrow & x_1 \bowtie x_2, \mathit{wtterm}(x_1,y_1) \bowtie \mathit{true}, \\
I(x,y) \to \mathit{true} & \Leftarrow & I(x,z) \bowtie \mathit{true}, I(z,y) \bowtie \mathit{true}, \\
I(x,y) \to \mathit{true} & \Leftarrow & I(y,x) \bowtie \mathit{true}, \\
I(f(\overline{x}),f(\overline{y})) \to \mathit{true} & \Leftarrow & \overline{I(x,y) \bowtie \mathit{true}}, \overline{\mathit{wtterm}(x,k) \bowtie \mathit{true}}, \\
 & & (\forall f : k_1 \ldots k_n \to k) \\
M(y,z) \to \mathit{true} & \Leftarrow & I(x,y) \bowtie \mathit{true}, M(x,z) \bowtie \mathit{true}.
\end{array}$$

In addition, for every equation

$$\begin{array}{rcl}
(\forall \mathcal{W})\ t(\overline{v}) = t'(\overline{v}) & \Leftarrow & a_1(\overline{v}) = b_1(\overline{v}) \wedge \ldots \wedge a_m(\overline{v}) = b_m(\overline{v}) \wedge \\
 & & w_1(\overline{v}) : s_1 \wedge \ldots \wedge w_p(\overline{v}) : s_p
\end{array}$$

in E, with $\mathcal{W} \subseteq \mathcal{V}$ containing all variables in $\overline{v}$, the following rule is added

$$\begin{array}{rcl}
I(t(\overline{x}),t'(\overline{x})) \to \mathit{true} & \Leftarrow & I(a_1(\overline{x}),b_1(\overline{x})) \bowtie \mathit{true}, \ldots, I(a_m(\overline{x}),b_m(\overline{x})) \bowtie \mathit{true}, \\
 & & M(w_1(\overline{x}),s_1) \bowtie \mathit{true}, \ldots, M(w_p(\overline{x}),s_p) \bowtie \mathit{true}, \\
 & & \overline{\mathit{wtterm}(x,k) \bowtie \mathit{true}},
\end{array}$$

where k_i is the kind of v_i; and similarly (replacing I with M) for memberships. The fact that equations can now be conditional forces us to use a symbol different from R to simulate the equational logic. Otherwise, the effect that we would actually obtain is the weakening of those conditions because it would be enough for a term to rewrite to another, instead of being equationally equal to it, to satisfy them.

One could think that there are some conditions of the form $\mathit{wtterm}(x,k) \bowtie \mathit{true}$ missing in some of the above rules; for example, $\overline{\mathit{wtterm}(y,k) \bowtie \mathit{true}}$ in the translation of the monotonicity rule. However, all these conditions follow as logical consequences of the rules *and* our assumption that no term has two different kinds.

Regarding the rewriting logic part, we have the three rules we saw in Section 8,

$$\begin{array}{rcl} R(x_1,x_2) \to true & \Leftarrow & x_1 \bowtie x_2, wtterm(x_1,y_1) \bowtie true \\ R(x,y) \to true & \Leftarrow & R(x,z) \bowtie true, R(z,y) \bowtie true, \\ R(f(\overline{x}),f(\overline{y})) \to true & \Leftarrow & R(x,y) \bowtie true, wtterm(x,k) \bowtie true, \\ & & (\forall f : k_1 \ldots k_n \to k) \end{array}$$

and two more

$$\begin{array}{rcl} R(x,y) \to true & \Leftarrow & R(x',y) \bowtie true, I(x,x') \bowtie true, \\ R(x,y) \to true & \Leftarrow & R(x,y') \bowtie true, I(y,y') \bowtie true. \end{array}$$

Now, the translation of a rewrite rule

$$[l(\overline{v})] \to [r(\overline{v})] \ \mathit{if} \ [a_1(\overline{v})] \to [b_1(\overline{v})] \wedge \ldots \wedge [a_m(\overline{v})] \to [b_m(\overline{v})]$$

is

$$\begin{array}{rcl} R(l(\overline{x}),r(\overline{x'})) \to true & \Leftarrow & R(a_1(\overline{x}),b_1(\overline{x})) \bowtie true, \ldots, R(a_m(\overline{x}),b_m(\overline{x})) \bowtie true, \\ & & wtterm(x,k) \bowtie true, wtterm(x',k) \bowtie true, \\ & & R(x,x') \bowtie true, \end{array}$$

with k_i the kind of v_i. Besides renaming of variables there is another little difference between this translation and the one used in Section 8: the conditions $R(x,x') \bowtie true$. The choice between either alternative is mainly a matter of taste, although the one used here reflects more accurately the rule **RP** of the RL calculus and simplifies a little bit one case in one of the proofs below.

As in Section 8, the conditional rewrite rules corresponding to equations and rewrite rules must be linearised. Here, however, it is not necessary to add a condition $x \bowtie x$ for those variables appearing just once, because we have already a condition $wtterm(x,y)$.

In what follows, we will write $\delta(T)$ for the CRWL-theory associated to a MERL-theory T. We begin by stating some simple results, very much like those of Section 8.

11.1 Lemma *Let T be an RL-theory, $\delta(T) = (\Sigma', \Gamma')$, and let $e, e_1, \ldots, e_n$ be expressions in $Expr_\perp(\Sigma, \mathcal{X})$. Then:*

1. *If $\delta(T) \vdash_{\mathrm{CRWL}} \perp \to e$, then $e = \perp$;*
2. *For every $c \in C^0_{\Sigma'}$, if $\delta(T) \vdash_{\mathrm{CRWL}} c \to e$, then either $e = \perp$ or $e = c$;*
3. *For every $f \in C^n_{\Sigma'}$, $n > 0$, if $\delta(T) \vdash_{\mathrm{CRWL}} f(e_1, \ldots, e_n) \to e$ then either $e = \perp$ or $e = f(e'_1, \ldots, e'_n)$ with $\delta(T) \vdash_{\mathrm{CRWL}} e_i \to e'_i$ for some $e'_i \in Expr_\perp(\Sigma', \mathcal{X})$, $i = 1, \ldots, n$;*

4. If $\delta(T) \vdash_{\text{CRWL}} \mathrm{Q}(e_1, e_2) \to e$ *with* $\mathrm{Q} \in \{R, I, M, \mathit{wtterm}\}$*, then either* $e = \bot$*, or* $e = \mathit{true}$*, or* $e = \mathrm{Q}(e'_1, e'_2)$ *with* $\delta(T) \vdash_{\text{CRWL}} e_i \to e'_i$ *for some* $e'_i \in \mathit{Expr}_\bot(\Sigma', \mathcal{X})$*,* $i = 1, 2$*.* □

11.2 Lemma *Let* T *be an RL-theory with signature* $((K, \Sigma, S), E)$*,* $\delta(T) = (\Sigma', \Gamma')$*, and let* $e, e' \in \mathit{Expr}_\bot(\Sigma', \mathcal{X})$*.*

1. *If* $\delta(T) \vdash_{\text{CRWL}} e \to e'$ *and* $e' \in T_\Sigma(\mathcal{V}) \cup K \cup S$ *then* $e = e'$*;*
2. *If* $\delta(T) \vdash_{\text{CRWL}} e \to e'$*,* $e \in T_\Sigma(\mathcal{V})$ *and* e' *is total, then* $e = e'$*;*
3. *If* $\delta(T) \vdash_{\text{CRWL}} e \bowtie e'$ *and* $e \in T_\Sigma(\mathcal{V})$ *or* $e' \in T_\Sigma(\mathcal{V})$ *then* $e = e'$*;*
4. $\delta(T) \vdash_{\text{CRWL}} \mathit{wtterm}(e, e') \to \mathit{true} \iff e' \in K$ *and* $e \in T_\Sigma(\mathcal{V})_{e'}$*.*

Proof.

1. Let us study the last rule in the derivation of $\delta(T) \vdash_{\text{CRWL}} e \to e'$.
   - **B** and **J** are not possible.
   - **RF**. Trivial.
   - **MN**. For constants or members of $\mathcal{V}, K$ or S it is immediate. For n-ary functions with $n > 0$, apply induction hypothesis.
   - **TR**. Apply induction hypothesis twice.
   - **R**. It is not possible: e' would be *true*.
2. Structural induction on e. If $e = v_i$ then $e' = \bot$ or $e' = v_i$ and the first option is not possible because e' is total. If $e = f(e_1, \ldots, e_n)$ then $e' = \bot$ or $e' = f(e'_1, \ldots, e'_n)$ with $\delta(T) \vdash_{\text{CRWL}} e_i \to e'_i$ for some $e'_i \in \mathit{Expr}_\bot(\Sigma', \mathcal{X})$, $i = 1, \ldots, n$: the first option is not possible and, as each e_i must be total, the result follows by induction hypothesis.
3. It follows from 1 and 2.
4. The implication from right to left is proved by structural induction on e. If $e = f(e_1, \ldots, e_n)$ with $f : k_1 \ldots k_n \to k \in \Sigma$ and $e_i \in T_\Sigma(\mathcal{V})_{k_i}$, then we have $\delta(T) \vdash_{\text{CRWL}} \mathit{wtterm}(e_i, k_i) \to \mathit{true}$, $1 \leq i \leq n$ by induction hypothesis, and the result follows by applying the rule $\mathit{wtterm}(f(x_1, \ldots, x_n)) \to \mathit{true} \Leftarrow \mathit{wtterm}(x_1, k_1) \bowtie \mathit{true}, \ldots, \mathit{wtterm}(x_n, k_n) \bowtie \mathit{true}$ associated to f. The other direction is proved by structural induction on the derivation. There are only two possibilities for the last rule applied: either **R** or **TR**. For the first case

the result is an easy consequence of the induction hypothesis. Otherwise we have

$$\frac{\mathit{wtterm}(e,e') \to e'' \qquad e'' \to \mathit{true}}{\mathit{wtterm}(e,e') \to \mathit{true}}$$

for some e'' that must be $\perp$, *true* or $\mathit{wtterm}(e_1,e'_1)$ with $\delta(T) \vdash_{\mathrm{CRWL}} e \to e_1$ and $\delta(T) \vdash_{\mathrm{CRWL}} e' \to e'_1$. The first option is not possible. If it is the second, the induction hypothesis provides the result. In the third case, we have $e'_1 \in K$ and $e_1 \in T_\Sigma(\mathcal{V})_{e'_1}$ by induction hypothesis, and by 1, $e = e_1$, $e' = e'_1$. □

11.3 Proposition *Let T be an RL-theory with signature $((K,\Sigma,S),E)$, $\delta(T) = (\Sigma',\Gamma')$, and let $t,t',s \in \mathit{Expr}_\perp(\Sigma',\mathcal{X})$:*

$$\begin{array}{rcl}\delta(T) \vdash_{\mathrm{CRWL}} I(t,t') \to \mathit{true} & \iff & (\exists k \in K)\; t,t' \in T_\Sigma(\mathcal{V})_k \;\mathit{and}\; E \vdash (\forall \mathcal{V})\; t = t' \\ \delta(T) \vdash_{\mathrm{CRWL}} M(t,s) \to \mathit{true} & \iff & (\exists k \in K)\; t \in T_\Sigma(\mathcal{V})_k, s \in S_k \;\mathit{and}\; E \vdash (\forall \mathcal{V})\; t : s\end{array}$$

Proof. Let us prove the left to right implication, by induction on the derivation. The last rule must have been **TR** or **R**. In the first case it is

$$\frac{I(t,t') \to e \qquad e \to \mathit{true}}{I(t,t') \to \mathit{true}},$$

with e equal to $\perp$, *true*, or $I(t_1,t'_1)$ for some t_1,t'_1 such that $\delta(T) \vdash_{\mathrm{CRWL}} t \to t_1$ and $\delta(T) \vdash_{\mathrm{CRWL}} t' \to t'_1$. The first case is not possible; for the second, the induction hypothesis immediately gives the result. For the third one, by induction hypothesis we have $t_1,t'_1 \in T_\Sigma(\mathcal{V})_k$ for some $k \in K$ and $E \vdash (\forall \mathcal{V})\; t_1 = t'_1$, and then, by Lemma 11.2(1), $t = t_1$ and $t' = t'_1$. Analogously with $M(t,s)$ in place of $I(t,t')$. Now suppose the last rule applied has been **R**; we have the following cases:

- If it is

$$\frac{t \bowtie t' \qquad \mathit{wtterm}(t,k) \bowtie \mathit{true}}{I(t,t') \to \mathit{true}}$$

  then, by Lemma 11.2(4), $k \in K$ and $t \in T_\Sigma(\mathcal{V})_k$ and by Lemma 11.2(3) $t = t'$, so $E \vdash (\forall \mathcal{V})\; t = t'$.

- For the following three rules (corresponding to transitivity, symmetry, and monotonicity) the result is an immediate consequence of the induction hypothesis, recalling that, by assumption, each well-typed term has just one kind. Note that for the monotonicity rule, the conditions $\mathit{wtterm}(x,k) \bowtie \mathit{true}$, assure us that t and t' are well-typed.

- If it is
$$\frac{I(t',t) \bowtie true \qquad M(t',s) \bowtie true}{M(t,s) \to true}$$
then, by induction hypothesis and because terms have just one kind, $t, t' \in T_\Sigma(\mathcal{V})_k$, $s \in S_k$, $E \vdash (\forall \mathcal{V})\ t = t'$ and $E \vdash (\forall \mathcal{V})\ t' : s$, and so we have $E \vdash (\forall \mathcal{V})\ t : s$.

- For a rule corresponding to an equation or membership simply note that the conditions $\overline{wtterm(x,k) \bowtie true}$ together with those $x \bowtie y_j$ resulting from linearising guarantee, by virtue of Lemma 11.2(4,3), that all variables which arose from the same one are instantiated with the same term, and that this has the correct kind. Then the result holds by induction hypothesis.

On the other hand, the faithfulness with which the conditional rewrite rules reflect the rules in membership equational logic makes the other implication easy to be proved by induction on the derivation. For example, if the last rule used is reflexivity

$$\frac{}{E \vdash (\forall \mathcal{V})\ t = t},$$

then by Lemma 11.2(4) we have $\delta(T) \vdash_{\mathrm{CRWL}} wtterm(t,k) \to true$, and instantiating x_1 and x_2 with t and y_1 with k in $I(x_1,x_2) \to true \Leftarrow x_1 \bowtie x_2, wtterm(x_1,y_1) \bowtie true$, we get the result. □

With these results we are already in a position to prove the correctness of the translation.

11.4 Proposition *Given any RL-theory T with signature $((K,\Sigma,S),E)$ and set of rules Γ, and given $l, r \in T_\Sigma(\mathcal{V})$, the following are equivalent:*

1. $T \vdash_{\mathrm{RL}} [l] \to [r]$;
2. $(\exists l' \in [l], \exists r' \in [r])\ \delta(T) \vdash_{\mathrm{CRWL}} R(l',r') \to true$;
3. $(\forall l' \in [l], \forall r' \in [r])\ \delta(T) \vdash_{\mathrm{CRWL}} R(l',r') \to true$.

Proof. The equivalence between 2 and 3 holds because of Proposition 11.3 and the rules $R(x,y) \to true \Leftarrow R(x',y) \bowtie true, I(x,x') \bowtie true$ and $R(x,y) \to true \Leftarrow R(x,y') \bowtie true, I(y,y') \bowtie true$.

Let us see $1 \Rightarrow 2$, by induction on the derivation of $T \vdash_{\mathrm{RL}} [l] \to [r]$. According to the last rule applied, we have:

- **RF**. In this case $[l] = [r]$ and we can take $l' = r' = l$.

- **TR**. From
$$\frac{T \vdash_{\mathrm{RL}} [l] \to [t] \qquad T \vdash_{\mathrm{RL}} [t] \to [r]}{T \vdash_{\mathrm{RL}} [l] \to [r]}$$
and the induction hypothesis it follows $\delta(T) \vdash_{\mathrm{CRWL}} R(l', t') \to \mathit{true}$ and $\delta(T) \vdash_{\mathrm{CRWL}} R(t'', r') \to \mathit{true}$ for some $l' \in [l]$, $r' \in [r]$, $t', t'' \in [t]$. But then, by the equivalence between 2 and 3, $\delta(T) \vdash_{\mathrm{CRWL}} R(t', r') \to \mathit{true}$, and we can deduce $\delta(T) \vdash_{\mathrm{CRWL}} R(l', r') \to \mathit{true}$.

- **CG**. Similarly.

- **RP**. We have, for some $[l(\overline{v})] \to [r(\overline{v})]$ *if* $[a_1(\overline{v})] \to [b_1(\overline{v})], \ldots, [a_m(\overline{v})] \to [b_m(\overline{v})]$ in Γ,
$$\frac{\begin{array}{c} T \vdash_{\mathrm{RL}} [w_1] \to [w_1'] \quad \ldots \quad T \vdash_{\mathrm{RL}} [w_n] \to [w_n'] \\ T \vdash_{\mathrm{RL}} [a_1(\overline{w}/\overline{v})] \to [b_1(\overline{w}/\overline{v})] \quad \ldots \quad T \vdash_{\mathrm{RL}} [a_m(\overline{w}/\overline{v})] \to [b_m(\overline{w}/\overline{v})] \end{array}}{T \vdash_{\mathrm{RL}} [l(\overline{w}/\overline{v})] \to [r(\overline{w'}/\overline{v})]}.$$
By induction hypothesis, and the equivalence between 2 and 3, $\delta(T) \vdash_{\mathrm{CRWL}} R(w_i, w_i') \to \mathit{true}$ for $i = 1, \ldots, n$, and $\delta(T) \vdash_{\mathrm{CRWL}} R(a_i(\overline{w}/\overline{v}), b_i(\overline{w}/\overline{v})) \to \mathit{true}$ for $i = 1, \ldots, m$. Working with the translated rule, if all variables x corresponding to a variable v_i (recall that rules are linearized) are instantiated with w_i, and analogously for x' and w_i', the conditions of the form $x \bowtie y_j$, $\mathit{wtterm}(x, k) \bowtie \mathit{true}$, are also satisfied and therefore we get $\delta(T) \vdash_{\mathrm{CRWL}} R(l(\overline{w}/\overline{v}), r(\overline{w'}/\overline{v})) \to \mathit{true}$.

For $2 \Rightarrow 1$, it will be enough to prove that if $\delta(T) \vdash_{\mathrm{CRWL}} R(l, r) \to \mathit{true}$ then $l, r \in T_\Sigma(\mathcal{V})$ and $T \vdash_{\mathrm{RL}} [l] \to [r]$. The last rule used in such a derivation must be **TR** or **R**.

- **TR**. From
$$\frac{\delta(T) \vdash_{\mathrm{CRWL}} R(l, r) \to e \qquad \delta(T) \vdash_{\mathrm{CRWL}} e \to \mathit{true}}{\delta(T) \vdash_{\mathrm{CRWL}} R(l, r) \to \mathit{true}}$$
it follows that e is $\bot$, *true*, or $R(l', r')$ with $\delta(T) \vdash_{\mathrm{CRWL}} l \to l'$, $\delta(T) \vdash_{\mathrm{CRWL}} r \to r'$. The first case is not possible; for *true* we apply the induction hypothesis; in the third case, by induction hypothesis, $l', r' \in T_\Sigma(\mathcal{V})$ and $T \vdash_{\mathrm{RL}} [l'] \to [r']$, and by Lemma 11.2(1), $l = l'$ and $r = r'$.

- **R**. Let us just consider the most complex case, namely, when the translation of one of the rules in Γ is used (the remaining cases are the rules reflecting the reflexivity, transitivity and congruence of the calculus, as well as the two rules dealing with rewriting modulo the set of equations):

$$\begin{array}{rl} R(l(\overline{x}), r(\overline{x'})) \to true \Leftarrow & R(a_1(\overline{x}), b_1(\overline{x})) \bowtie true, \ldots, R(a_m(\overline{x}), b_m(\overline{x})) \bowtie true, \\ & \overline{wtterm(x,k) \bowtie true}, \overline{wtterm(x',k) \bowtie true}, \\ & \overline{R(x,x') \bowtie true}. \end{array}$$

The conditions $\overline{wtterm(x,k) \bowtie true}$, $\overline{wtterm(x',k) \bowtie true}$ ensure that variables are instantiated with terms with the right kind, and the conditions imposed by linearisation (not reflected above) that the necessary identifications are made. Then the result follows by induction hypothesis, applying the replacement rule of RL. □

§12. Reflection in CRWL and in RL

Intuitively, a reflective logic is a logic in which important aspects of its metatheory, such as theories and entailment, can be represented and reasoned about *in* the logic. A general axiomatic notion of reflective logic was recently proposed in [6]. The notion is itself expressed in terms of the notion of an entailment system (see Section 2.2).

Given an entailment system $\mathcal{E}$ and a nonempty set of theories $\mathcal{C}$ in it, a theory U is $\mathcal{C}$-universal if there is a function, called a representation function,

$$\overline{(\_ \vdash \_)} : \bigcup_{T \in \mathcal{C}} (\{T\} \times sen(T)) \to sen(U),$$

such that for each $T \in \mathcal{C}$, $\varphi \in sen(T)$,

$$T \vdash \varphi \quad \text{iff} \quad U \vdash \overline{T \vdash \varphi}.$$

If, in addition, $U \in \mathcal{C}$, then the entailment system $\mathcal{E}$ is called $\mathcal{C}$-reflective. Finally, a reflective logic is a logic whose entailment system is $\mathcal{C}$-reflective for $\mathcal{C}$ the class of all finitely presentable theories in the logic.

Note that in a reflective logic, since U itself is representable, representation can be iterated; hence we immediately have a "reflective tower"

$$T \vdash \varphi \quad \text{iff} \quad U \vdash \overline{T \vdash \varphi} \quad \text{iff} \quad U \vdash \overline{U \vdash \overline{T \vdash \varphi}} \ldots$$

RL has been proved to be reflective in [6, 7, 10] and we will use this result to obtain an analogous one for CRWL. Strictly speaking, the notion of reflection does not apply to CRWL as it was observed in Section 5 that it does not have an associated entailment system. Even in the case of RL, except for the original result in [6], which made use of a slightly restricted version of the entailment system $\mathcal{E}_{\mathrm{RL}}^{\mathrm{unc}}$ presented in Section 6, subsequent generalizations do not fit within the formal definition of reflection as they allow conditional sentences on the left of the entailment relation but not on the right. For this reason, in what follows we consider a "loose" definition of reflection (making use of some kind of "weak" entailment system) general enough to encompass all the cases just discussed.

Let $\mathcal{U}$ be a universal theory for RL; we will use it to prove that CRWL is also reflective. In Sections 7 and 8 we defined mappings α and β that map a theory T in CRWL, respectively in RL, to a theory in RL, respectively in CRWL, which simulates the behaviour of T. By abuse of notation we will also use α and β to name the translation of sequents defined in those sections. (Recall that $\alpha(T)$ is *not* obtained by simply applying α to every $\varphi \in T$, and analogously for β.) Then, given an arbitrary CRWL-theory T and a statement φ, we have the following chain of equivalences:

$$\begin{array}{rcl} T \vdash_{\mathrm{CRWL}} \varphi & \iff & \alpha(T) \vdash_{\mathrm{RL}} \alpha(\varphi) \\ & \iff & \mathcal{U} \vdash_{\mathrm{RL}} \overline{\alpha(T) \vdash \alpha(\varphi)} \\ & \iff & \beta(\mathcal{U}) \vdash_{\mathrm{CRWL}} \beta(\overline{\alpha(T) \vdash \alpha(\varphi)}), \end{array}$$

which proves that $\beta(\mathcal{U})$ is universal in CRWL.

As it has been pointed out previously, the reflective results about RL have been proved only for special cases. The one we need here is that in which the underlying equational logic is unsorted and the rules are conditional [10]. Moreover, the results presented here allow us to extend the reflective results to the full power of RL with membership equational logic. For that, we can consider that α yields theories in RL over membership equational logic because unsorted equational logic is naturally embeddable in it, and let δ be the function defined in Section 11 such that, for T a MERL-theory, $T \vdash_{\mathrm{RL}} \varphi \iff \delta(T) \vdash_{\mathrm{CRWL}} \delta(\varphi)$. Then, for $\mathcal{W}$ any universal theory in CRWL,

$$\begin{array}{rcl} T \vdash_{\mathrm{RL}} \varphi & \iff & \delta(T) \vdash_{\mathrm{CRWL}} \delta(\varphi) \\ & \iff & \mathcal{W} \vdash_{\mathrm{CRWL}} \overline{\delta(T) \vdash \delta(\varphi)} \\ & \iff & \alpha(\mathcal{W}) \vdash_{\mathrm{RL}} \alpha(\overline{\delta(T) \vdash \delta(\varphi)}), \end{array}$$

which proves that $\alpha(\mathcal{W})$ is universal in MERL.

We can try to develop further these results and, for that, let us take a closer look at the translation of Section 7. It can be seen that the rewriting relation does not play a key role, in the sense that nondeterminism is not involved and it seems that, at most, it forbids the derivation of equalities such as $\mathit{tterm}(v_0) = \mathit{pexpr}(v_1)$ that could be obtained if the rewriting rules were substituted by equations. Forgetting about equations for the moment, we can safely move from RL to membership equational logic by substituting the rewriting relation by memberships.

More precisely, given a signature Σ with constructors in CRWL we associate to it a membership equational theory $((K, \Sigma', S), \Gamma')$ over a signature with just one kind, $K = \{*\}$, and a unique sort $S_* = \{\mathit{true}\}$, and $\Sigma' = C_\Sigma \cup F_\Sigma \cup \{\mathit{tterm}, \mathit{pterm}, \mathit{pexpr}, \bot, \bowtie, R\} \cup \mathcal{V}$, with *tterm*, *pterm*, *pexpr*, R and $\bowtie$, binary functions, and $\bot$ and the elements of $\mathcal{V}$, constants. The sentences of Γ' are obtained from the rewrite rules of Section 7 by substituting memberships for the rewrite relation; for example, a rewrite rule like

$$R(x, \bot) \to \mathit{true} \ \mathit{if} \ \mathit{pexpr}(x) \to \mathit{true}$$

becomes the conditional membership

$$(\forall \mathcal{X}) \ R(x, \bot) : \mathit{true} \Leftarrow \mathit{pexpr}(x) : \mathit{true}.$$

Then, given a CRWL-theory $T = (\Sigma, \Gamma)$ we associate to it the membership equational theory $\alpha'(T)$ resulting from adding to $((K, \Sigma', S), \Gamma')$ a conditional membership

$$\begin{array}{rcl} (\forall \mathcal{X}) \ R(l(\overline{x}), r(\overline{x})) : \mathit{true} & \Leftarrow & a_1(\overline{x}) \bowtie b_1(\overline{x}) : \mathit{true} \wedge \ldots \wedge a_m(\overline{x}) \bowtie b_m(\overline{x}) : \mathit{true} \wedge \\ & & \mathit{pterm}(x_1) : \mathit{true} \wedge \ldots \wedge \mathit{pterm}(x_n) : \mathit{true} \end{array}$$

for every $l(\overline{v}) \to r(\overline{v}) \Leftarrow a_1(\overline{v}) \bowtie b_1(\overline{v}), \ldots, a_m(\overline{v}) \bowtie b_m(\overline{v})$ in Γ. (Recall that we had different sets of variables for CRWL and RL.) Then, it can be proved that

12.1 Proposition *Given a CRWL-theory* $T = (\Sigma, \Gamma)$ *such that its associated membership equational theory is* $\alpha'(T) = ((K, \Sigma', S), \Gamma')$*, then, if* $l, r, a, b \in T_{\Sigma'}(\mathcal{X})$*:*

$$\begin{array}{rcl} l, r \in \mathit{Expr}_\bot(\Sigma, \mathcal{V}) \ \mathit{and} \ T \vdash_{\mathrm{CRWL}} l \to r & \Longleftrightarrow & \alpha'(T) \vdash (\forall \mathcal{X}) \ R(l, r) : \mathit{true} \\ a, b \in \mathit{Expr}_\bot(\Sigma, \mathcal{V}) \ \mathit{and} \ T \vdash_{\mathrm{CRWL}} a \bowtie b & \Longleftrightarrow & \alpha'(T) \vdash (\forall \mathcal{X}) \ a \bowtie b : \mathit{true}. \end{array}$$

□

The proof parallels that of Section 7; we will prove the analogous to Lemmas 7.2 and 7.3, but leave out the proof of the main proposition which it is a bit longer.

12.2 Lemma *Let* $T = (\Sigma, \Gamma)$ *be a CRWL-theory,* $\alpha'(T) = ((K, \Sigma', S), \Gamma')$*, and* $e, e' \in T_{\Sigma'}(\mathcal{X})$*. If* $\alpha'(T) \vdash (\forall \mathcal{X}) \ e = e'$*, then* $e = e'$*.*

Proof. Trivial by induction on the derivation, as the only sentences that can be instantiated by modus ponens are memberships. □

(Note that Lemma 7.2 had a second case which would be trivial here and that, in fact, will not be needed.)

12.3 Lemma *If* $T = (\Sigma, \Gamma)$ *is a CRWL-theory,* $\alpha'(T) = ((K, \Sigma', S), \Gamma')$*, and* $e \in T_{\Sigma'}(\mathcal{X})$*, then:*

1. $e \in \mathit{Term}(\Sigma, \mathcal{V}) \iff \alpha'(T) \vdash (\forall \mathcal{X})\ \mathit{tterm}(e) : \mathit{true}$,
2. $e \in \mathit{Term}_\perp(\Sigma, \mathcal{V}) \iff \alpha'(T) \vdash (\forall \mathcal{X})\ \mathit{pterm}(e) : \mathit{true}$,
3. $e \in \mathit{Expr}_\perp(\Sigma, \mathcal{V}) \iff \alpha'(T) \vdash (\forall \mathcal{X})\ \mathit{pexpr}(e) : \mathit{true}$.

Proof.

1. For the $\Rightarrow$ part, by structural induction on the expression e: If $e = v_i$ the results follows by modus ponens, applying $(\forall \mathcal{X})\ \mathit{tterm}(v_i) : \mathit{true}$; if $e = h(e_1, \ldots, e_n)$ with $h \in C^n_\Sigma$, from $(\forall \mathcal{X})\ \mathit{tterm}(h(x_1, \ldots, x_n)) : \mathit{true} \Leftarrow \mathit{tterm}(x_1) : \mathit{true} \wedge \ldots \wedge \mathit{tterm}(x_n) : \mathit{true}$ and the induction hypothesis.

   For the $\Leftarrow$ part, by induction on the last rule of the derivation:

   - Reflexivity, symmetry, transitivity, and congruence, are not possible.
   - Membership. If for some $e' \in T_{\Sigma'}(\mathcal{X})$

     $$\frac{\alpha'(T) \vdash e' = e \qquad \alpha'(T) \vdash \mathit{tterm}(e') : \mathit{true}}{\alpha'(T) \vdash \mathit{tterm}(e) : \mathit{true}},$$

     then, by the induction hypothesis we have $e' \in \mathit{Term}(\Sigma, \mathcal{V})$ and, by Lemma 12.2, $e = e'$.
   - Modus ponens. The only sentences that can be used are $(\forall \mathcal{X})\ \mathit{tterm}(v_i) : \mathit{true}$ with $v_i \in \mathcal{V}$, and $(\forall \mathcal{X})\ \mathit{tterm}(h(x_1, \ldots, x_n)) : \mathit{true} \Leftarrow \mathit{tterm}(x_1) : \mathit{true} \wedge \ldots \wedge \mathit{tterm}(x_n) : \mathit{true}$ with $h \in C^n_\Sigma$. In the first case there is nothing to prove; in the second, simply apply the induction hypothesis.
2. Facts 2 and 3 are proved in an analogous way. $\square$

Let then T be a membership equational theory. As T can be seen as a MERL-theory with empty set of rules the result of Proposition 11.3 applies and, given terms t and t' and $\mathcal{W}$ universal in CRWL, we have the following chain of equivalences:

$$\begin{array}{rcl} T \vdash (\forall \mathcal{X})\ t = t' & \iff & \delta(T) \vdash_{\text{CRWL}} I(t, t') \to \mathit{true} \\ & \iff & \mathcal{W} \vdash_{\text{CRWL}} \overline{\delta(T) \vdash I(t, t') \to \mathit{true}} \\ & \iff & \alpha'(\mathcal{W}) \vdash \alpha'(\overline{\delta(T) \vdash I(t, t') \to \mathit{true}}), \end{array}$$

and similarly for a membership $(\forall \mathcal{X})\ t : s$. These equivalences *nearly* prove that $\alpha'(\mathcal{W})$ is universal in membership equational logic. The point missing is that we have been working with the set $\mathcal{X}$ of *all* variables instead of an arbitrary set of variables containing all those appearing in the terms t and t'. Without this restriction we would only have a left to right implication instead of the first equivalence because of the well-known complications with quantification in many-sorted equational deduction [13]. While we can always safely add more variables to those in the quantification of a provable sentence, it is possible to find a theory T, terms t and t', and a set $\mathcal{Y} \subseteq \mathcal{X}$ containing all variables in t and t' such that $T \vdash (\forall \mathcal{X})\ t = t'$, but $T \not\vdash (\forall \mathcal{Y})\ t = t'$; therefore, from $\delta(T) \vdash_{\mathrm{CRWL}} I(t, t') \to \mathit{true}$ it does not follow that $T \vdash (\forall \mathcal{Y})\ t = t'$. A way out of this problem would be to consider only signatures with nonempty *kinds*, so that we could forget about making variables explicit in the equations. Under this assumption, the above chain of equivalences shows that membership equational logic is reflective.

§13. An Institution for CRWL

We associate to CRWL an institution $\mathcal{I}_{\mathrm{CRWL}} = (\mathbf{Sign}, \mathit{sen}, \mathbf{Mod}, \models)$, given by:

- **Sign**: the category of signatures with constructors and signature morphisms, as defined in Section 3;
- $\mathit{sen} : \mathbf{Sign} \to \mathbf{Set}$ the functor assigning to each signature Σ the set of all conditional rewrite rules over it, and to each signature morphism, its extension to rewrite rules;
- $\mathbf{Mod} : \mathbf{Sign}^{\mathrm{op}} \to \mathbf{Cat}$ the functor assigning to each signature the category of CRWL-algebras and homomorphisms over it, and to each $\sigma : \Sigma \to \Sigma'$ the forgetful functor taking $\mathcal{A}' \in |\mathbf{Mod}(\Sigma')|$ to the CRWL-algebra $\mathcal{A}'_\sigma$ with the same underlying poset and such that $h^{\mathcal{A}'_\sigma} = \sigma(h)^{\mathcal{A}'}$ for all $h \in \Sigma$, and which is the identity over homomorphisms;
- $\models$ the satisfiability relation in CRWL.

In [24], a restricted version of this institution (although more suitable for the study of modules intended there) is assigned to CRWL. Our own definition is also presented there, but no proofs of its correction are given. Also in [24] a weaker version of the following proposition, in which signatures share the same constructor symbols and signature morphisms are the identity over them, is proved.

13.1 Proposition *Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism and let $\mathcal{A}' \in |\mathbf{Mod}(\Sigma')|$ be a CRWL-algebra. Then, for every expression $e \in \mathit{Expr}_\perp(\Sigma, \mathcal{X})$ and every valuation η over $\mathcal{A}'$,*

$$[\![\sigma(e)]\!]^{\mathcal{A}'}\eta = [\![e]\!]^{\mathcal{A}'_\sigma}\eta.$$

Proof. Structural induction on e. For $e = x$ it is trivial because $\sigma(x) = x$. If $e = h(e_1, \dots, e_n)$ with $h \in C^n_\Sigma \cup F^n_\Sigma$, then we have $[\![\sigma(e_i)]\!]^{\mathcal{A}'}\eta = [\![e_i]\!]^{\mathcal{A}'_\sigma}\eta$, $1 \le i \le n$, by induction hypothesis, and

$$\begin{aligned} [\![\sigma(h(e_1, \dots, e_n))]\!]^{\mathcal{A}'}\eta &= [\![\sigma(h)(\sigma(e_1), \dots, \sigma(e_n))]\!]^{\mathcal{A}'}\eta \\ &= \sigma(h)^{\mathcal{A}'}([\![\sigma(e_1)]\!]^{\mathcal{A}'}\eta, \dots, [\![\sigma(e_n)]\!]^{\mathcal{A}'}\eta) \\ &= h^{\mathcal{A}'_\sigma}([\![e_1]\!]^{\mathcal{A}'_\sigma}\eta, \dots, [\![e_n]\!]^{\mathcal{A}'_\sigma}\eta) \\ &= [\![h(e_1, \dots, e_n)]\!]^{\mathcal{A}'_\sigma}\eta. \end{aligned}$$

□

13.2 Proposition $\mathcal{I}_{\mathrm{CRWL}}$ *is an institution.*

Proof. It is immediate to see that **Sign** is a category and *sen* is a functor. For **Mod**, we must first check that it is well defined over signature morphisms. Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism, $\mathcal{A}', \mathcal{B}' \in |\mathbf{Mod}(\Sigma')|$, and $F : \mathcal{A}' \to \mathcal{B}'$ a homomorphism. Is $F = \mathbf{Mod}(\sigma)(F)$ a homomorphism from $\mathcal{A}'_\sigma$ to $\mathcal{B}'_\sigma$?

- for all $u \in D^{\mathcal{A}'_\sigma} = D^{\mathcal{A}'}$, there is $v \in D^{\mathcal{B}'_\sigma} = D^{\mathcal{B}'}$ such that $F(u) = \langle v \rangle$, because $F : \mathcal{A}' \to \mathcal{B}'$ is a homomorphism;
- $F(\bot_{\mathcal{A}'_\sigma}) = F(\bot_{\mathcal{A}'}) = \bot_{\mathcal{B}'} = \bot_{\mathcal{B}'_\sigma}$;
- for all $c \in C^n_\Sigma, u_i \in D^{\mathcal{A}'_\sigma}$: $F(c^{\mathcal{A}'_\sigma}(u_1, \dots, u_n)) = F(\sigma(c)^{\mathcal{A}'}(u_1, \dots, u_n)) = \sigma(c)^{\mathcal{B}'}(F(u_1), \dots, F(u_n)) = c^{\mathcal{B}'_\sigma}(F(u_1), \dots, F(u_n))$;
- for all $f \in F^n_\Sigma, u_1 \in D^{\mathcal{A}'_\sigma}$: $F(f^{\mathcal{A}'_\sigma}(u_1, \dots, u_n)) = F(\sigma(f)^{\mathcal{A}'}(u_1, \dots, u_n)) \subseteq \sigma(f)^{\mathcal{B}'}(F(u_1), \dots, F(u_n)) = f^{\mathcal{B}'_\sigma}(F(u_1), \dots, F(u_n))$.

It is easy to see that **Mod** preserves identities and composition, so **Mod** is indeed a functor.

It only remains to prove that the satisfaction condition is verified. Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism, $\mathcal{A}' \in |\mathbf{Mod}(\Sigma')|$, and $\varphi \in sen(\Sigma)$. We have to prove that

$$\mathcal{A}' \models \sigma(\varphi) \iff \mathcal{A}'_\sigma \models \varphi.$$

By Proposition 13.1,

$$[\![e]\!]^{\mathcal{A}'_\sigma}\eta = [\![\sigma(e)]\!]^{\mathcal{A}'}\eta$$

for every $e \in Expr_\bot(\Sigma, \mathcal{X})$ and valuation η. Let $\varphi = e \to e'$ be a reduction statement. Then, for any valuation η,

$$\begin{aligned} (\mathcal{A}', \eta) \models \sigma(\varphi) &\iff [\![\sigma(e')]\!]^{\mathcal{A}'}\eta \subseteq [\![\sigma(e)]\!]^{\mathcal{A}'}\eta \\ &\iff [\![e']\!]^{\mathcal{A}'_\sigma}\eta \subseteq [\![e]\!]^{\mathcal{A}'_\sigma}\eta \quad \iff \quad (\mathcal{A}'_\sigma, \eta) \models \varphi, \end{aligned}$$

and analogously for φ a joinability statement. Now, if $l \to r \Leftarrow C$ is a conditional rewrite rule, it follows that $\mathcal{A}'_\sigma \models C \iff \mathcal{A}' \models \sigma(C)$ and $\mathcal{A}'_\sigma \models l \to r \iff \mathcal{A}' \models \sigma(l \to r)$, and thus the satisfaction condition is indeed verified. $\square$

We will now study some of the properties of this institution. First of all, every category of models has products. The construction make explicit below is the standard one, though a bit obscure due to its generality in order to deal with infinite products and the own characteristics of CRWL.

13.3 Proposition *If* $T = (\Sigma, \Gamma)$ *is a CRWL-theory, the category* $\mathbf{Mod}(T)$ *has products.*

Proof. Let I be a set and $\{\mathcal{A}_i \mid i \in I\}$ a family of CRWL-algebras in $\mathbf{Mod}(T)$. We define their product $\prod_{i\in I} \mathcal{A}_i$ by

- The poset $D^{\Pi\mathcal{A}_i} = \{a : I \to \bigcup_{i\in I} D^{\mathcal{A}_i} \mid a(i) \in \mathcal{A}_i\}$, with $a \sqsubseteq a'$ if and only if $a(i) \sqsubseteq a'(i)$ for all $i \in I$;
- For all $f \in F^n_\Sigma$, the function $f^{\Pi\mathcal{A}_i}(a_1, \ldots, a_n) = \prod_{i\in I} f^{\mathcal{A}_i}(a_1(i), \ldots, a_n(i)) = \{a \in D^{\Pi\mathcal{A}_i} \mid a(i) \in f^{\mathcal{A}_i}(a_1(i), \ldots, a_n(i)) \text{ for all } i \in I\}$;
- For all $c \in C^n_\Sigma$, the function $c^{\Pi\mathcal{A}_i}(a_1, \ldots, a_n) = \prod_{i\in I} c^{\mathcal{A}_i}(a_1(i), \ldots, a_n(i)) = \{a \in D^{\Pi\mathcal{A}_i} \mid a(i) \in c^{\mathcal{A}_i}(a_1(i), \ldots, a_n(i)) \text{ for all } i \in I\}$.

Clearly, $D^{\Pi\mathcal{A}_i}$ is a poset, the bottom being the function mapping each $i \in I$ to the bottom in $\mathcal{A}_i$. If $a \in f^{\Pi\mathcal{A}_i}(a_1, \ldots, a_n)$ and $a' \sqsubseteq a$ then, for all $i \in I$ we have $a'(i) \sqsubseteq a(i) \in f^{\mathcal{A}_i}(a_1(i), \ldots, a_n(i))$ and, as $f^{\mathcal{A}_i}(a_1(i), \ldots, a_n(i))$ is a cone, it must be $a'(i) \in f^{\mathcal{A}_i}(a_1(i), \ldots, a_n(i))$ and thus $f^{\Pi\mathcal{A}_i}(a_1, \ldots, a_n)$ is a cone too. Besides, it is easy to check that $f^{\Pi\mathcal{A}_i}$ is monotone. For $c \in C^n_\Sigma$ it turns out that $c^{\Pi\mathcal{A}_i}(a_1, \ldots, a_n) = \langle a \rangle$, where $c^{\mathcal{A}_i}(a_1(i), \ldots, a_n(i)) = \langle a(i) \rangle$ for all $i \in I$. Therefore $\prod_{i\in I} \mathcal{A}_i$ is well-defined.

We also have to prove that $\prod_{i\in I} \mathcal{A}_i$ satisfies Γ. The proof follows along the same lines as the previous paragraph. An element a is maximal in $\prod_{i\in I} \mathcal{A}_i$ if and only if it is the product of maximal elements in $\mathcal{A}_i$, i.e., $a(i)$ is maximal in $\mathcal{A}_i$ for all $i \in I$. Valuations in $\prod_{i\in I} \mathcal{A}_i$ are products of valuations in $\mathcal{A}_i$. For every expression e and valuations $\eta_i : \mathcal{X} \to \mathcal{A}_i$, it is easily proved by structural induction that $[\![e]\!]^{\Pi\mathcal{A}_i} \prod_{i\in I} \eta_i = \prod_{i\in I} [\![e]\!]^{\mathcal{A}_i} \eta_i$. But then $\prod_{i\in I} \mathcal{A}_i$ satisfies a reduction or a joinability statement if and only if every $\mathcal{A}_i$ satisfies it, and therefore $\prod_{i\in I} \mathcal{A}_i$ satisfies Γ.

Finally, $\prod_{i\in I} \mathcal{A}_i$ has the universal property required for products. The projections $P_j : \prod_{i\in I} \mathcal{A}_i \to \mathcal{A}_j$, defined as $P_j(a) = \langle a(j) \rangle$, are CRWL-homomorphisms.

Clearly P_j is monotone and, for example, given $c \in C^n_\Sigma$, $a_1, \dots, a_n \in \prod_{i \in I} \mathcal{A}_i$, and if $c^{\mathcal{A}_j}(a_1(j), \dots, a_n(j)) = \langle v_j \rangle$ for some $v_j \in \mathcal{A}_j$, then

$$\begin{array}{rcl} P_j(c^{\Pi \mathcal{A}_i}(a_1, \dots, a_n)) & = & \bigcup \{ \langle a(j) \rangle \mid a \in c^{\Pi \mathcal{A}_i}(a_1, \dots, a_n) \} \\ & = & \bigcup \{ \langle a(j) \rangle \mid a \in D^{\Pi \mathcal{A}_i}, a(i) \in c^{\mathcal{A}_i}(a_1(i), \dots, a_n(i)) \} \\ & = & \langle v_j \rangle \\ & = & c^{\mathcal{A}_j}(a_1(j), \dots, a_n(j)) \\ & = & c^{\mathcal{A}_j}(P_j(a_1), \dots, P_j(a_n)), \end{array}$$

and so P_j preserves constructors. Given $\mathcal{B} \in \mathbf{Mod}(T)$ and homomorphisms $F_i : \mathcal{B} \to \mathcal{A}_i$, the unique homomorphism $G : \mathcal{B} \to \prod_{i \in I} \mathcal{A}_i$ such that $P_i \circ G = F_i$ for every $i \in I$ is given by $G(b) = \prod_{i \in I} F_i(b)$. Uniqueness is clear, G is element-valued because an ideal in $\prod_{i \in I} \mathcal{A}_i$ is a product of ideals in $\mathcal{A}_i$, and preservation of functions and constructors follows without difficulty. □

$\mathbf{Mod}(T)$ is not complete, however, as in Section 16 it is shown that, in general, $\mathbf{Mod}(T)$ does not have equalizers.

The institution associated to CRWL in [24] was proved to be semiexact there. This result can be generalized to our case.

13.4 Proposition $\mathcal{I}_{\mathrm{CRWL}}$ *is a semiexact institution.*

Proof. We first prove that the category **Sign** has pushouts. For that, let Σ_0, Σ_1 and Σ_2 be signatures and let $\sigma_1 : \Sigma_0 \to \Sigma_1$ and $\sigma_2 : \Sigma_0 \to \Sigma_2$ be signature morphisms. For simplicity of notation and without loss of generality, in what follows we assume Σ_1 and Σ_2 to be disjoint (renaming if necessary). The construction for the pushout is analogous to that of pushouts in **Set**: we consider the union—it would have been the disjoint union if we had not supposed Σ_1 and Σ_2 to be disjoint—of Σ_1 and Σ_2 (keeping constructors and function symbols separate) and identify $\sigma_1(h)$ with $\sigma_2(h)$ for all $h \in \Sigma_0$. Schematically,

$$\begin{tikzcd} \Sigma_0 \arrow[r, "\sigma_1"] \arrow[d, "\sigma_2"'] & \Sigma_1 \arrow[d, "\mu_1"] \\ \Sigma_2 \arrow[r, "\mu_2"] & \Sigma_3 = (\Sigma_1 \cup \Sigma_2)/\equiv \end{tikzcd}$$

where $\equiv$ is the least equivalence relation in $\Sigma_1 \cup \Sigma_2$ verifying $\sigma_1(h) \equiv \sigma_2(h)$, and μ_1 and μ_2 take each element to its quotient class. Note that only symbols with the same arity can be related. The proof that Σ_3 is the pushout is the same as that in **Set**.

Let us now check that **Mod** takes a pushout in **Sign** into a pullback in **Cat**. The diagram

$$\begin{tikzcd}
\mathbf{Mod}(\Sigma_3) \arrow[r, "\mathbf{Mod}(\mu_2)"] \arrow[d, "\mathbf{Mod}(\mu_1)"'] & \mathbf{Mod}(\Sigma_2) \arrow[d, "\mathbf{Mod}(\sigma_2)"] \\
\mathbf{Mod}(\Sigma_1) \arrow[r, "\mathbf{Mod}(\sigma_1)"] & \mathbf{Mod}(\Sigma_0)
\end{tikzcd}$$

commutes, because it is obtained by applying a functor to a commutative diagram. To see that it satisfies the universal property of pullbacks, let C be a category and $F_1 : C \to \mathbf{Mod}(\Sigma_1)$, $F_2 : C \to \mathbf{Mod}(\Sigma_2)$ be functors such that $\mathbf{Mod}(\sigma_1) \circ F_1 = \mathbf{Mod}(\sigma_2) \circ F_2$. We have to find a unique functor $F : C \to \mathbf{Mod}(\Sigma_3)$ such that $\mathbf{Mod}(\mu_1) \circ F = F_1$ and $\mathbf{Mod}(\mu_2) \circ F = F_2$. For that, if $c \in |C|$, let us consider the algebras $\mathcal{A}_1 = F_1(c)$ and $\mathcal{A}_2 = F_2(c)$. Due to the hypothesis, $\mathbf{Mod}(\sigma_1)(\mathcal{A}_1) = \mathbf{Mod}(\sigma_2)(\mathcal{A}_2)$ so, by the definition of **Mod** over signature morphisms, the underlying posets of $\mathcal{A}_1$ and $\mathcal{A}_2$ coincide. We can then define $F(c)$ to be the algebra $\mathcal{A}_3$ given by

- $D^{\mathcal{A}_3} = D^{\mathcal{A}_1} = D^{\mathcal{A}_2}$,
- for every $[h] \in \Sigma_3$, $[h]^{\mathcal{A}_3} = h^{\mathcal{A}_1}$ if $h \in \Sigma_1$, or $[h]^{\mathcal{A}_3} = h^{\mathcal{A}_2}$ if $h \in \Sigma_2$.

In order to see that $\mathcal{A}_3$ is well-defined let $\sim$ be the relation defined over $\Sigma_1 \cup \Sigma_2$ by

$$\begin{cases} \text{if } h_1, h_2 \in \Sigma_1, & h_1 \sim h_2 \iff h_1^{\mathcal{A}_1} = h_2^{\mathcal{A}_1} \\ \text{if } h_1, h_2 \in \Sigma_2, & h_1 \sim h_2 \iff h_1^{\mathcal{A}_2} = h_2^{\mathcal{A}_2} \\ \text{if } h_1 \in \Sigma_1, h_2 \in \Sigma_2, & h_1 \sim h_2 \iff h_1^{\mathcal{A}_1} = h_2^{\mathcal{A}_2}. \end{cases}$$

It is easy to check that $\sim$ is an equivalence relation and that, as $\mathbf{Mod}(\sigma_1)(\mathcal{A}_1) = \mathbf{Mod}(\sigma_2)(\mathcal{A}_2)$, $\sigma_1(h) \sim \sigma_2(h)$ for all $h \in \Sigma_0$. Then, by definition, the relation $\equiv$ is included in $\sim$. Given $[h] \in \Sigma_3$ and $h_1 \in [h]$, i.e., $h \equiv h_1$, it follows that $h \sim h_1$ and assuming (without loss of generality) that $h \in \Sigma_1$ and $h_1 \in \Sigma_2$, we have $h^{\mathcal{A}_1} = h_1^{\mathcal{A}_2}$ and therefore $[h]^{\mathcal{A}_3}$ is well-defined. $[h]$ for some $h \in \Sigma_1 \cup \Sigma_2$, $\mathcal{A}_3$ is well-defined. Over morphisms, as $\mathbf{Mod}(\sigma_1)$ and $\mathbf{Mod}(\sigma_2)$ are, by definition, the identity, we have by the hypothesis that F_1 and F_2 are equal and we define F to coincide with them: F so defined is actually a functor. Besides, for all $h \in \Sigma_1$

$$h^{\mathbf{Mod}(\mu_1)(\mathcal{A}_3)} = \mu_1(h)^{\mathcal{A}_3} = [h]^{\mathcal{A}_3} = h^{\mathcal{A}_1}$$

and it follows that $\mathbf{Mod}(\mu_1) \circ F = F_1$; analogously $\mathbf{Mod}(\mu_2) \circ F = F_2$. The uniqueness of F is immediate taking the definitions of $\mathbf{Mod}(\mu_1)$ and $\mathbf{Mod}(\mu_2)$ into account. □

§14. An Institution for RL

We again turn our atention to the version of RL which has unsorted and unconditional equational logic as underlying logic. The task of assigning an institution to it will prove to be harder than expected; we begin discussing the most obvious possibilities and the difficulties they pose.

The first idea is to define the category of signatures **Sign** and the functor *sen* in the same way as in the entailment system of Section 6; that is, **Sign** would be the category of equational theories and theory morphisms, and *sen* the functor assigning to an equational theory the set of all its associated conditional rewrite rules. The choice for the functor $\mathbf{Mod} : \mathbf{Sign}^{\mathrm{op}} \to \mathbf{Cat}$ is not so obvious, but a possible one would be to map a signature (Σ, E) to the category $\mathcal{R}$-**Sys**, where $\mathcal{R}$ is the RL-theory over (Σ, E) with empty set of rules; its behaviour over morphisms would be the expected one. But as soon as we face the notion of satisfaction, problems start to arise. As discussed in Section 6 satisfaction in RL is only defined for unconditional rules, and there are at least two ways, both of them seemingly correct, in which this notion can be extended to conditional rewrite rules. The first possibility would be to define, given an $\mathcal{R}$-system $\mathcal{S}$,

$$\mathcal{S} \models [t] \to [t'] \textit{ if } [a_1] \to [b_1] \wedge \ldots \wedge [a_m] \to [b_m]$$

if

$$\mathcal{S} \models [a_i] \to [b_i] \quad i = 1, \ldots, m \quad \Longrightarrow \quad \mathcal{S} \models [t] \to [t'].$$

The other alternative is that already presented in Section 6, namely

$$\mathcal{S} \models [t] \to [t'] \textit{ if } [a_1] \to [b_1] \wedge \ldots \wedge [a_m] \to [b_m]$$

if there exists a natural transformation

$$\alpha : t_{\mathcal{S}} \circ J_{\mathcal{S}} \Rightarrow t'_{\mathcal{S}} \circ J_{\mathcal{S}},$$

where $J_{\mathcal{S}} : \mathit{Subeq}((a_{j\mathcal{S}}, b_{j\mathcal{S}})_{1 \le j \le m}) \to \mathcal{S}^n$. The first option looks very natural, but the second one is more in accordance with the spirit of $\mathcal{R}$-systems: which one should we choose? Before thinking of the answer to this question we should ask ourselves whether these two definitions are actually nonequivalent. In this regard it is not hard to show that the second definition implies the first one, but that the converse does not hold is proved by the following counterexample.

Assume an RL-theory $\mathcal{R}$ with $\Sigma = \{c, d, f, g\}$, c and d constants and f and g unary functions, and with empty sets of equations and rules. Let $\mathcal{S}$ be the $\mathcal{R}$-system given by a discrete category with just two objects A and B, and the following interpretation for the elements of Σ: $c_{\mathcal{S}} = A$, $d_{\mathcal{S}} = B$, $f_{\mathcal{S}} = id_{\mathcal{S}}$, $g_{\mathcal{S}}$ the constant functor equal to

B. Clearly, there does not exist a natural transformation $f_{\mathcal{S}} \Rightarrow g_{\mathcal{S}}$ and so, using the first definition, $\mathcal{S}$ trivially satisfies the conditional rule $c \to d \textit{ if } f(x) \to g(x)$. On the other hand, as the subequalizer of $f_{\mathcal{S}}$ and $g_{\mathcal{S}}$ is not the empty category (the pair (B, id_B) belongs to it) and no morphism from $c_{\mathcal{S}}$ to $d_{\mathcal{S}}$ exists, $\mathcal{S}$ is not a model of $c \to d \textit{ if } f(x) \to g(x)$ under the second definition.

Therefore we are forced to choose between one of the two options, the second one being more restrictive than the first one. Meseguer's idea when defining RL was to build an instance of a categorical logic. In a categorical logic, the notion of satisfaction is more restrictive than in a classical one in the "constructive" sense that all the syntactic elements of a sentence must be reified inside any of its models; this is precisely the role played by the subequalizer in the second definition. This comment, in addition to the definition of $\mathcal{R}$-systems (and the existence of a sound and complete calculus), supports the choice of this second alternative as our definition of satisfaction for conditional rewrite rules.

After all the discussion in the preceding paragraphs we are finally left with a proposed institution for RL; however, this definition is not entirely satisfactory. Up to this point we have omitted any explicit mention of the set of labels of an RL-theory. Cannot we simply consider it a part of the signature, as pointed out when defining the entailment system $\mathcal{E}^{\mathrm{c}}_{\mathrm{RL}}$? Not here. Due to the set of labels L in an RL-theory $\mathcal{R} = (\Sigma, E, L, \Gamma)$, the elements of Γ become special, *labelled* rewrite rules. These rules force $\mathcal{R}$-systems to have a certain internal structure: not only $\mathcal{R}$-systems must satisfy them, but also must associate to them a *distinguished* natural transformation that—in addition to the operations in the signature—must be preserved by homomorphisms. And this structure cannot be imposed by the set of labels alone (as there is no information about the source or the target in the label itself), but in conjunction with Γ. In our proposed institution, however, this structure is not considered because we have defined $\mathbf{Mod}(\Sigma, E) = (\Sigma, E, \emptyset, \emptyset)$-$\mathbf{Sys}$. In particular, homomorphisms are not subjected to preserve any rewrite rule and, for Γ a set of rewrite rules and $\mathcal{R}$ the RL-theory given by (omitting labels) (Σ, E, Γ), the categories $\mathcal{R}$-$\mathbf{Sys}$ and $\mathbf{Mod}(\Gamma)$ turn out to be different, in opposition to our original intention.

Even if equality with $\mathcal{R}$-$\mathbf{Sys}$ were not a primary concern, the category $\mathbf{Mod}(\Gamma)$ would still be unsatisfactory since it doesn't follow one of the "sacred" principles in algebraic specification: the existence of initial objects. To see this consider a signature consisting of just two constants a and b, and $\Gamma = \{a \to b\}$. A category $\mathcal{S}$ with two objects A and B interpreting a and b, and two morphisms m and n from A to B, is an element of $\mathbf{Mod}(\Gamma)$; mapping m to n and n to m we obtain a homomorphism $F : \mathcal{S} \to \mathcal{S}$. If $\mathcal{I}$ were an initial object in $\mathbf{Mod}(\Gamma)$, there should exist a morphism $h : a_{\mathcal{I}} \to b_{\mathcal{I}}$ in $\mathcal{I}$, and a (unique) homomorphism $G : \mathcal{I} \to \mathcal{S}$ preserving the operations in Σ. But then, $a_{\mathcal{I}} \neq b_{\mathcal{I}}$ and $G(h)$ should be equal to either m or n;

composing G with F we would obtain another homomorphism from $\mathcal{I}$ to $\mathcal{S}$.

The above remarks show that the set of labels cannot be simply discarded, and compel us to find a better definition in which the extra structure provided by labelled rules is taken into account. The obvious step towards this direction is to redefine the functor **Mod** so that models are endowed with a richer structure. Now, the objects in $\mathbf{Mod}(\Sigma, E)$ can be the objects in $\bigcup_{\substack{\mathcal{R}=(\Sigma,E,L_R,R)\\ R\subseteq sen(\Sigma,E)}} \mathcal{R}\text{-}\mathbf{Sys}$ (L_R is any labelling of R); that is, we consider all possible RL-theories $\mathcal{R}$ over (Σ, E) and collect the corresponding $\mathcal{R}$-systems. The $\mathcal{R}$-systems in our original definition are included in this new one by taking $L_R = R = \emptyset$. In addition, now we also consider $\mathcal{R}$-systems with distinguished natural transformations. Regarding morphisms $F : \mathcal{S} \to \mathcal{S}'$ in $\mathbf{Mod}(\Sigma, E)$ we have several possibilities. One of them would be just to allow homomorphisms, preserving operations in the signature as well as rewrite rules, between objects $\mathcal{S}$ and $\mathcal{S}'$ belonging to the same category $\mathcal{R}$-**Sys**. Also, we could relax this condition and allow morphisms $F : \mathcal{S} \to \mathcal{S}'$ when $\mathcal{S} \in |\mathcal{R}\text{-}\mathbf{Sys}|$, $\mathcal{S}' \in |\mathcal{R}'\text{-}\mathbf{Sys}|$, and $\mathcal{R} \subseteq \mathcal{R}'$: the additional structure in $\mathcal{S}'$ would simply be ignored. We will not consider more possibilities and rest on these two for intuition. Note, however, that the definitions presented here are two different instances of a very general Grothendieck construction. The first one corresponds to a functor $G : \mathcal{T} \to \mathbf{Cat}$, where $\mathcal{T}$ is the *discrete* category of all RL-theories, mapping $\mathcal{R}$ to $\mathcal{R}$-**Sys**. Enriching $\mathcal{T}$ with inclusion morphisms, G can be extended to $G' : \mathcal{T}^{\mathrm{op}} \to \mathbf{Cat}$ by mapping an inclusion $\mathcal{R} \to \mathcal{R}'$ to the forgetful functor $\mathcal{R}'\text{-}\mathbf{Sys} \to \mathcal{R}\text{-}\mathbf{Sys}$; its associated Grothendieck construction would give rise to the second alternative. By considering different families of morphisms among the RL-theories, we would obtain different notions of morphism in $\mathbf{Mod}(\Sigma, E)$.

Despite all our efforts in the previous paragraph we are still left, essentially, with the same problems we had when we decided to revise our original definition of **Mod**. Whatever Γ might be, there will be models in $\mathbf{Mod}(\Gamma)$ with no distinguished natural transformations associated to the elements in Γ, just because the notion of satisfaction requires the existence of a natural transformation, but not of a *distinguished* one. But then again, and in contrast with what happens in $\mathcal{R}$-**Sys**, homomorphisms between such models are not subjected to preserve the rules in Γ. And the same counterexample as before shows that $\mathbf{Mod}(\Gamma)$ doesn't have an initial object.

Finally, let us consider a third alternative. The last proposal has made it clear that the source of the problem has shifted from the definition of **Mod** to that of satisfaction. In fact, in this context a new definition of satisfaction naturally arises: $\mathcal{S}$ satisfies a rewrite rule if it has a *distinguished* natural transformation associated to it. Combined with the second definition for morphisms in $\mathbf{Mod}(\Sigma, E)$ above (that in which $F : \mathcal{S} \to \mathcal{S}'$ is allowed when $\mathcal{S} \in |\mathcal{R}\text{-}\mathbf{Sys}|$, $\mathcal{S}' \in |\mathcal{R}'\text{-}\mathbf{Sys}|$ and $\mathcal{R} \subseteq \mathcal{R}'$), since now $\mathcal{R}$-systems in $\mathbf{Mod}(\Gamma)$ have distinguished natural transformations associated to the rules in Γ and homomorphisms must preserve them, at least initial objects in

$\mathbf{Mod}(\Gamma)$ are recovered. However, this notion of satisfaction is most restrictive. In order to see why, let us take a closer look at what we understand for distinguished natural transformations in $\mathcal{S} \in \mathbf{Mod}(\Sigma, E)$. For that we mean a triple consisting of a label r, a rule $[t] \to [t']$ *if* $[a_1] \to [b_1] \wedge \ldots \wedge [a_m] \to [b_m]$, and a natural transformation $r_{\mathcal{S}}$ between the appropiate functors. $\mathcal{S}$ satisfies a rule if *and* only if the rule is part of one of the triples associated to $\mathcal{S}$. Consider then a signature Σ with four constants a, b, c, and d, and $\mathcal{S} \in |\mathbf{Mod}(\Sigma, \emptyset)|$ with a distinguished natural transformation $(r, a \to b, r_{\mathcal{S}})$: even in the case wherein $c_{\mathcal{S}} = a_{\mathcal{S}}$ and $d_{\mathcal{S}} = b_{\mathcal{S}}$, $\mathcal{S}$ doesn't satisfy $c \to d$. It follows then that an $\mathcal{R}$-system only satisfies as many rewrite rules as distinguished natural transformations has, what seems to be a bit odd. (It could be argued that it is the notion of distinguished natural transformation what is actually restrictive, and that the label, and even the rule, should be removed from the triple. Against this point of view let us note that the same rewrite rule can belong *twice* to an RL-theory $\mathcal{R}$ under two *different* labels. $\mathcal{R}$-systems are then forced to provide two, possibly different, interpretations—natural transformations—for the same rule, each of them to be preserved by homomorphisms. So, neither the label nor the rule can be discarded in the definition of a distinguished natural transformation.)

Summing up the discussion in the previous paragraphs, we began with a naive proposal of institution in which the structure imposed on models by the rewrite rules in an RL-theory was not taken into account. To remedy this, labelled rules were considered and distinguished natural transformations, that should be preserved by homomorphisms, were associated to them in the models. But even then, the notion of satisfaction did not compel models of a rewrite rule to possess a distinguished natural transformation associated to it, and so again homomorphisms were released from the requirement of preserving it. Accordingly, a new definition of satisfaction was proposed that, though solving some of the previous problems, turned out to be very restrictive.

And now, what? So far, all our attempts to supply RL with a suitable institution have proved to be unsatisfactory. We started with very simple, seemingly natural definitions, were obliged to progressively complicate them, and ended up discussing the subtleties behind the notion of distinguished natural transformation, far away from our initial ideals of simplicity and naturalness. The source of all the difficulties we found through our way lies at the very heart of RL: models are assigned to theories instead of signatures, as it is customary. This gives rise to the distinction between two types of sentences: labelled rules, belonging to RL-theories and imposing a certain structure on the models, and unlabelled rules. An institution for RL which has as its category of signatures the category of equational theories is not able to reflect this duplicity, and that is why all our previous attempts were bound to failure. For this reason we are led to an institution in which the category **Sign** subsumes all the information of an RL-theory. More precisely, we propose to associate to RL the institution $\mathcal{I}_{\mathrm{RL}} = (\mathbf{Sign}, sen, \mathbf{Mod}, \models)$, where:

- **Sign** is the *discrete* category of RL-theories.
- *sen* : **Sign** → **Set** maps each RL-theory to its corresponding set of conditional rewrite rules.
- **Mod** : $\mathbf{Sign}^{\mathrm{op}} \to \mathbf{Cat}$ maps an RL-theory $\mathcal{R}$ to the category $\mathcal{R}$-**Sys**.
- $\models$ is satisfaction in RL, with the extension to conditional rewrite rules involving subequalizers.

Since **Sign** is discrete, this trivially defines an institution. Admittedly, this restriction seems to be not justified. In fact, two types of morphisms of RL-theories are proposed in [20]. Basically, they are equational theory morphisms "preserving" the rules in the RL-theories; however, their formal definition requires the use of 2-categorical notions and will not be given here. A complete study of the institution $\mathcal{I}_{\mathrm{RL}}$ extended with these morphisms will be undertaken in a future occasion. For our purposes, the present definition is general enough as it stands, and its extension would not modify the use we will make of it in Sections 16 and 17. Note also that terminology under this institution can be a bit confusing, as a theory is given by an RL-theory (making use of labelled rules) *plus* some unlabelled rules.

§15. An Institution for ACRWL

Analogously to the untyped case, we can associate an institution to ACRWL. There is only one missing ingredient: signature morphisms. Given two polymorphic signatures Σ and Σ', we propose to define a polymorphic signature morphism $\sigma : \Sigma \to \Sigma'$ as a triple of functions (denoted with the same σ)

$$\sigma : TC_\Sigma \to TC_{\Sigma'}, \quad \sigma : C_\Sigma \to C_{\Sigma'} \quad \text{and} \quad \sigma : F_\Sigma \to F_{\Sigma'},$$

mapping n-ary symbols to n-ary symbols. Composition is that of functions; the free extension to types, expressions, program rules and equational axioms, is the obvious one. A signature morphism $\sigma : \Sigma \to \Sigma'$ is *well-typed* if, for each $h : (\tau_1, \ldots, \tau_n) \to \tau_0$ in Σ, we have $\sigma(h) : (\sigma(\tau_1), \ldots, \sigma(\tau_n)) \to \sigma(\tau_0)$ in Σ'.

We can then define $\mathcal{I}_{\mathrm{ACRWL}} = (\mathbf{Sign}, sen, \mathbf{Mod}, \models)$, where:

- **Sign** is the category of polymorphic signatures and signature morphisms.
- *sen* : **Sign** → **Set** associates to each signature the set of conditional rewrite rules and equational axioms over it, and to signature morphisms the corresponding extension.

- **Mod** : $\mathbf{Sign}^{\mathrm{op}} \to$ **Cat** associates to Σ the category of PT-algebras and homomorphisms over Σ and, to each $\sigma : \Sigma \to \Sigma'$, the functor $\mathbf{Mod}(\sigma)$ mapping every $\mathcal{A}' \in |\mathbf{Mod}(\Sigma')|$ to the algebra $\mathcal{A}'_\sigma$ with the same underlying poset and universe of types, $:^{\mathcal{A}'_\sigma} = :^{\mathcal{A}'}$, and for every $s \in \Sigma$, $s^{\mathcal{A}'_\sigma} = \sigma(s)^{\mathcal{A}'}$. $\mathbf{Mod}(\sigma)$ is defined to be the identity over PT-homomorphisms.
- $\models$ is the satisfaction relation in ACRWL.

15.1 Proposition *$\mathcal{I}_{\mathrm{ACRWL}}$ is an institution.*

Proof. Since composition of functions is associative, **Sign** is a category and *sen* is a functor. To see that **Mod** is well defined over signature morphisms, let $\sigma : \Sigma \to \Sigma'$ and $\mathcal{A}' \in |\mathbf{Mod}(\Sigma')|$; then, $\mathbf{Mod}(\sigma)(\mathcal{A}')$ is trivially well defined. If $\mathcal{B}' \in |\mathbf{Mod}(\Sigma')|$ and $F : \mathcal{A}' \to \mathcal{B}'$ is a homomorphism, we have to check that F is also a homomorphism from $\mathcal{A}'_\sigma$ to $\mathcal{B}'_\sigma$:

- for each type constructor $T \in TC_\Sigma$, and $l_1, \ldots, l_n \in \mathcal{U}^{\mathcal{A}'_\sigma} = \mathcal{U}^{\mathcal{A}'}$, we have $F_t(T^{\mathcal{A}'_\sigma}(l_1, \ldots, l_n)) = F_t(\sigma(T)^{\mathcal{A}'}(l_1, \ldots, l_n)) = \sigma(T)^{\mathcal{B}'}(F_t(l_1), \ldots, F_t(l_n)) = T^{\mathcal{B}'_\sigma}(F_t(l_1), \ldots, F_t(l_n))$.
- the other conditions are proved in a similar manner (see also the proof of Proposition 13.2).

Again, it is immediate to see that **Mod** preserves identities and composition, and so it is indeed a functor.

As for the satisfaction condition, let $\sigma : \Sigma \to \Sigma'$ be a signature morphism and $\mathcal{A}' \in |\mathbf{Mod}(\Sigma')|$. The analogous of Proposition 13.1 also holds (and the proof would be the same): For every expression $e \in \mathit{Expr}_\perp(\Sigma, \mathcal{X})$ and valuation $\xi = (\eta, \mu)$ over $\mathcal{A}'$, $[\![\sigma(e)]\!]^{\mathcal{A}'}\eta = [\![e]\!]^{\mathcal{A}'_\sigma}\eta$. Then, given a conditional rewrite rule φ, $\mathcal{A}' \models \sigma(\varphi) \iff \mathcal{A}'_\sigma \models \varphi$ is proved in exactly the same way as in Proposition 13.2. For an equational axiom $s \approx t$, since a data valuation is safe for a term t if and only if it is safe for $\sigma(t)$, we have

$$\begin{aligned}
\mathcal{A}' \models \sigma(s \approx t) &\iff \begin{cases} [\![\sigma(s)]\!]^{\mathcal{A}'}\eta \supseteq [\![\sigma(t)]\!]^{\mathcal{A}'}\eta & \text{(for all } \eta \text{ safe for } s) \\ [\![\sigma(t)]\!]^{\mathcal{A}'}\eta \supseteq [\![\sigma(s)]\!]^{\mathcal{A}'}\eta & \text{(for all } \eta \text{ safe for } t) \end{cases} \\
&\iff \begin{cases} [\![s]\!]^{\mathcal{A}'_\sigma}\eta \supseteq [\![t]\!]^{\mathcal{A}'_\sigma}\eta & \text{(for all } \eta \text{ safe for } s) \\ [\![t]\!]^{\mathcal{A}'_\sigma}\eta \supseteq [\![s]\!]^{\mathcal{A}'_\sigma}\eta & \text{(for all } \eta \text{ safe for } t) \end{cases} \\
&\iff \mathcal{A}'_\sigma \models s \approx t,
\end{aligned}$$

and thus the satisfaction property is verified. □

It is interesting to note that the institution $\mathcal{I}_{\mathrm{ACRWL}}$ can be restricted so that only well-typed signature morphisms and well-typed PT-algebras and homomorphisms are considered. To justify this assertion it is enough to notice that the composition of two well-typed signature morphisms is well-typed, and analogously for well-typed PT-homomorphisms. In addition, we must check that for all well-typed signature morphisms $\sigma : \Sigma \to \Sigma'$, $\mathbf{Mod}(\sigma)$ preserves well-typedness. Let then $\mathcal{A}'$ be a well-typed PT-algebra: Is $\mathcal{A}'_\sigma = \mathbf{Mod}(\sigma)(\mathcal{A})$ well-typed? By structural induction it is immediate to prove that $[\![\tau]\!]^{\mathcal{A}'_\sigma}\mu = [\![\sigma(\tau)]\!]^{\mathcal{A}'}\mu$ for any type τ over Σ and type valuation μ, hence it follows that

$$\begin{array}{rcl}\mathcal{E}^{\mathcal{A}'_\sigma}([\![\tau]\!]^{\mathcal{A}'_\sigma}\mu) & = & \{d \in D^{\mathcal{A}'_\sigma} \mid d :^{\mathcal{A}'_\sigma} [\![\tau]\!]^{\mathcal{A}'_\sigma}\mu\} \\ & = & \{d \in D^{\mathcal{A}'} \mid d :^{\mathcal{A}'} [\![\sigma(\tau)]\!]^{\mathcal{A}'}\mu\} \\ & = & \mathcal{E}^{\mathcal{A}'}([\![\sigma(\tau)]\!]^{\mathcal{A}'}\mu).\end{array}$$

Now, if $h : (\tau_1, \ldots, \tau_n) \to \tau_0 \in \Sigma$, and $d_i \in \mathcal{E}^{\mathcal{A}'_\sigma}([\![\tau_i]\!]^{\mathcal{A}'_\sigma}\mu)$, $1 \leq i \leq n$, by the above equality and the well-typedness of σ and $\mathcal{A}'$, we have

$$\begin{array}{rcl}h^{\mathcal{A}'_\sigma}(d_1, \ldots, d_n) & = & \sigma(h)^{\mathcal{A}'}(d_1, \ldots, d_n) \\ & \subseteq & \mathcal{E}^{\mathcal{A}'}([\![\sigma(\tau_0)]\!]^{\mathcal{A}'}\mu) \\ & = & \mathcal{E}^{\mathcal{A}'_\sigma}([\![\tau_0]\!]^{\mathcal{A}'_\sigma}\mu),\end{array}$$

which proves that $\mathcal{A}'_\sigma$ is well-typed. For F a well-typed PT-homomorphism, since $\mathcal{E}^{\mathcal{A}'_\sigma}(l) = \mathcal{E}^{\mathcal{A}'}(l)$ for all $l \in U^{\mathcal{A}'_\sigma} = U^{\mathcal{A}'}$ and $F = \mathbf{Mod}(\sigma)(F)$, the result is straightforward.

§16. Embedding CRWL in RL

We would like to relate the institutions $\mathcal{I}_{\mathrm{CRWL}}$ and $\mathcal{I}_{\mathrm{RL}}$ by means of a map of institutions $(\Phi, \alpha, \beta) : \mathcal{I}_{\mathrm{CRWL}} \to \mathcal{I}_{\mathrm{RL}}$ having nice properties, in such a way that it indicated that $\mathcal{I}_{\mathrm{CRWL}}$ could be considered as a subinstitution of $\mathcal{I}_{\mathrm{RL}}$. The formal definition of subinstitution appeared originally in [19] and has been further generalized in subsequent articles. One of those extensions was introduced by Meseguer in [22], where it is called an *embedding*. The only requirement imposed on a map of institutions $(\Phi, \alpha, \beta) : \mathcal{I} \to \mathcal{I}'$ to be an embedding is that for each $T \in \mathbf{Th}_{\mathcal{I}}$, the functor $\beta_T : \mathbf{Mod}'(\Phi(T)) \to \mathbf{Mod}(T)$ must be an equivalence of categories.

The goal of this section is to show that it does not exist an embedding from $\mathcal{I}_{\mathrm{CRWL}}$ into $\mathcal{I}_{\mathrm{RL}}$. For that it will be enough to find a categorical property which is preserved by an equivalence of categories and a theory $T \in \mathbf{Th}_{\mathrm{CRWL}}$ such that $\mathbf{Mod}_{\mathrm{RL}}(\Phi(T))$, but not $\mathbf{Mod}_{\mathrm{CRWL}}(T)$, has the property. In this case it will be proved that for any RL-theory $\mathcal{R}$ the category $\mathcal{R}$-$\mathbf{Sys}$ has equalizers, but a theory T will be shown in CRWL such that $\mathbf{Mod}_{\mathrm{CRWL}}(T)$ does not have them.

Let Σ be a signature with constructors such that $C_\Sigma = \emptyset$ and $F_\Sigma = F^0_\Sigma = \{f_1, f_2\}$, $\Gamma = \{f_2 \to x \Leftarrow f_1 \bowtie f_1\}$ a CRWL-program over Σ, and consider the CRWL-theory $T = (\Sigma, \Gamma)$. We define two CRWL-algebras over Σ: $\mathcal{A}$ given by

- the set $D^{\mathcal{A}} = \{\bot, a_1, a_2\}$ with partial order $\bot \sqsubseteq a_1 \sqsubseteq a_2$;
- the cones $f_1^{\mathcal{A}} = \langle a_1 \rangle$ and $f_2^{\mathcal{A}} = \langle \bot \rangle$;

and $\mathcal{B}$ with

- $D^{\mathcal{B}} = \{\bot, b_1\}$;
- the cones $f_1^{\mathcal{B}} = f_2^{\mathcal{B}} = \langle \bot \rangle$.

$\mathcal{A}, \mathcal{B} \in \mathbf{Mod}_{\mathrm{CRWL}}(T)$ trivially, because they do not satisfy the condition $f_1 \bowtie f_1$.

Let us now define two CRWL-homomorphisms $F, G : \mathcal{A} \to \mathcal{B}$ given by:

$$F(x) = \langle \bot \rangle \quad \text{and} \quad G(x) = \begin{cases} \langle \bot \rangle & \text{if } x = \bot, a_1 \\ \langle b_1 \rangle & \text{if } x = a_2. \end{cases}$$

Clearly, F and G preserve both f_1 and f_2, so that they are actually homomorphisms. We will prove that there does not exist the equalizer of F and G.

For let us suppose that $E : \mathcal{E} \to \mathcal{A}$ is such an equalizer and let $H : \mathcal{A} \to \mathcal{A}$ be a homomorphism given by

$$H(x) = \begin{cases} \langle \bot \rangle & \text{if } x = \bot \\ \langle a_1 \rangle & \text{if } x = a_1, a_2. \end{cases}$$

Then, there must exist a (unique) homomorphism $M : \mathcal{A} \to \mathcal{E}$ such that $E \circ M = H$. Let $e_1 \in \mathcal{E}$ such that $M(a_1) = \langle e_1 \rangle$ and $E(e_1) = \langle a_1 \rangle$. As E and M loosely preserve defined functions,

$$E(f_2^{\mathcal{E}}) \subseteq f_2^{\mathcal{A}} = \langle \bot \rangle$$

so that $e_1 \notin f_2^{\mathcal{E}}$, and

$$\langle e_1 \rangle = M(f_1^{\mathcal{A}}) \subseteq f_1^{\mathcal{E}}.$$

Therefore, as $\mathcal{E} \in \mathbf{Mod}_{\mathrm{CRWL}}(T)$, there must exist $e_2 \in \mathcal{E}$ such that $e_1 \sqsubset e_2$: otherwise, $\mathcal{E}$ would not satisfy Γ. Besides, due to the monotonicity of E and the equality $F \circ E = G \circ E$, it is $E(e_2) = \langle a_1 \rangle$. But then we have $M_1, M_2 : \mathcal{B} \to \mathcal{E}$ given by

$$M_1(x) = \begin{cases} \langle \bot \rangle & \text{if } x = \bot \\ \langle e_1 \rangle & \text{if } x = b_1 \end{cases} \quad \text{and} \quad M_2(x) = \begin{cases} \langle \bot \rangle & \text{if } x = \bot \\ \langle e_2 \rangle & \text{if } x = b_1, \end{cases}$$

two different homomorphisms satisfying $E \circ M_1 = E \circ M_2$, a contradiction with the universal property of equalizers.

In contrast with what happens in CRWL, the following proposition shows a construction for equalizers in RL.

16.1 Proposition *For all RL-theories $\mathcal{R} = (\Sigma, E, L, \Gamma)$, the category $\mathcal{R}$-**Sys** has equalizers.*

Proof. Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two $\mathcal{R}$-systems and let $F, G : \mathcal{S}_1 \to \mathcal{S}_2$ be two $\mathcal{R}$-homomorphisms between them; let us build their equalizer $\mathcal{E} \xrightarrow{E} \mathcal{S}_1$.

The objects in the category $\mathcal{E}$ are those $s \in \mathcal{S}_1$ such that $F(s) = G(s)$; the arrows, those $f : s \to s'$ in $\mathcal{S}_1$ such that $F(f) = G(f)$ (which implies, in particular, that F and G also coincide over s and s'); composition is that of $\mathcal{S}_1$. $\mathcal{E}$ is well defined because functors preserve identities and composition.

Next, we assign a (Σ, E)-algebra structure to $\mathcal{E}$. For each $f \in \Sigma_n$ we define $f_{\mathcal{E}} = f_{\mathcal{S}_1}|_{\mathcal{E}}$. Let us check that this is a valid definition. If $e_1, \ldots, e_n \in |\mathcal{E}|$ then

$$\begin{array}{rcll} F(f_{\mathcal{S}_1}(e_1, \ldots, e_n)) & = & f_{\mathcal{S}_2}(F(e_1), \ldots, F(e_n)) & (F \text{ is homomorphism}) \\ & = & f_{\mathcal{S}_2}(G(e_1), \ldots, G(e_n)) & (e_i \in |\mathcal{E}|) \\ & = & G(f_{\mathcal{S}_1}(e_1, \ldots, e_n)) & (G \text{ is homomorphism}) \end{array}$$

and thus $f_{\mathcal{S}_1}(e_1, \ldots, e_n) \in |\mathcal{E}|$. Analogously for arrows. With this definition it is easy to prove by structural induction that $t_{\mathcal{E}} = t_{\mathcal{S}_1}|_{\mathcal{E}^n}$ for all $t(x_1, \ldots, x_n) \in T_{\Sigma}(\mathcal{X})$. Therefore, for each $t = t' \in E$ it is $t_{\mathcal{E}} = t'_{\mathcal{E}}$.

The only thing missing in the definition of $\mathcal{E}$ are the natural transformations associated to the rewrite rules. Let

$$r : [t(\overline{x})] \to [t'(\overline{x})] \text{ } \mathit{if} \text{ } [a_1(\overline{x})] \to [b_1(\overline{x})] \wedge \ldots \wedge [a_m(\overline{x})] \to [b_m(\overline{x})]$$

be a rule in $\mathcal{R}$. We have to define a natural transformation

$$r_{\mathcal{E}} : t_{\mathcal{E}} \circ J_{\mathcal{E}} \Rightarrow t'_{\mathcal{E}} \circ J_{\mathcal{E}},$$

where $J_{\mathcal{E}} : \mathit{Subeq}((a_{j\mathcal{E}}, b_{j\mathcal{E}})_{1 \leq j \leq m}) \to \mathcal{E}^n$ is the subequalizer functor. Using the construction of Section 4 and the fact that $a_{j\mathcal{E}} = a_{j\mathcal{S}_1}|_{\mathcal{E}^n}$ and $b_{j\mathcal{E}} = b_{j\mathcal{S}_1}|_{\mathcal{E}^n}$, $1 \leq j \leq m$, it follows that $\mathit{Subeq}((a_{j\mathcal{E}}, b_{j\mathcal{E}})_{1 \leq j \leq m})$ is a subcategory of $\mathit{Subeq}((a_{j\mathcal{S}_1}, b_{j\mathcal{S}_1})_{1 \leq j \leq m})$ and that $J_{\mathcal{E}}$ is just the restriction of the corresponding $J_{\mathcal{S}_1}$. Then we can define $r_{\mathcal{E}}$ by simply restricting $r_{\mathcal{S}_1}$, which is obviously a natural transformation, and this finishes our construction of $\mathcal{E}$ as an $\mathcal{R}$-system.

Let us now move to the definition of E and the proof that it is an $\mathcal{R}$-homomorphism. E is simply the inclusion functor. If $f \in \Sigma_n$ and $e_1, \ldots, e_n \in |\mathcal{E}|$, then

$$E(f_{\mathcal{E}}(e_1, \ldots, e_n)) = f_{\mathcal{E}}(e_1, \ldots, e_n) = f_{\mathcal{E}}(E(e_1), \ldots, E(e_n)),$$

so E is a Σ-algebra homomorphism.

For a rewrite rule $r : [t(\overline{x})] \to [t'(\overline{x})]$ *if* $[a_1(\overline{x})] \to [b_1(\overline{x})] \wedge \ldots \wedge [a_m(\overline{x})] \to [b_m(\overline{x})]$ in Γ, is the natural transformation $E \circ r_{\mathcal{E}}$ equal to $r_{\mathcal{S}_1} \circ E^{\bullet}$? Let $(\overline{e}^n, \overline{u}^m) \in \mathit{Subeq}((a_{j\mathcal{E}}, b_{j\mathcal{E}})_{1 \le j \le m})$. Regarding $E^{\bullet}$, we only need to know that $E^{\bullet}(\overline{e}^n, \overline{u}^m) = (E^n(\overline{e}^n), E^m(\overline{u}^m))$. Now,

$$\begin{aligned}
(E \circ r_{\mathcal{E}})(\overline{e}^n, \overline{u}^m) &= E(r_{\mathcal{E}}(\overline{e}^n, \overline{u}^m)) \\
&= r_{\mathcal{E}}(\overline{e}^n, \overline{u}^m) \\
&= r_{\mathcal{S}_1}(\overline{e}^n, \overline{u}^m) \\
&= r_{\mathcal{S}_1}(E^n(\overline{e}^n), E^m(\overline{u}^m)) \\
&= r_{\mathcal{S}_1}(E^{\bullet}(\overline{e}^n, \overline{u}^m)) \\
&= (r_{\mathcal{S}_1} \circ E^{\bullet})(\overline{e}^n, \overline{u}^m),
\end{aligned}$$

so E is an $\mathcal{R}$-homomorphism.

We already know that $\mathcal{E}$ is an $\mathcal{R}$-system and that E is an $\mathcal{R}$-homomorphism; the one thing missing is the equalizer property. Let then $H : \mathcal{C} \to \mathcal{S}_1$ be an $\mathcal{R}$-homomorphism such that $F \circ H = G \circ H$; we have to find a unique $M : \mathcal{C} \to \mathcal{E}$ such that $E \circ M = H$. As E is the inclusion functor the uniqueness is clear, because the only possibility for all objects c and arrows u in $\mathcal{C}$ is $M(c) = H(c)$ and $M(u) = H(u)$. It remains to prove that this is a valid definition. First, because of the equality $F \circ H = G \circ H$ the image of H is included in $\mathcal{E}$ and M is well-defined; as H is a functor, so is M. Given $f \in \Sigma_n$ and $c_1, \ldots, c_n \in |\mathcal{C}|$, we have

$$\begin{aligned}
M(f_{\mathcal{C}}(c_1, \ldots, c_n)) &= H(f_{\mathcal{C}}(c_1, \ldots, c_n)) \\
&= f_{\mathcal{S}_1}(H(c_1), \ldots, H(c_n)) \\
&= f_{\mathcal{E}}(M(c_1), \ldots, M(c_n)),
\end{aligned}$$

and M is a Σ-algebra homomorphism. Finally, if $r : [t(\overline{x})] \to [t'(\overline{x})]$ *if* $[a_1(\overline{x})] \to [b_1(\overline{x})] \wedge \ldots \wedge [a_m(\overline{x})] \to [b_m(\overline{x})]$ is a rewrite rule in Γ and $(\overline{c}^n, \overline{u}^m)$ is an object of $\mathit{Subeq}((a_{j\mathcal{C}}, b_{j\mathcal{C}})_{1 \le j \le m})$, then

$$\begin{aligned}
r_{\mathcal{E}}(M^{\bullet}(\overline{c}^n, \overline{u}^m)) &= r_{\mathcal{E}}(M^n(\overline{c}^n), M^m(\overline{u}^m)) \\
&= r_{\mathcal{S}_1}(H^n(\overline{c}^n), H^m(\overline{u}^m)) \\
&= r_{\mathcal{S}_1}(H^{\bullet}(\overline{c}^n, \overline{u}^m)) \\
&= H(r_{\mathcal{C}}(\overline{c}^n, \overline{u}^m)) && (H \text{ is homomorphism}) \\
&= M(r_{\mathcal{C}}(\overline{c}^n, \overline{u}^m))
\end{aligned}$$

and we have $M \circ r_{\mathcal{C}} = r_{\mathcal{E}} \circ M^{\bullet}$. □

Note that, in the above proof, the equalizer $\mathcal{E}$ is a model of all the rewrite rules that $\mathcal{S}_1$ satisfies. Therefore the result is still valid when the RL-theory $\mathcal{R}$ is replaced by the category $\mathbf{Mod}_{\mathrm{RL}}(\Gamma)$ for some set Γ of rewrite rules in the institution $\mathcal{I}_{\mathrm{RL}}$. As an immediate consequence we have the following

16.2 Corollary $\mathcal{I}_{\mathrm{CRWL}}$ *is not embeddable in* $\mathcal{I}_{\mathrm{RL}}$.

Proof. Let T be the CRWL-theory defined at the beginning of this section. It has been shown that $\mathbf{Mod}_{\mathrm{CRWL}}(T)$ doesn't have all equalizers, whereas $\mathbf{Mod}_{\mathrm{RL}}(\Phi(T))$ has, regardless of the actual definition of Φ. Therefore, there cannot exist an equivalence of categories $\beta_T : \mathbf{Mod}_{\mathrm{RL}}(\Phi(T)) \to \mathbf{Mod}_{\mathrm{CRWL}}(\Phi)$. □

§17. Embedding RL in CRWL

In the previous section we have begun studying the relationship between CRWL and RL at the semantic level in a formal framework, namely, that of embedding of institutions. Of the two possible directions which can be explored we have just studied one. What about the other way around? Can we embed $\mathcal{I}_{\mathrm{RL}}$ in $\mathcal{I}_{\mathrm{CRWL}}$? When we began preparing this work our intuition was that we would be able to view CRWL as a "subpart" of RL in the first place, but also that the converse would not be true. The previous section has shown that our intuition was wrong about the first point and the purpose of this one is to deal with the second.

In order to prove that RL cannot be embedded in CRWL we have to find, in the same way as we did in Section 16, an RL-theory T such that $\mathbf{Mod}_{\mathrm{RL}}(T)$ has a categorical property that no category of models in CRWL has. In what follows two such theories are shown.

For the first one, note that for any CRWL-theory T there exists a CRWL-algebra $\mathcal{A} \in |\mathbf{Mod}_{\mathrm{CRWL}}(T)|$ with an infinite number of automorphisms. Simply consider $\mathcal{A}$ given by $D^{\mathcal{A}} = \{\bot, a, b_1, b_2, \ldots\}$ with $\bot \sqsubseteq a$, $\bot \sqsubseteq b_1 \sqsubseteq b_2 \sqsubseteq \ldots$, the image of all functions associated to constructor symbols to be $\langle a \rangle$, and the corresponding one of defined functions symbols to be $D^{\mathcal{A}}$. This way $\mathcal{A}$ is clearly a CRWL-algebra, satisfies all conditional rewrite rules, and the set $\{F_i : \mathcal{A} \to \mathcal{A}\}_{i \in \mathbb{N}}$, where

$$F_i(x) = \begin{cases} \langle \bot \rangle & \text{if } x = \bot \\ \langle a \rangle & \text{if } x = a \\ \langle b_i \rangle & \text{if } x = b_j \ (j \in \mathbb{N}) \end{cases}$$

is an infinite family of automorphisms of $\mathcal{A}$. On the other hand, in RL, if $\mathcal{R}$ is the RL-theory given by $(\{c\}, \{x = c\}, \emptyset, \emptyset)$ then, for all $\mathcal{R}$-systems $\mathcal{S}$, the equality

$id_{\mathcal{S}} = c_{\mathcal{S}}$, where $c_{\mathcal{S}}$ is a constant functor, forces $\mathcal{S}$ to be a category with just one object and one arrow, and no infinite family of homomorphisms can exist. Therefore (as an equivalence of categories is full and faithful), $\mathbf{Mod}_{\mathrm{RL}}(\mathcal{R})$ is not categorically equivalent to $\mathbf{Mod}_{\mathrm{CRWL}}(\Phi(\mathcal{R}))$, whatever Φ might be.

For the second one note that, if $\mathcal{R} = (\Sigma, E, L, \Gamma)$ is an RL-theory and there are no constants in Σ, then $\emptyset$ is the initial $\mathcal{R}$-system. In addition, for each other $\mathcal{R}$-system $\mathcal{S}$ we have $\emptyset \times \mathcal{S} = \emptyset$ (although we have not built products in RL, they follow the same structure as in **Cat**). In CRWL, however, the initial algebra $\mathcal{I}$ is never empty for any CRWL-theory ($\perp \in D^{\mathcal{I}}$, see the construction in [14]) and, if we choose a CRWL-algebra $\mathcal{A}$ whose underlying poset has a greater cardinality than that of $\mathcal{I}$ (generalizing, if necessary, the construction in the previous paragraph), we will have that $\mathcal{A} \times \mathcal{I}$ is not even isomorphic to $\mathcal{I}$. Since equivalences of categories should preserve both limits and colimits, we conclude the following

17.1 Proposition *$\mathcal{I}_{\mathrm{RL}}$ is not embeddable in $\mathcal{I}_{\mathrm{CRWL}}$.*

Proof. It follows from any of the two preceding paragraphs. □

Admittedly, although these two counterexamples formally solve the problem, they are so particular that they cannot be considered to truly reflect the real differences between RL and CRWL. Future work should concentrate on finding a more illustrative example.

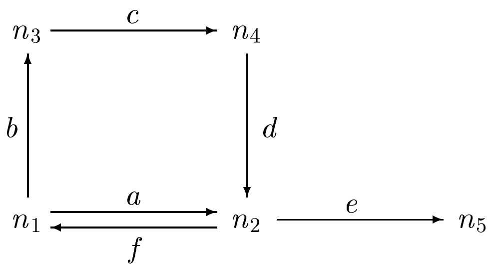
§18. The Maude Language

Maude is a high-level language and high-performance system supporting both equational and rewriting computation. *Functional modules* take charge of the equational part, whereas *system modules* provide the full power of RL. There is also a third kind of module in Maude, *object-oriented modules.* However, these can be reduced to system modules by a "desugaring" process so, in essence, we only have functional and system modules. In what follows we review the basic notions that will be needed to understand the program in Section 19; for a more detailed exposition with plenty of examples see [9, 8].

18.1. Functional Modules

Functional modules define data types and functions on them by means of equational theories whose equations are Church-Rosser and terminating. A mathematical model of the data and the functions is provided by the *initial algebra* defined by the theory, whose elements consist of equivalence classes of ground terms modulo the equations. Evaluation of any expression to its reduced form using the equations as rewrite rules assigns to each equivalence class a unique canonical representative.

The equational logic on which Maude is based is membership equational logic so, in addition to sorts, subsorts and overloading of functions, there are also (conditional) membership axioms. We illustrate all these points with the following example taken from [8]. Consider the following graph.



The module `PATH` forms path over this graph. As not all random sequences of edges constitute a path, concatenation is defined, not at the level of paths but at a supersort `Path?` and then, a conditional membership axiom is used to separate the correct paths.

```
fmod PATH is
  protecting MACHINE-INT .

  sorts  Edge Path Path? Node .
  subsorts Edge < Path < Path? .

  ops n1 n2 n3 n4 n5 : -> Node [ctor] .
  ops a b c d e f : -> Edge [ctor] .
  op _;_ : Path? Path? -> Path? [ctor assoc] .
  ops source target : Path -> Node .
  op length : Path -> MachineInt .

  var E : Edge . var P : Path .

  cmb (E ; P) : Path if target(E) == source(P) .

  ceq source(E ; P) = source(E) if E ; P : Path .
  ceq target(E ; P) = target(P) if E ; P : Path .
  eq length(E) = 1 .
  ceq length(E ; P) = 1 + length(P) if E ; P : Path .

  eq source(a) = n1 . eq target(a) = n2 .
```

```
  eq source(b) = n1 . eq target(b) = n3 .
  eq source(c) = n3 . eq target(c) = n4 .
  eq source(d) = n4 . eq target(d) = n2 .
  eq source(e) = n2 . eq target(e) = n5 .
  eq source(f) = n2 . eq target(f) = n1 .
endfm
```

The keyword `protecting` is used to import the predefined module `MACHINE-INT` defining the integers, without changing the data belonging to the sorts in the initial algebra of the imported module. Notice that this is a semantic condition and it is the user who must take care of its fulfilment. There also exists in Maude another mechanism for importation, using `including`, which allows changing these data. Next come the declaration of sorts and subsorts; the sort `Path?` is used as a kind of error supersort so that the concatenation operation is a total function. This operator is declared using “infix” syntax, with the underbars indicating the places where the first and second argument should be placed. Some of the operations have associated *equational attributes* written between brackets. `ctor` plays simply a methodological role, pointing out which operations will be used to construct or generate the data. The attribute `assoc` states that `_;_` is associative. The Maude engine then uses this information to rewrite modulo associativity. Others attribute available are `comm` and `id:`, for commutativity and specifying an identity element, respectively. Finally, membership axioms and equations are introduced by the keywords `mb` and `eq`, respectively, prefixed by a `c` were they conditional. The condition `if E ; P : Path` appearing in some of the equations is not really necessary, but it has been included to illustrate the use of conditional equations.

We can then use the command `reduce` (abbreviated `red`) to find the canonical form of some expressions. Notice the error result obtained when applying the operation `source` to a term not representing a correct path.

```
Maude> red (b ; c ; d) .
result Path: (b ; c ; d)

Maude> red length(b ; c ; d) .
result MachineInt: 3

Maude> red source(a ; b ; c) .
result Error(Node): source(a ; b ; c)
```

Besides `MACHINE-INT` there are other predefined modules in Maude which we will use later. Among them are `BOOL`, defining a sort `Bool` and the usual Boolean operations, and `QID`, where a sort `Qid` for quoted identifiers is declared.

18.2. System Modules

In functional modules, due to the confluence and terminating properties of the equations when interpreted as rewrite rules, the rewriting of a term always finishes, and this process can only lead to one value. However, for many interesting theories, not only have we infinite chains of rewritings but also not confluent paths.

System modules in Maude generalize functional modules and correspond to RL-theories: they specify the initial model of a rewrite theory. From a system perspective this model describes all the concurrent behaviours that the system so axiomatized can exhibit.

As an example of a system module, the following module `SORTING` for sorting vectors of integers is introduced, taken from [8].

```
mod SORTING is
  protecting MACHINE-INT .

  sorts Pair PairSet .
  subsort Pair < PairSet .

  op <_;_> : MachineInt MachineInt -> Pair [ctor] .
  op empty : -> PairSet [ctor] .
  op __ : PairSet PairSet -> PairSet [ctor assoc comm id: empty] .
  vars I J X Y : MachineInt .

  crl [sort] : < J ; X > < I ; Y > => < J ; Y > < I ; X >
         if (J < I) and (X > Y) .
endm
```

Using the `rewrite` (abbreviated `rew`) command, we can use Maude's default interpreter for executing expressions in system modules.

```
Maude> rew < 1 ; 3 > < 2 ; 2 > < 3 ; 1 > .
result PairSet: < 1 ; 1 > < 2 ; 2 > < 3 ; 3 >
```

This way, we have no control at all over the application of the rules in the system module. Although not a problem in this example, in many other cases the rewriting inference process could go in many undesired directions, so we would like to have good ways of controlling it by means of adequate *strategies*. Even in this case, we could use strategies to specify different sorting algorithms. In Maude, thanks to its reflective capabilities, strategies can be made internal to the system. The starting point for defining strategies is the module `META-LEVEL` with which we deal in the following section. A strategy specifying a depth-first search over a tree is presented in Section 19.

18.3. The `META-LEVEL` *module*

In Section 12 we commented that RL is reflective. In Maude, key functionality of the universal theory has been efficiently implemented in the functional module `META-LEVEL`. Furthermore, several other useful functions are also built-in for efficiency reasons. Some of the most important features supplied by the module `META-LEVEL`, and which we review in the next sections following [9], are:

- Maude terms are reified as elements of a data type `Term` of terms;
- Maude modules are reified as terms in a data type `Module` of modules;
- the processes of reducing a term to normal form in a functional module and of finding whether such a normal form has a given sort are reified by a function `meta-reduce`;
- the process of applying a rule of a system module to a subject term is reified by a function `meta-apply`;
- the process of rewriting a term in a system module using Maude's default interpreter is reified by a function `meta-rewrite`.

For more details see [9, 8].

18.3.1 Representing terms

Terms are reified as elements of the data type `Term` of terms, by means of the following operations:

```
subsort Qid < Term .
subsort Term < TermList .
op {_}_ : Qid Qid -> Term [ctor] .
op _[_] : Qid TermList -> Term [ctor] .
op _,_ : TermList TermList -> TermList [ctor assoc] .
op _:_ : Term Qid -> Term [ctor] .
op error* : -> Term .
```

The first declaration, making `Qid` a subsort of `Term`, is used to represent variables by the corresponding quoted identifiers. Thus, the variable `N` is represented by `'N`. The operation `{_}_` is used for representing constants as pairs, with the first argument the constant in quoted form, and the second argument the sort of the constant, also in quoted form. For example, the constant `n1` of sort `Node` in the module `PATH` in Section 18.1 is represented as `{'n1}'Node`. The operation `_[_]` corresponds to the recursive construction of terms out of subterms, with the first argument the top operation in quoted form, and the second argument the list of its subterms, where list concatenation is denoted `_,_`. For example, the term `source(c ; d)` of sort `Node` in module `PATH` is metarepresented as the following term (also denoted mathematically $\overline{\texttt{source(c ; d)}}$) of sort `Term` in module `META-LEVEL`:

```
'source['_;_[{'c}'Edge, {'d}'Edge]]
```

Note that, since terms in the module `META-LEVEL` can be metarepresented just as terms in any other module, the representation of terms can be iterated.

Membership predicates $t : s$ are meta-represented by means of the binary constructor `_:_` with the first argument the representation of the term, and the second the representation of the sort as a quoted identifier. The last declaration above for the data type of terms is a constant `error*` to be used as an error element.

18.3.2 Representing modules

Functional and system modules are metarepresented in a syntax very similar to their original user syntax. The main differences are that: (1) terms in equations, membership axioms, and rules are now metarepresented as we have already explained; (2) in the metarepresentation of modules we follow a *fixed order* in introducing the different kinds of declarations for sorts, subsorts, variables, equations, etc., whereas in the user syntax there is considerable flexibility for introducing such different declarations in an interleaved and piecemeal way; and (3) sets of identifiers—used in declarations of sorts—are represented as sets of quoted identifiers built with an associative and commutative operation `_;_`.

The module `META-LEVEL` provides sorts and constructors for each of the declarations that can appear in modules. For example, we have sorts `OpDecl` and `OpDeclSet` to represent declarations of operations and sets of declarations of operations, respectively. If there are no operation declarations in a particular module, then this argument will be `none`. Otherwise, the set is given by the associative and commutative constructor `__`, which is declared with identity element `none`. Each of the operation declarations is then given by a term of sort `OpDecl` using the constructor

```
op op_:_->_[_]. : Qid QidList Qid AttrSet -> OpDecl [ctor] .
```

The last argument of this constructor corresponds to the set of attributes of an operation.

The `META-LEVEL` syntax for the top-level operations representing functional and system modules is as follows:

```
  op fmod_is_______endfm : Qid ImportList SortDecl
        SubsortDeclSet OpDeclSet
        VarDeclSet MembAxSet EquationSet -> Module [ctor] .

  op mod_is________endm : Qid ImportList SortDecl
        SubsortDeclSet OpDeclSet
        VarDeclSet MembAxSet EquationSet RuleSet -> Module [ctor] .
```

To illustrate the general syntax for representing modules, we use a module `NAT` for natural numbers with zero, successor, and commutative addition (this example is borrowed from [9]). We first give `NAT`'s ordinary Maude syntax:

```
fmod NAT is
  sorts Zero Nat .
  subsort Zero < Nat .
  op 0 : -> Zero [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat [comm] .
  vars N M : Nat .
  eq 0 + N = N .
  eq s(N) + M = s(N + M) .
endfm
```

The representation $\overline{\texttt{NAT}}$ of `NAT` as a term of sort `Module` in `META-LEVEL` is

```
fmod 'NAT is
  nil
  sorts 'Zero ; 'Nat .
  subsort 'Zero < 'Nat .
  op '0 : nil -> 'Zero [ctor] .
  op 's : 'Nat -> 'Nat [ctor] .
  op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
  var 'N : 'Nat .
  var 'M : 'Nat .
  none
  eq '_+_[{'0}'Nat, 'N] = 'N .
  eq '_+_['s['N], 'M] = 's['_+_['N, 'M]] .
endfm
```

Since `NAT` has no list of imported submodules and no membership axioms, those fields are filled by the constant `nil` of sort `ImportList`, and the constant `none` of sort `MembAxSet`, respectively. Similarly, `none` is the empty set of attributes for operations that have no attributes.

Note that—just as in the case of terms—terms of sort `Module` can be metarepresented again, yielding then a term of sort `Term`, and this can be iterated an arbitrary number of times.

18.3.3 Descent functions

The module `META-LEVEL` has three built-in operations for meta-evaluation, also called *descent functions*: `meta-reduce`, `meta-rewrite` and `meta-apply`.

The operation `meta-reduce` has the following declaration, where we have omitted the lengthy list of bindings to C++ code in its declaration as a `special` built-in operation:

```
  op meta-reduce : Module Term -> Term [special (...) ] .
```

It takes as arguments the representations of a module and of a term t in that module, and returns the representation of the canonical form of the term t using the equations in the module, e.g,

```
Maude> red meta-reduce(N̅A̅T̅,s̅(̅0̅)̅ ̅+̅ ̅0̅) .
result Term: s̅(̅0̅)̅
```

The operation `meta-rewrite` is declared as

```
  op meta-rewrite : Module Term MachineInt -> Term [special (...) ] .
```

It is entirely analogous to `meta-reduce`, but instead of using only the equational part of a module it now uses both the equations and the rules to rewrite the term using Maude's default strategy. Its first two arguments are the representations of a module $\mathcal{R}$ and of a term t, and its third argument is a positive machine integer n. Its result is the representation of the term obtained from t after at most n applications of the rules in $\mathcal{R}$ using the strategy of Maude's default interpreter. When the value 0 is given as the third argument, no bound is given to the number of rewrites, and rewriting proceeds to the bitter end.

The operation `meta-apply` is declared as

```
op meta-apply : Module Term Qid Substitution
                MachineInt -> ResultPair [special (...) ] .
```

The first four arguments are representations in `META-LEVEL` of a module $\mathcal{R}$, a term t in $\mathcal{R}$, a label l of some rules[3] in $\mathcal{R}$, and a set of assignments (possibly empty) defining a partial substitution σ for the variables in those rules. The last argument is a natural number n. `meta-apply` then returns a pair, of sort `ResultPair`, consisting of a term and a substitution. The operation `meta-apply` is evaluated as follows:

1. the term t is first fully reduced using the equations in $\mathcal{R}$;
2. the resulting term is matched (at the top, with extension) against all rules with label l partially instantiated with σ, with matches that fail to satisfy the condition of their rule discarded;
3. the first n successful matches are discarded; if there is an $(n+1)$th match, its rule is applied using that match and the steps 4 and 5 below are taken; otherwise the pair `{error*, none}` is returned, where `none` denotes the identity substitution;
4. the term resulting from applying the given rule with the $(n+1)$th match is fully reduced using the equations in $\mathcal{R}$;
5. the pair formed using the constructor `{_,_}` whose first element is the representation of the resulting fully reduced term and whose second element is the representation of the match used in the reduction is returned.

§19. A Maude Interpreter for CRWL

The goal of this section is to write a Maude program to interpret CRWL. In Section 7 a translation from CRWL to RL was given and proved to be correct; however, we will not use it here. In the first place, that translation makes use of rules which include rewriting in the conditional part, and this is a feature not currently supported in Maude. In addition, the calculus BRC simulated there is a logical calculus, useful for proving statements and for theoretical reasoning, but unsuitable for solving goals. Instead, here we will employ the Constructor-based Lazy Narrowing Calculus (CLNC) which is used in [14] to define the operational semantics of CRWL, and that we briefly present now.

---

[3] A user can declare several rules with the same label; then `meta-apply` will try to apply some rule with that label.

19.1. The CLNC-calculus

Let Γ be a CRWL-program over a signature Σ. An *admissible goal* for Γ has the form $G \equiv \exists\overline{u}.\ S \,\square\, P \,\square\, E$, where:

- $evar(G) \equiv \overline{u}$ is the set of so-called existential variables of the goal G.
- $S \equiv x_1 = s_1, \dots, x_n = s_n$, with $x_i \in \mathcal{X}$, $s_i \in Term(\Sigma, \mathcal{X})$, is a set of equations, called *solved part*.
- $P \equiv e_1 \to t_1, \dots, e_k \to t_k$ is a multiset of approximation statements, with t_i total, called *produced part*. $pvar(P) = var(t_1) \cup \dots \cup var(t_1)$ is the set of *produced variables* of the goal G.
- $E \equiv a_1 \bowtie b_1, \dots, a_m \bowtie b_m$ is a multiset of joinability statments, and $dvar(E) = \{x \in \mathcal{X} \mid x \equiv a_i \text{ or } x \equiv b_i, \text{ for some } 1 \leq i \leq m\}$ is called the set of *demanded variables* of the goal G.

Besides, admissible goals must satisfy a number of technical conditions, the details of which will not concern us here and can be found in [14].

It is assumed, by convention, that in an *initial goal* G only the joinability part is non-empty. If $G \equiv \exists\overline{u}.\ S \,\square\, P \,\square\, E$ is an admissible goal and θ is a substitution mapping variables to partial terms, θ is a solution for G if, roughly, $x_i\theta = s_i\theta$ for all $x_i = s_i \in S$ and the statements in $(P \,\square\, E)\theta$ are provable. Again, some technical details are missing. With all these definitions, the following result is proved in [14].

19.1 Theorem *Let Γ be a program, G an initial goal, θ a substitution mapping variables to partial terms. Then the following statements are equivalent:*

(a) θ is a solution for G.

(b) $\Gamma \vdash_{\mathrm{CRWL}} G\theta$.

(c) $(\mathcal{A}, \eta) \models G\theta$, for all $\mathcal{A} \models \Gamma$ and η a totally defined valuation. □

Finally, the calculus CLNC for solving goals is presented. It consists of a set of transformation rules, each of which has the form $G \Vdash G'$, specifying one of the possible ways of performing one step of goal solving. Derivations are sequences of $\Vdash$-steps. Joinability statements are considered to be symmetric for the purposes of the calculus. The notation $svar(e)$ in some of the rules stands for the set of all *safe* variables occurring in e at some position whose ancestor positions are all occupied by constructors. We also use $c \in C_\Sigma$, $f \in F_\Sigma$ and $\{e/x\}$ for the substitution mapping the variable x to e.

The rules of the calculus can be divided into three big groups. First of all, the rules for $\bowtie$:

DC1 Decomposition:
$\exists \overline{u}.\ S \Box P \Box c(\overline{a}) \bowtie c(\overline{b}), E \ \Vdash\ \exists \overline{u}.\ S \Box P \Box \ldots, a_i \bowtie b_i, \ldots, E.$

ID Identity:
$\exists \overline{u}.\ S \Box P \Box x \bowtie x, E \ \Vdash\ \exists \overline{u}.\ S \Box P \Box E$
if $x \notin \mathit{pvar}(P)$.

BD Binding:
$\exists \overline{u}.\ S \Box P \Box x \bowtie s, E \ \Vdash\ \exists \overline{u}.\ x = s, (S \Box P \Box E)\sigma$
if $s \in \mathit{Term}(\Sigma, \mathcal{X})$, $\mathit{var}(s) \cap \mathit{pvar}(P) = \emptyset$, $x \notin \mathit{var}(s)$, $x \notin \mathit{pvar}(P)$, where $\sigma = \{s/x\}$.

IM Imitation:
$\exists \overline{u}.\ S \Box P \Box x \bowtie c(\overline{e}), E \ \Vdash\ \exists \overline{y}, \overline{u}.\ x = c(\overline{y}), (S \Box P \Box \ldots, y_i \bowtie e_i, \ldots, E)\sigma$
if $c(\overline{e}) \notin \mathit{Term}(\Sigma, \mathcal{X})$ or $\mathit{var}(c(\overline{e})) \cap \mathit{pvar}(P) \neq \emptyset$, and $x \notin \mathit{pvar}(P)$, $x \notin \mathit{svar}(c(\overline{e}))$, where $\sigma = \{c(\overline{y})/x\}$, $\overline{y}$ are new variables.

NR1 Narrowing:
$\exists \overline{u}.\ S \Box P \Box f(\overline{e}) \bowtie a, E \ \Vdash\ \exists \overline{y}, \overline{u}.\ S \Box \ldots, e_i \to t_i, \ldots, P \Box C, r \bowtie a, E$
where $R : f(\overline{t}) \to r \Leftarrow C$ is a variant of a rule in Γ, with $\overline{y} = \mathit{var}(R)$ new variables.

The rules associated to $\to$:

DC2 Decomposition:
$\exists \overline{u}.\ S \Box c(\overline{e}) \to c(\overline{t}), P \Box E \ \Vdash\ \exists \overline{u}.\ S \Box \ldots, e_i \to t_i, \ldots, P \Box E.$

OB Output Binding:

OB1 $\exists \overline{u}.\ S \Box x \to t, P \Box E \ \Vdash\ \exists \overline{u}.\ x = t, (S \Box P \Box E)\sigma$
if $t \notin \mathcal{X}$, $x \notin \mathit{pvar}(P)$, where $\sigma = \{t/x\}$.

OB2 $\exists x, \overline{u}.\ S \Box x \to t, P \Box E \ \Vdash\ \exists \overline{u}.\ S \Box (P \Box E)\sigma$
if $t \notin \mathcal{X}$, $x \in \mathit{pvar}(P)$, where $\sigma = \{t/x\}$.

IB Input Binding:
$\exists x, \overline{u}.\ S \Box t \to x, P \Box E \ \Vdash\ \exists \overline{u}.\ S \Box (P \Box E)\sigma$
if $t \in \mathit{Term}(\Sigma, \mathcal{X})$, where $\sigma = \{t/x\}$.

IIM Input Imitation:
$\exists x,\overline{u}.\ S \,\Box\, c(\overline{e}) \to x, P \,\Box\, E \ \Vdash\ \exists \overline{y},\overline{u}.\ S \,\Box (\ldots, e_i \to y_i, \ldots, P \,\Box\, E)\sigma$
if $c(\overline{e}) \notin \mathit{Term}(\Sigma, \mathcal{X})$, $x \in \mathit{dvar}(E)$, where $\sigma = \{c(\overline{y})/x\}$, $\overline{y}$ new variables.

EL Elimination:
$\exists x,\overline{u}.\ S \,\Box\, e \to x, P \,\Box\, E \ \Vdash\ \exists \overline{u}.\ S \,\Box\, P \,\Box\, E$
if $x \notin \mathit{var}(P \,\Box\, E)$.

NR2 Narrowing:
$\exists \overline{u}.\ S \,\Box\, f(\overline{e}) \to t, P \,\Box\, E \ \Vdash\ \exists \overline{y},\overline{u}.\ S \,\Box \ldots, e_i \to t_i, \ldots, r \to t, P \,\Box\, C, E$
if $t \notin \mathcal{X}$ or $t \in \mathit{dvar}(E)$, where $R : f(\overline{t}) \to r \Leftarrow C$ is a variant of a rule in Γ, with $\overline{y} = \mathit{var}(R)$ new variables.

Finally, the failure rules:

CF1 Conflict:
$\exists \overline{u}.\ S \,\Box\, P \,\Box\, c(\overline{a}) \bowtie d(\overline{b}), E \ \Vdash\ \mathit{FAIL}$ if $c \not\equiv d$.

CY Cycle:
$\exists \overline{u}.\ S \,\Box\, P \,\Box\, x \bowtie a, E \ \Vdash\ \mathit{FAIL}$ if $x \not\equiv a$, $x \in \mathit{svar}(a)$.

CF2 Conflict:
$\exists \overline{u}.\ S \,\Box\, c(\overline{a}) \to d(\overline{t}), P \,\Box\, E \ \Vdash\ \mathit{FAIL}$ if $c \not\equiv d$.

Goals of the form *FAIL* or $\exists \overline{u}.\ S \,\Box\ \Box$ are $\Vdash$-irreducible and are called *solved goals*. Associated to the goal $\exists \overline{u}.\ S \,\Box\ \Box$ there is a substitution $\sigma_S = \{t_1/x_1, \ldots, t_n/x_n\}$, which is a solution of it. Taking these solutions as the answers provided by the calculus, it turns out that CLNC is sound and complete.

19.2. Simulating CLNC

Taking advantage of the good properties of rewriting logic (and therefore, of Maude), given a CRWL-theory $T = (\Sigma, \Gamma)$, it is relatively straightforward to translate the CLNC-rules into a system module in Maude in which to solve admissible goals about T. This translation has got some drawbacks, however, and we will be forced to adopt a different point of view to make things work in a satisfactory manner.

19.2.1 Working at the object level

We will present the main ideas behind this translation by means of an example, that of the well-known theory of the natural numbers with addition and multiplication.

The symbols in the signature Σ will be those of $C^0_\Sigma = \{\mathit{zero}\}$, $C^1_\Sigma = \{s\}$ and $F^2_\Sigma = \{+, *\}$, while the rewrite rules in Γ will be (in infix form)

$$\begin{array}{ll} e_1 + \mathit{zero} \to e_1 & e_1 * \mathit{zero} \to \mathit{zero} \\ e_1 + s(e_2) \to s(e_1 + e_2) & e_1 * s(e_2) \to (e_1 * e_2) + e_1. \end{array}$$

Our first task will be to represent CRWL variables in Maude. Simply trying to use Maude's own variables will not work, as we will need to treat them as objects which can be manipulated and, in particular, some of the rules will force us to create fresh variables. In a similar way as it is done in [9] we can then declare

```
mod NAT-CRWL is
  protecting MACHINE-INT .

  sorts Var VarSet .
  subsort Var < VarSet .

  op v : MachineInt -> Var [ctor] .
  op empty-var : -> VarSet [ctor].
  op _U_ : VarSet VarSet -> VarSet [ctor assoc comm id: empty-var] .
  op intersection : VarSet VarSet -> VarSet .
  op _in_ : Var VarSet -> Bool .
  op new : VarSet -> Var .
  op maxIndex : VarSet -> MachineInt .
```

Variables are then represented as terms of the form `v(`n`)`, n being an integer. Next we extend the module so as to represent elements in $\mathit{Term}(\Sigma, \mathcal{X})$ and $\mathit{Expr}(\Sigma, \mathcal{X})$ too. Since it is important to distinguish whether an expression is also a term, a subsort declaration is used.

```
  sorts Term Expr .
  subsort Var < Term < Expr .

  op zero : -> Term [ctor] .
  op s : Expr -> Expr [ctor] .
  op s : Term -> Term [ctor] .
  ops + * : Expr Expr -> Expr [ctor] .
```

Note that constructor symbols are overloaded, so that they will produce terms when applied to terms. As `Term < Expr`, this is not necessary when the arity is 0. `+` and `*` have a `ctor` attribute despite being function symbols because they are used to *construct* the expressions. The sorts and operations for goals are introduced in a similar way.

```
sorts Equation SolvedPart ApproxStatement ProducedPart
      JoinStatement JoinPart Goal .

op goal : VarSet SolvedPart ProducedPart JoinPart -> Goal [ctor] .
op FAIL : -> Goal [ctor] .

subsort Equation < SolvedPart .
op eq : Var Term -> Equation [ctor] .
op empty-eq : -> SolvedPart [ctor] .
op __ : SolvedPart SolvedPart -> SolvedPart
                                     [ctor assoc comm id: empty-eq] .
var EQ : Equation .
eq EQ EQ = EQ . *** the solved part is a set

subsort ApproxStatement < ProducedPart .
op approx : Expr Term  -> ApproxStatement [ctor] .
op empty-approx : -> ProducedPart [ctor] .
op __ : ProducedPart ProducedPart -> ProducedPart
                                 [ctor assoc comm id: empty-approx] .

subsort JoinStatement < JoinPart .
op join : Expr Expr -> JoinStatement [ctor comm] .
op empty-join : -> JoinPart [ctor] .
op __ : JoinPart JoinPart -> JoinPart [ctor assoc comm id: empty-join] .
```

Notice the attribute `comm` associated to the operation `join`. The syntax used reflects faithfully that of CLNC, but for one important detail. Goals in CLNC are prefixed by a set $\overline{u}$ of existential variables; however, these variables play no role at all in the application of the rules, so they can be omitted. The first argument associated to the operation `goal`, then, is not used to represent this set, but the set of variables that have been employed up to a certain moment in the derivation process. (In fact, what is really needed are the current variables in the goal, but keeping this set up to date would be too expensive.) Its purpose is to feed the `new` function to create fresh variables. (Another possibility would be to remove this argument and to define an operation to find the variables in a goal.) Then, for example, the goal

$$\exists y.\ x = zero \,\square\, zero + s(zero) \to y \,\square\, s(s(zero)) \bowtie z, zero \bowtie zero * zero$$

is represented as follows,

```
goal(v(0) U v(1) U v(2),
     eq(v(0),s(zero)),
     approx(+(zero,s(zero),v(1)),
     join(s(s(zero)),v(2)) join(zero,*(zero,zero)))
```

where the first argument must satisfy the condition that it contains all variables appearing in the goal.

Before proceeding to translate the rules of the calculus into Maude it will be necessary to define some auxiliary operations entrusted with calculating variables in expressions and performing substitutions. For example, the following declarations and equations are used for extracting the safe variables in an expression:

```
var V : Var .
var E E' : Expr .

op safeVar : Expr -> VarSet .
eq safeVar(V) = V .
eq safeVar(zero) = empty-var .
eq safeVar(s(E)) = safeVar(E) .
eq safeVar(+(E,E')) = empty-var .
eq safeVar(*(E,E')) = empty-var .
```

The rest of the operations needed are

```
op variables : Expr -> VarSet .
op variables : ProducedPart -> VarSet .
op variables : JoinPart -> VarSet .

op producedVar : ProducedPart -> VarSet .

op demandVar : JoinPart -> VarSet .

op subst : Expr Var Term -> Expr .
op subst : SolvedPart Var Term -> SolvedPart .
op subst : ProducedPart Var Term -> ProducedPart .
op subst : JoinPart Var Term -> JoinPart .
```

Now we are ready to finish the translation. Some of the CLNC-rules are independent of the theory and, with all the machinery developed by now, easily encoded. It is the case of **ID** and **BD**, for example.

```
var UVars : VarSet .
var T : Term .          vars E1 E2 : Expr .
var SP : SolvedPart .   var PP : ProducedPart .
var JP : JoinPart .

crl [ID] : goal(UVars,SP,PP,join(V,V) JP) => goal(UVars,SP,PP,JP)
```

```
            if not (V in producedVar(PP)) .

  crl [BD] : goal(UVars,SP,PP,join(V,T) JP) =>
             goal(UVars,eq(V,T) subst(SP,V,T),subst(PP,V,T),subst(JP,V,T))
             if intersection(variables(T),producedVar(PP)) == empty-var
                and not V in (variables(T) U producedVar(PP)) .
```

Others are theory dependent (either on the signature or on the program rules) and may have to be translated by a number of rewrite rules. For example:

```
  rl [NR1] : goal(UVars,SP,PP,join(+(E1,E2),E) JP) =>
             goal(new(UVars) U UVars,SP,
                  approx(E1,new(UVars)) approx(E2,zero) PP,
                  join(new(UVars),E) JP) .
  rl [NR1] : goal(UVars,SP,PP,join(+(E1,E2),E) JP) =>
             goal(new(new(UVars) U UVars) U UVars,SP,
                  approx(E1,new(UVars))
                      approx(E2,s(new(new(UVars) U UVars))) PP,
                  join(s(+(new(UVars),new(new(UVars) U UVars))),E) JP) .
  rl [NR1] : goal(UVars,SP,PP,join(*(E1,E2),E) JP) =>
             goal(new(UVars) U UVars,SP,
                  approx(E1,new(UVars)) approx(E2,zero) PP,
                  join(zero,E) JP) .
  rl [NR1] : goal(UVars,SP,PP,join(*(E1,E2),E) JP) =>
             goal(new(new(UVars) U UVars) U UVars,SP,
                  approx(E1,new(UVars))
                       approx(E2,s(new(new(UVars) U UVars))) PP,
                  join(+(*(new(UVars),new(new(UVars) U UVars)),
                         new(UVars)),E) JP) .

  rl [DC2] : goal(UVars,SP,approx(zero,zero) PP,JP) =>
             goal(UVars,SP,PP,JP) .
  rl [DC2] : goal(UVars,SP,approx(s(E),s(T)) PP,JP) =>
             goal(UVars,SP,approx(E,T) PP,JP) .

  crl [IIM] : goal(UVars,SP,approx(s(E),V) PP,JP) =>
              goal(new(UVars) U UVars,SP,
                   subst(approx(E,new(UVars)) PP,V,s(new(UVars))),
                   subst(JP,V,s(new(UVars))))
              if not (s(E) : Term) and (V in demandVar(JP)) .

  rl [CF2] : goal(UVars,SP,approx(zero,s(E)) PP,JP) => FAIL .
  rl [CF2] : goal(UVars,SP,approx(s(E),zero) PP,JP) => FAIL .
```

And so we are done; the complete program can be found in Appendix A.

It is worth noting how easily and faithfully we have been able to simulate CLNC in the rewriting logic framework, a fact supporting the claims made in [18]. Now we can ask Maude to solve goals for us and be confident that it will find a solution, can't we? Let us try with a simple one.

```
Maude> rew goal(v(0), empty-eq, empty-approx,
                join(*(s(zero),v(0)), zero)) .
result Goal: goal(v(0) U v(1), eq(v(0), zero), empty-approx, empty-join)
```

The resulting goal is in solved form and its solved part gives indeed the correct solution; the extra variable `v(1)` which appears in the first argument is created by the application of the third rule associated to **NR1**. So everything seems to work properly. However, let us simply interchange the positions of the factors in the goal.

```
Maude> rew goal(v(0), empty-eq, empty-approx,
                join(*(v(0),s(zero)),zero)) .
result Goal: FAIL
```

What has happened? Well, the calculus happens to be nondeterministic and, although mainly of a *don't care* nature, this is not the case for the rules **NR1** and **NR2**. So, when Maude faces this goal it applies the third of the rules corresponding to **NR1**, getting a new goal in which the (representation of the) approximation statement $s(zero) \to zero$ appears and then, by the rule **CF2**, we are led to the *FAIL* goal. As Maude does not have a built-in backtracking strategy, the rewriting process is stopped and no solution is found.

Another problem with this approach is that it is too particular: for every CRWL-theory we have to build a whole new module. In addition, we would like to supply just the symbols and the program rules and leave the hard work of building the module to Maude.

19.2.2 Working at the metalevel

All the remarks pointed out at the end of the previous section lead us to adopt a different perspective. Instead of writing for each CRWL-theory a *system* module that simulates its operational semantics, we are going to write a unique *functional* module which, given a signature and a program, returns the *metarepresentation* of such a system module. This way the difficulties previously mentioned are overcome: the module now works for every CRWL-theory and, as we get a module metarepresented, we can apply to it the strategy of our choice in order to explore the derivation tree generated by the rewriting relation.

The only information our module will need to be given is the list of constructors, the list of function symbols and the list of program rules. Constructor and function symbols are given by pairs consisting of a `Qid` for the symbol and a `MachineInt` for its arity. For rules, the metarepresentation (using Maude syntax) of every expression appearing in it is required.

```
fmod MAP-PHI is
  protecting META-LEVEL .
  protecting MACHINE-INT .
  protecting BOOL .

  sorts Constructor ConstructorL Function FunctionL
        RuleCRWL RuleCRWLL CRWLTheory
        PairTerm PairTermL .

  op cSymbol : Qid MachineInt -> Constructor [ctor].
  op nilC : -> ConstructorL [ctor] .
  op consC : Constructor ConstructorL -> ConstructorL [ctor] .

  op fSymbol : Qid MachineInt -> Function [ctor] .
  op nilF : -> FunctionL [ctor] .
  op consF : Function FunctionL -> FunctionL [ctor] .

  op <_;_> : Term Term -> PairTermL [ctor] .
  op nilPT : -> PairTermL [ctor] .
  op consPT : PairTerm PairTermL -> PairTermL [ctor] .
  op _->_<=_ : Term Term PairTermL -> RuleCRWL [ctor] .
  op nilR : -> RuleCRWLL [ctor] .
  op consR : RuleCRWL RuleCRWLL -> RuleCRWLL [ctor] .

  op crwlth : ConstructorL FunctionL RuleCRWLL -> CRWLTheory [ctor] .
```

Recalling the natural numbers theory of Section 19.2.1 and removing, for the sake of simplicity, the two program rules defining multiplication, we would have the following term representing it:

```
  crwlth(consC(cSymbol('zero, 0), consC(cSymbol('s, 1), nilC)),
         consF(fSymbol('+,2), consF(fSymbol('*, 2), nilF)),
         consR('+['v[{'0}'MachineInt], {'zero}'Term] ->
             'v[{'0}'MachineInt] <= nilPT,
            consR('+['v[{'0}'MachineInt],'s['v[{'1}'MachineInt]]] ->
                  's['+['v[{'0}'MachineInt],'v[{'1}'MachineInt]]]
                   <= nilPT,
            nilR)))
```

As this very simple example shows, the input to the module has to be given in a very cumbersome way. Improving it would require us to build a parser for expressions in CRWL and, even though Maude offers tools for doing it in a simple way (see [8]), that is not our main objective. Similarly, the lists of constructor and functions symbols are not checked to see whether there are repeated elements in them, and expressions of the form `crwlth(...)` are not verified to see that they really represent well defined CRWL-theories either. Again, that is not our main concern.

The main operation in the module is `phi` which is responsible for activating all the necessary operations to generate the metarepresentation of the module.

```
op phi : CRWLTheory -> Module .

var CL : ConstructorL .  var FL : FunctionL .
var RL : RuleCRWLL .
var ODS : OpDeclSet .    var VDS : VarDeclSet .
var ES : EquationSet .
var Q Q0 : Qid .         var M N N0 : MachineInt .
var T1 T2 : Term .       var TL1 TL2 : TermList .

eq phi(crwlth(CL,FL,RL)) =
   (mod 'CRWLTHEORY is
     including 'MACHINE-INT .
     including 'BOOL .
     sortCRWL
     subsortCRWL
     addC(CL,addF(FL,opCRWL))
     addVar(max(maxC(CL),maxF(FL)),varCRWL)
     none
     addEqC(CL,addEqF(FL,eqCRWL))
     (ruleCRWL ruleDC1(CL) ruleIM(CL) ruleNR1(RL)
               ruleDC2(CL) ruleIIM(CL) ruleNR2(RL)
               ruleCF1(CL) ruleCF2(CL))
   endm) .
```

Sort and subsort declarations are independent of the theory and we can use two constants to include all of them.

```
op sortCRWL : -> SortDecl .
op subsortCRWL : -> SubsortDeclSet .

eq sortCRWL = (sorts 'Var ; 'VarSet ; 'Term ; 'Expr ; 'Equation ;
                     'SolvedPart ; 'ApproxStatement ; 'ProducedPart ;
                     'JoinStatement ; 'JoinPart ; 'Goal .) .
```

```
eq subsortCRWL = (subsort 'Var < 'VarSet .)(subsort 'Var < 'Term .)
      (subsort 'Term < 'Expr .) (subsort 'Equation < 'SolvedPart .)
      (subsort 'ApproxStatement < 'ProducedPart .)
      (subsort 'JoinStatement < 'JoinPart .) .
```

Recalling the system module **NAT-CRWL** associated to the natural numbers in Section 19.2.1 (which will be convenient to bear in mind from now on), it can be observed that these constants are simply assigned the metarepresentation of the declarations present there.

In all the other cases we will have to distinguish a common part and a theory dependent part. For operations, the common part is formed by the auxiliary functions for substitutions, those dealing with variables, and the ones used to represent goals. All of these are grouped together in a constant.

```
op opCRWL : -> OpDeclSet .

eq opCRWL =
     (op 'v : 'MachineInt -> 'Var [ctor] .)
     (op 'empty-var : nil -> 'VarSet [ctor] .)
     (op '_U_ : 'VarSet 'VarSet -> 'VarSet
                       [ctor assoc comm id({'empty-var}'VarSet)] .)
     (op 'intersection : 'VarSet 'VarSet -> 'VarSet [none] .)
     (op '_in_ : 'Var 'VarSet -> 'Bool [none] .)
     (op 'new : 'VarSet -> 'Var [none] .)
     (op 'maxIndex : 'VarSet -> 'MachineInt [none] .)
     (op 'goal : 'VarSet 'SolvedPart 'ProducedPart 'JoinPart ->
                                                 'Goal [ctor] .)
     (op 'FAIL : nil -> 'Goal [ctor] .)
     (op 'eq : 'Var 'Term -> 'Equation [ctor] .)
     (op 'empty-eq : nil -> 'SolvedPart [ctor] .)
     (op '__ : 'SolvedPart 'SolvedPart -> 'SolvedPart
                    [ctor assoc comm id({'empty-eq}'SolvedPart)] .)
     (op 'variables : 'Expr -> 'VarSet [none] .)
     ...
```

Besides, we also have to include all the symbols of the corresponding signature. For that we will use an auxiliary function which, given a quoted identifier and a number n, generates a list with n occurrences of the identifier.

```
op repeatL : Qid MachineInt -> QidList .
eq repeatL(Q, N) = if N > 0 then Q repeatL(Q, _-_(N, 1))
                            else nil fi .
```

(The strange looking prefix notation `_-_(N, 1)` is used because the term `N-1` in infix notation becomes ambiguous at the metalevel and cannot be parsed correctly.) Then, `addC` and `addF` will add the corresponding declarations for constructor and function symbols, respectively. Recall that constructor symbols were overloaded.

```
op addC : ConstructorL OpDeclSet -> OpDeclSet .
op addF : FunctionL OpDeclSet -> OpDeclSet .

eq addC(nilC, ODS) = ODS .
eq addC(consC(cSymbol(Q,N),CL),ODS) =
   addC(CL,op Q : repeatL('Expr,N) -> 'Expr [ctor] .
           op Q : repeatL('Term,N) -> 'Term [ctor] . ODS) .
eq addF(nilF,ODS) = ODS .
eq addF(consF(fSymbol(Q,N),FL),ODS) =
   addF(FL,op Q : repeatL('Expr,N) -> 'Expr [ctor] . ODS) .
```

Similarly for variables. There are a few ones which are shared by all metarepresented modules (mainly those taking part in the equations associated to auxiliary operations) and can be assigned to a constant.

```
op varCRWL : -> VarDeclSet .

eq varCRWL = (var 'N : 'MachineInt .) (var 'V : 'Var .)
      (var 'V0 : 'Var .) (var 'UVars : 'VarSet .)
      (var 'VS : 'VarSet .) (var 'VS0 : 'VarSet .)
      (var 'EQ : 'Equation .) (var 'SP : 'SolvedPart .)
      (var 'PP : 'ProducedPart .)
      (var 'JP : 'JoinPart .)
      (var 'E : 'Expr .) (var 'E0 : 'Expr .)
      (var 'T : 'Term .) (var 'T0 : 'Term .)  .
```

But even for auxiliary operations we will sometimes need more variables, in a number which may vary from one module to another. For example, given a signature with a function symbol f of arity 4, one of the equationss associated to `variables` (which returns the variables appearing in an expression) would be

```
eq variables(f(E1, E2, E3, E4)) = variables(E1) U variables(E2) U
                                  variables(E3) U variables(E4) .
```

In general, we will need as many variables of sort `Expr` as the maximum arity of a symbol in the signature, and the same number of variables of sort `Term`. For that, two functions `maxC` and `maxF` calculating the maximum arity among constructor and function symbols, respectively, are defined. The operation `index`, appearing in the definition of `addVar`, is declared in the module `QID` and, given a quoted identifier and an integer, appends the number to the identifier.

```
op max : MachineInt MachineInt -> MachineInt .
op maxC : ConstructorL -> MachineInt .
op maxF : FunctionL -> MachineInt .
op addVar : MachineInt VarDeclSet -> VarDeclSet .

eq max(M,N) = if M > N then M else N fi .
eq maxC(nilC) = 0 .
eq maxC(consC(cSymbol(Q,N),CL)) = max(N,maxC(CL)) .
eq maxF(nilF) = 0 .
eq maxF(consF(fSymbol(Q,N),FL)) = max(N,maxF(FL)) .
eq addVar(N,VDS) = if N > 0
      then addVar(_-_(N,1), (var index('A,N) : 'Expr .)
                            (var index('B,N) : 'Expr .)
                            (var index('E,N) : 'Expr .)
                            (var index('T,N) : 'Term .) VDS)
      else VDS fi .
```

The additional variables `A`*i*, `B`*i*, will be only used in the metarepresentation of the rules **DC1** and **CF1**.

The following argument in the metarepresentation of the module is the one dealing with memberships; in our case its value is simply `none`.

Next we come to equations. As usual, we introduce a constant for the common ones.

```
op eqCRWL : -> EquationSet .

eq eqCRWL =
     (eq '_U_['V,'V] = 'V .)
     (eq '_in_['V,{'empty-var}'VarSet] = {'false}'Bool .)
     (eq '_in_['V,'_U_['V0,'VS]] = '_or_['_==_['V,'V0],
                                        '_in_['V,'VS]] .)
     (eq 'maxIndex[{'empty-var}'VarSet] = {'0}'MachineInt .)
     (eq 'maxIndex['_U_['v['N],'VS]] =
         'if_then_else_fi['_>_['N,'maxIndex['VS]],'N,
                                          'maxIndex['VS]] .)
     (eq 'new['VS] = 'v['_+_['maxIndex['VS],{'1}'MachineInt]] .)
     (eq 'intersection['VS,{'empty-var}'VarSet] =
                                   {'empty-var}'VarSet .)
     (eq 'intersection['VS,'_U_['V,'VS0]] =
         'if_then_else_fi['_in_['V,'VS],
```

```
                        '_U_['V,'intersection['VS,'VS0]],
                        'intersection['VS,'VS0]] .)
   (eq '__['EQ,'EQ] = 'EQ .)
   (eq 'variables['V] = 'V .)
   (eq 'variables[{'empty-approx}'ProducedPart] =
                                   {'empty-var}'VarSet .)
   (eq 'variables['__['approx['E,'E0],'PP]] =
          '_U_['variables['E],'variables['E0],'variables['PP]] .)
   ...
```

And now we have to add those equations associated to `variables`, `safeVar` and `subst` that depend on the signature. Three auxiliary functions will be used here, all of them generating the metarepresentation of the list of arguments of an operation.

```
op genArgs : Qid MachineInt -> TermList .
op auxEq1 : Qid MachineInt -> TermList .
op auxEq2 : MachineInt -> TermList .

eq genArgs(Q,N) = if N > 1 then genArgs(Q,_-_(N,1)), index(Q,N)
                           else index(Q,1) fi .
eq auxEq1(Q,N) = if N > 0 then auxEq1(Q,_-_(N,1)), Q[index('E,N)]
                          else {'empty-var}'VarSet fi .
eq auxEq2(N) = if N > 1
               then auxEq2(_-_(N, 1)), 'subst[index('E,N),'V,'T]
               else 'subst['E1,'V,'T] fi .
```

The easiest way to understand how they work is by means of an example. Let us recall, then, the equation `variables(f(E1, E2, E3, E4)) = variables(E1) U variables(E2) U variables(E3) U variables(E4)` displayed some lines above. Its metarepresentation could be obtained, using these functions, in the following way:

```
eq 'variables['f[genArgs('E,4)]] = '_U_[auxEq1('variables,4)] .
```

The working of `auxEq2` is similar. Notice that a single occurrence of the operation `_U_` is needed in the metarepresentation because `assoc` is one of its equational attributes. With their help it is easy to define the functions in charge of completing the metarepresentation of the set of equations.

```
op addEqC : ConstructorL EquationSet -> EquationSet .
op addEqF : FunctionL EquationSet -> EquationSet .
```

```
eq addEqC(nilC,ES) = ES .
eq addEqC(consC(cSymbol(Q,N),CL),ES) = if N == 0
     then addEqC(CL,
                 (eq 'variables[{Q}'Term] = {'empty-var}'VarSet .)
                 (eq 'safeVar[{Q}'Term] = {'empty-var}'VarSet .)
                 (eq 'subst[{Q}'Term,'V,'T] = {Q}'Term .) ES)
     else addEqC(CL,
                 (eq 'variables[Q[genArgs('E,N)]] =
                     '_U_[auxEq1('variables,N)] .)
                 (eq 'safeVar[Q[genArgs('E,N)]] =
                     '_U_[auxEq1('safeVar,N)] .)
                 (eq 'subst[Q[genArgs('E,N)],'V,'T] =
                      Q[auxEq2(N)] .) ES)
     fi .
```

The reason why we distinguish two cases is that the metarepresentation of constants in Maude's metalevel is not a particular case of the general metarepresentation of terms whose top symbol has arity greater than zero. The corresponding case for `addEqF` is entirely equal except for one of the equations associated to `safeVar`.

Finally, we are only left with the metarepresentation of the rules in Maude corresponding to the CLNC-rules. Rules **ID**, **BD**, **OB1**, **OB2**, **IB**, **EL** and **CY** pose no problem, as they are theory independent and so are their metarepresentations.

```
op ruleCRWL : -> RuleSet .
eq ruleCRWL =
  (crl['ID] : 'goal['UVars,'SP,'PP,'__['join['V,'V],'JP]] =>
              'goal['UVars,'SP,'PP,'JP]
              if 'not_['_in_['V,'producedVar['PP]]] = {'true}'Bool .)
  (crl['BD] : 'goal['UVars,'SP,'PP,'__['join['V,'T],'JP]] =>
              'goal['UVars,'__['eq['V,'T],'subst['SP,'V,'T]],
                           'subst['PP,'V,'T],'subst['JP,'V,'T]]
              if '_and_['_==_['intersection['variables['T],
                                             'producedVar['PP]],
                              {'empty-var}'VarSet],
                         'not_['_in_['V,'_U_['variables['T],
                                             'producedVar['PP]]]]] =
                 {'true}'Bool .)
  (crl['OB1] : 'goal['UVars,'SP,'__['approx['V,'T],'PP],'JP] =>
               'goal['UVars,'__['eq['V,'T],'subst['SP,'V,'T]],
                            'subst['PP,'V,'T],'subst['JP,'V,'T]]
               if '_and_['not_['T : 'Var],
                          'not_['_in_['V,'producedVar['PP]]]] =
                  {'true}'Bool .)
  (crl['OB2] : 'goal['UVars,'SP,'__['approx['V,'T],'PP],'JP] =>
```

```
               'goal['UVars,'SP,'subst['PP,'V,'T],'subst['JP,'V,'T]]
               if '_and_['not_['T : 'Var],
                         '_in_['V,'producedVar['PP]]] = {'true}'Bool .)
(rl['IB] : 'goal['UVars,'SP,'__['approx['T,'V],'PP],'JP] =>
           'goal['UVars,'SP,'subst['PP,'V,'T],'subst['JP,'V,'T]] .)
(crl['EL] : 'goal['UVars,'SP,'__['approx['E,'V],'PP],'JP] =>
            'goal['UVars,'SP,'PP,'JP]
            if 'not_['_in_['V,'_U_['variables['PP],
                                   'variables['JP]]]] =
               {'true}'Bool .)
(crl['CY] : 'goal['UVars,'SP,'PP,'__['join['V,'E],'JP]] =>
            {'FAIL}'Goal
            if '_and_['_=/=_['V,'E],'_in_['V,'safeVar['E]]] =
               {'true}'Bool .) .
```

For the rest of the rules things become a bit harder. Let us first examine how the metarepresentation of the rules corresponding to **DC1**, an easy one, is obtained. We will have to generate one metarepresentation for each constructor symbol. The only point worth mentioning here is the auxiliary *zip*-like function `_joinTerm_` which, given two lists of metaterms, returns a list of terms metarepresenting the `join` of corresponding terms in the lists.

```
op _joinTerms_ : TermList TermList -> TermList .
op ruleDC1 : ConstructorL -> RuleSet .

eq T1 joinTerms T2 = 'join[T1,T2] .
eq ( T1,TL1 ) joinTerms ( T2,TL2 ) =
                            'join[T1,T2], (TL1 joinTerms TL2) .
eq ruleDC1(nilC) = none .
eq ruleDC1(consC(cSymbol(Q,N),CL)) = if N > 0
  then rl['DC1] : 'goal['UVars,'SP,'PP,
                        '__['join[Q[genArgs('A,N)],
                                  Q[genArgs('B,N)]],'JP]] =>
                  'goal['UVars,'SP,'PP,
                        '__[genArgs('A,N) joinTerms genArgs('B,N),
                        'JP]] .
       ruleDC1(CL)
  else rl['DC1] : 'goal['UVars,'SP,'PP,'__['join[{Q}'Term,
                                                 {Q}'Term],'JP]] =>
                  'goal['UVars,'SP,'PP,'JP] .
       ruleDC1(CL)
  fi .
```

Again, the reason to distinguish two cases in the definition of `ruleDC1` is imposed (mainly, though not uniquely) by the way terms are metarepresented in Maude.

Some of the remaining rules make use of fresh variables and, to create them, we introduce the function `genNewVars` which, given the metarepresentation of a set of variables and a positive integer n, returns a list of n terms metarepresenting variables different from those in the original set.

```
op genNewVars : Term MachineInt -> TermList .
eq genNewVars(Q,N) =
      if N > 1 then 'new[Q],genNewVars('_U_[Q,'new[Q]],_-_(N,1))
               else 'new[Q] fi .
```

`genNewVars` is used, for example, in the construction of the metarepresentation of the rules corresponding to **IIM**. The operation `_approxTerms_` also appearing in it is the equivalent to `_joinTerms_` for `approx`.

```
op ruleIIM : ConstructorL -> RuleSet .
eq ruleIIM(nilC) = none .
eq ruleIIM(consC(cSymbol(Q,N),CL)) = if N > 0 then
    crl['IIM]: 'goal['UVars,'SP,
                     '__['approx[Q[genArgs('E,N)],'V],'PP],'JP] =>
               'goal['_U_[genNewVars('UVars,N),'UVars],'SP,
                     'subst['__[genArgs('E,N) approxTerms
                                         genNewVars('UVars,N),'PP],
                            'V,Q[genNewVars('UVars,N)]],
                     'subst['JP,'V,Q[genNewVars('UVars,N)]]]
               if '_and_['not_[Q[genArgs('E,N)] : 'Term],
                         '_in_['V,'demandVar['JP]]] = {'true}'Bool .
    ruleIIM(CL)
  else ruleIIM(CL) fi .
```

Rule **CF2** (and **CF1**) poses a different difficulty. Now we have to metarepresent a rule for each *pair* of constructor symbols, so we will need to declare sorts to build pairs and operations to cope with them.

```
sorts PairC PairCL .
subsort PairC < PairCL .
op <_,_> : Constructor Constructor -> PairC [ctor] .
op nilP : -> PairCL [ctor] .
op consP : PairCL PairCL -> PairCL [ctor assoc id: nilP] .
op pairsL : ConstructorL -> PairCL .
```

The operation `pairsL` returns a list of all possible pairs of constructors in the signature and, with this information, `ruleCF2Aux` undertakes the rest of the work.

```
op createTerm : Qid Qid MachineInt -> Term .
eq createTerm(Q, Q0, N) = if N > 0 then Q[genArgs(Q0,N)]
                                   else {Q}'Term fi .

op ruleCF2 : ConstructorL -> RuleSet .
op ruleCF2Aux : PairCL -> RuleSet .
eq ruleCF2(CL) = ruleCF2Aux(pairsL(CL)) .
eq ruleCF2Aux(nilP) = none .
eq ruleCF2Aux(consP(< cSymbol(Q,N), cSymbol(Q0,N0) >,PCL)) =
  (rl['CF2]: 'goal['UVars,'SP,
                  '__['approx[createTerm(Q,'E,N),
                              createTerm(Q0,'T,N0)],'PP],
                  'JP]
        => {'FAIL}'Goal .)
  (rl['CF2]: 'goal['UVars,'SP,
                  '__['approx[createTerm(Q0,'E,N0),
                              createTerm(Q,'T,N)],'PP],
                  'JP]
        => {'FAIL}'Goal .)
   ruleCF2Aux(PCL) .
```

The only purpose of the operation `createTerm` is to save typing by avoiding to have to distinguish the case $n = 0$. It was not used in previous occassions because additional modifications would have been required.

The only rules not totally covered by the techniques described above are **NR1** and **NR2**. These are, by far, the more difficult rules to metarepresent. The reason is that when we use an instance of a program rule we must employ fresh variables and we are entrusted with, first, creating them, and then performing the necessary substitutions so as to rename the original rule. However, this is an issue bearing no special interest and which can be solved with a slight extension of the methods we have already employed. As the details of this part are extensive and contribute with nothing really new, we do not include them here; they can be found in Appendix B.

19.2.3 Building a depth-first strategy

The functional module we have just finished explaining simply supplies us with a metarepresentation of a module simulating CLNC for some CRWL theory, and naively applying to it the descent functions to solve goals will not work, as the nondeterminism is still there. What remains to be done is, taking advantage of the reflective capabilities of Maude, to build a module defining a strategy in order to control the rewriting of a term and the search in the tree of possible rewritings of a term. Our inspiration in this regard comes from [25], from which we borrow some of the operations explained in this section.

The module implementing the search strategy is parameterized with respect to two constants, equal to the metarepresentation of the Maude module which we want to work with and the list of rules in the module we are allowed to apply, respectively.

```
(fth AMODULE is
  including META-LEVEL .
  op MOD : -> Module .
  op labels : -> QidList .
endfth)
```

Parameterized modules, *theories*, and *views* are not directly supported by Maude, but belong to an extension called Full Maude. Parameterized datatypes use theories to specify the requirements the parameters must satisfy, while views perform the instantiations. Unlike that of modules, the semantics of theories is *loose*. The new syntax introduced is minimal and, hopefully, self-explanatory. Note however that the input to Full Maude must be given enclosed in parentheses. More details can be found in [9, 8, 11].

The module containing the strategy will be the parameterized module `SEARCH[M :: MODULE]`. The strategy controls the possible rewritings of a term by means of the descent function `meta-apply`. `meta-apply` returns *one* of the possible rewritings at the top level of a given term. Our first step will be to define an operation `allRew` that returns all the possible *one-step* rewritings of a given term by using rewrite rules with labels in the list `labels`. We first need to extend the sort `TermList` with an identity element.

```
(mod SEARCH[M :: AMODULE] is
  including META-LEVEL .

  op ~ : -> TermList [ctor] .
  var TL : TermList .
  eq ~, TL = TL .
  eq TL, ~ = TL .
```

The operations needed to find all the possible rewritings, and their definitions, are as follows.

```
  var T : Term .
  var SB : Substitution .
  op extTerm : ResultPair -> Term .
  eq extTerm({T, SB}) = T .
```

```
  op allRew : Term QidList -> TermList .
  op topRew : Term Qid MachineInt -> TermList .
  op lowerRew : Term Qid -> TermList .
  op rewArguments : Qid TermList TermList Qid -> TermList .
  op rebuild : Qid TermList TermList TermList -> TermList .

  var LS : QidList .
  vars L C S OP : Qid .
  vars Before After : TermList .
  var N : MachineInt .

  eq allRew(T, nil) = ~ .
  eq allRew(T, L LS) =
         topRew(T, L, 0), *** rewriting at the top of T with label L
         lowerRew(T, L),  *** rewriting of (proper) subterms with label L
         allRew(T, LS) .  *** rewriting with labels LS

  eq topRew(T, L, N) =
      if extTerm(meta-apply(MOD, T, L, none, N)) == error* then ~
      else extTerm(meta-apply(MOD, T, L, none, N)),
           topRew(T, L, N + 1) fi .

  eq lowerRew({C}S, L) = ~ .
  eq lowerRew(OP[TL], L) = rewArguments(OP, ~, TL, L) .

  eq rewArguments(OP, Before, T, L) =
      rebuild(OP, Before, allRew(T, L), ~) .
  eq rewArguments(OP, Before, (T, After), L) =
      rebuild(OP, Before, allRew(T, L), After) ,
      rewArguments(OP, (Before, T), After, L) .

  eq rebuild(OP, Before, ~, After) = ~ .
  eq rebuild(OP, Before, T, After) =
      meta-reduce(MOD, OP[Before, T, After]) .
  eq rebuild(OP, Before, (T, TL), After) =
      meta-reduce(MOD, OP[Before, T, After]),
      rebuild(OP, Before, TL, After) .
```

In order for our strategy to be as general as possible, we assume that our metarepresented module `MOD` has an operation `ok` which returns a value of sort `Answer` such that `ok(T) = solution`, `maybe-sol` or `no-solution` depending on whether the term `T` is one of the terms we are looking for, it is not but may rewrite to one of those, or it is not and no solution can be found below it in the tree. In our case, this means that we must modify our module `MAP-PHI` so that the constants `sortCRWL`, `opCRWL`, and `eqCRWL` also include the metarepresentation of the following:

```
  sort Answer .
  ops solution maybe-sol no-solution : -> Answer [ctor] .
  op ok : Goal -> Answer .
  eq ok(FAIL) = no-solution .
  eq ok(goal(UVars, SP, PP, JP)) =
        if (PP == empty-approx) and (JP == empty-join)
           then solution else maybe-sol fi .
```

This definition of `ok` is suitable because it is proved in [14] that if a goal is not in solved form and is different from *FAIL* then there exists some CLNC transformation applicable to it.

Now we can define a strategy that searches through the tree of possible rewritings in a depth-first manner.

```
  op rewDepth : TermList -> Term .

  eq rewDepth(~) = error* .
  eq rewDepth(T) =
      if meta-reduce(MOD, 'ok[T]) == {'solution}'Answer then T
      else (if meta-reduce(MOD, 'ok[T]) == {'no-solution}'Answer
            then T
            else rewDepth(allRew(T, labels))
            fi)
      fi .
  eq rewDepth( (T,TL) ) =
      if meta-reduce(MOD, 'ok[T]) == {'solution}'Answer then T
      else (if meta-reduce(MOD, 'ok[T]) == {'no-solution}'Answer
            then rewDepth(TL)
            else rewDepth( (allRew(T, labels), TL) )
            fi)
      fi .
endm)
```

Note that the first equation associated to `rewDepth` is never used; in the second, if `ok` applied to the term gives `no-solution`, then the term must be `FAIL` and is returned as answer, as all the tree has already been explored. (Applying this strategy to another module might require a slight modification of these points.)

Let us now look at how to use this strategy to solve goals, using again our example of the natural numbers. We first need to get the metarepresentation of the module associated to it.

```
(mod MAP-PHI-NAT is
```

```
  including MAP-PHI .

  op consNat : -> ConstructorL .
  op funcNat : -> FunctionL .
  op rulNat : -> RuleCRWLL .

  eq consNat = consC(cSymbol('zero, 0), consC(cSymbol('s, 1),nilC)) .
  eq funcNat = consF(fSymbol('+, 2), consF(fSymbol('*, 2), nilF)) .
  eq rulNat = consR( '+['v[{'0}'MachineInt],{'zero}'Term] ->
                      'v[{'0}'MachineInt] <= nilPT,
               consR( '+['v[{'1}'MachineInt],'s['v[{'3}'MachineInt]]] ->
                       's['+['v[{'1}'MachineInt],'v[{'3}'MachineInt]]]
                       <= nilPT ,
               consR( '*['v[{'0}'MachineInt],{'zero}'Term] ->
                       {'zero}'Term <= nilPT,
               consR( '*['v[{'0}'MachineInt],'s['v[{'1}'MachineInt]]] ->
                       '+['*['v[{'0}'MachineInt],'v[{'1}'MachineInt]],
                          'v[{'0}'MachineInt]] <= nilPT,
               nilR)))) .

  op METANAT : -> Module .
  op CLNClabels : -> QidList .

  eq METANAT = phi(crwlth(consNat, funcNat, rulNat)) .
  eq CLNClabels = 'DC1 'ID 'BD 'IM 'NR1 'DC2 'OB1 'OB2 'IB 'IIM 'EL
                  'NR2 'CF1 'CY 'CF2 .
endm)
```

We then declare a view and instantiate the module `SEARCH` with it.

```
(view ModuleEx from AMODULE to MAP-PHI-NAT is
  op MOD to METANAT .
  op labels to CLNClabels .
endv)

(mod SEARCH-NAT is
  protecting SEARCH[ModuleEx]
endm)
```

And now, the problematic goal

```
  goal(v(0), empty-eq, empty-approx, join(*(v(0),s(zero)),zero))
```

does not longer lead to `FAIL`:

```
Maude> (red rewDepth('goal['v[{'0}'MachineInt],
                           {'empty-eq}'SolvedPart,
                           {'empty-approx}'ProducedPart,
                           'join['*['v[{'0}'MachineInt],'s[{'zero}'Term]],
                                 {'zero}'Term]]) .)

Result Term : 'goal['_U_['v[{'0}'MachineInt], 'v[{'1}'MachineInt],
                         'v[{'2}'MachineInt], 'v[{'3}'MachineInt],
                         'v[{'4}'MachineInt]],
                     'eq['v[{'0}'MachineInt], {'zero}'Term],
                     {'empty-approx}'ProducedPart,
                     {'empty-join}'JoinPart]
```

At this point some remarks should be made. As it was pointed out before and this last example makes clear again, the interface of this module is somewhat cumbersome and, in order to work realistically with it, a parser should be developed. In this regard, it should be mentioned that there exist in Full Maude two functions called `up` and `down` that automatically shift from the object-level to the meta-level representation, and vice versa, and that could be used to avoid some of the "cumbersomity". In addition, the resolution of goals is time-consuming. Among the reasons for this slowness are the fact that we are working at the metalevel, the multitude of equational attributes (`comm`, `assoc`, `id`) in the declarations of the operations (so that rewriting has to be performed modulo them), and that the search strategy we have chosen (depth-first search) is not efficient. (In $\mathcal{TOY}$, an experimental language and system that implements the CRWL paradigm, a different set of rules from those of CLNC is used, together with a demand driven computational strategy to apply them.) On the other hand, the approach we have followed has been easy to develop, and can be extended without difficulty.

§20. Conclusions

The main outcome of the research carried out in this paper has been the clarification of the relationship between RL and CRWL. Both logics have been proved to be expressive enough to simulate deduction in each other in a simple way. On the other hand, neither can RL be considered as a sublogic of CRWL, nor can CRWL with respect to RL, and this certainly was not something we had expected.

During the preparation of this work we have been forced to take a close look at the notions of entailment system and institution, and the difficulties we have found have shown us that intuition can be misleading in this field. The conclusion we have reached is that it would be very convenient to develop some kind of generalization of these concepts. One reason supporting this claim is the fact that, although it seems clear that CRWL should fit within the frame of entailment systems, the lack of the transitivity property forbids it to be considered so. In addition, there have been several occasions wherein we have had to make a distinction between two types of sentences within the same logic. The most outstanding case was that of labelled and unlabelled rewrite rules in RL, but we should also emphasize that rules in CRWL-programs are a restricted class of the more general class of reduction statement, and that, when we talked about reflection, we had to "weaken" its definition in order to encompass some of the results about RL, due to conditional sentences not being treated like unconditional ones. What all these examples have in common is that sentences belonging to a theory are given a different treatment from the rest of sentences and, with the current definitions of entailment system and institution, there is no way of taking this distinction into account.

For future work, besides the generalizations just mentioned, it would be interesting to complete the definition of the institution $\mathcal{I}_{\mathrm{RL}}$ with signature morphisms (morphisms of RL-theories), as well as finding a more illustrative example showing the reasons why RL is not embeddable in CRWL.

References

[1] Puri Arenas-Sánchez and Mario Rodríguez-Artalejo. A general framework for lazy functional logic programming with algebraic polymorphic types. *Theory and Practice of Logic Programming*, 1(2):185–245, March 2001.

[2] Purificación Arenas-Sánchez. *Programación Declarativa con Restricciones sobre Tipos de Datos Algebraicos.* PhD thesis, Universidad Complutense de Madrid, Spain, December 1998.

[3] Purificación Arenas-Sánchez, Francisco J. López-Fraguas, and Mario Rodríguez-Artalejo. Embedding multiset constraints into a lazy functional logic language. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *Principles of Declarative Programming, 10th International Symposium, PLIP'98 Held Jointly with the 6th Conference ALP'98, Pisa, Italy, September 16–18, 1998, Proceedings*, volume 1490 of *Lecture Notes in Computer Science*, pages 429–444. Springer-Verlag, 1998.

[4] Michael Barr and Charles Wells. *Category Theory for Computing Science.* Centre de Recherches Mathématiques, third edition, 1999.

[5] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[6] Manuel Clavel. *Reflection in General Logics and in Rewriting Logic, with Applications to the Maude Language.* PhD thesis, Universidad de Navarra, Spain, February 1998.

[7] Manuel Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications.* CSLI Lecture Notes. CSLI Publications, 2000.

[8] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. Manual distributed as documentation of the Maude system, Computer Science Laboratory, SRI International. `http://maude.csl.sri.com/manual`, January 1999.

[9] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. A Maude tutorial. Tutorial distributed as documentation of the Maude system, Computer Science Laboratory, SRI International. Presented at the *European Joint Conference on Theory and Practice of Software, ETAPS 2000*, Berlin, Germany, March 25, 2000. `http://maude.csl.sri.com/tutorial`, March 2000.

[10] Manuel Clavel and José Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 2001. To appear.

[11] Francisco Durán. *A Reflective Module Algebra with Applications to the Maude Language.* PhD thesis, Universidad de Málaga, Spain, June 1999. `http://maude.csl.sri.com/papers`.

[12] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.

[13] Joseph A. Goguen and José Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985. Preliminary version appeared in: *SIGPLAN Notices*, July 1981, Volume 16, Number 7, pages 24-37.

[14] Juan Carlos González-Moreno, M. Teresa Hortalá-González, Francisco J. López-Fraguas, and Mario Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.

[15] Juan Carlos González-Moreno, M. Teresa Hortalá-González, and Mario Rodríguez-Artalejo. Polymorphic types in functional logic programming. *Journal of Functional and Logic Programming*, 2001. To appear.

[16] J. Lambek. Subequalizers. *Canadian Mathematical Bulletin*, 13:337–349, 1970.

[17] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, second edition, 1998.

[18] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993. To appear in D. Gabbay, editor, *Handbook of Philosophical Logic, Second Edition, Volume 6*, Kluwer Academic Publishers, 2001. `http://maude.csl.sri.com/papers`.

[19] José Meseguer. General logics. In H.-D. Ebbinghaus, J. Fernández-Prida, M. Garrido, D. Lascar, and M. Rodríguez-Artalejo, editors, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.

[20] José Meseguer. Rewriting as a unified model of concurrency. Technical Report SRI-CSL-90-02, SRI International, Computer Science Laboratory, February 1990. Revised June 1990.

[21] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[22] José Meseguer. Membership algebra as a logical framework for equational specification. In Francesco Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1998.

[23] Hiroyuki Miyoshi. Modelling conditional rewriting logic in structured categories. In José Meseguer, editor, *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 20–34. Elsevier, 1996. `http://www.elsevier.nl/locate/entcs/volume4.html`.

[24] Juan Miguel Molina-Bravo. *Modularidad en Programación Lógico-Funcional de Primer Orden.* PhD thesis, Universidad de Málaga, December 2000.

[25] Alberto Verdejo and Narciso Martí-Oliet. Executing and verifying CCS in Maude. Technical Report 99-00, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, February 2000. `http://maude.csl.sri.com/casestudies/ccs`.

§A. The Module NAT-CRWL

```
mod NAT-CRWL is
  protecting MACHINE-INT .

  sorts Var VarSet .
  subsort Var < VarSet .

  op v : MachineInt -> Var [ctor] .
  op empty-var : -> VarSet [ctor].
  op _U_ : VarSet VarSet -> VarSet [ctor assoc comm id: empty-var] .
  op intersection : VarSet VarSet -> VarSet .
  op _in_ : Var VarSet -> Bool .
  op new : VarSet -> Var .
  op maxIndex : VarSet -> MachineInt .

  var N : MachineInt .
  vars V V' : Var .
  vars VS VS' : VarSet .

  eq V U V = V .
  eq V in empty-var = false .
  eq V in (V' U VS) = (V == V') or (V in VS) .

  eq maxIndex(empty-var) = 0 .
  eq maxIndex(v(N) U VS) =
       if (N > maxIndex(VS)) then N else maxIndex(VS) fi .
  eq new(VS) = v(maxIndex(VS) + 1) .

  eq intersection(VS,empty-var) = empty-var .
  eq intersection(VS,V U VS') =
       if (V in VS) then (V U intersection(VS,VS'))
                    else intersection(VS,VS') fi .

  sorts Term Expr Equation SolvedPart
        ApproxStatement ProducedPart JoinStatement JoinPart
        Goal .
  subsort Var < Term < Expr .

  op goal : VarSet SolvedPart ProducedPart JoinPart -> Goal [ctor] .
  op FAIL : -> Goal [ctor] .

  subsort Equation < SolvedPart .
  op eq : Var Term -> Equation [ctor] .
  op empty-eq : -> SolvedPart [ctor] .
  op __ : SolvedPart SolvedPart -> SolvedPart
                                        [ctor assoc comm id: empty-eq] .
```

```
var EQ : Equation .
eq EQ EQ = EQ . *** the solved part is a set

subsort ApproxStatement < ProducedPart .
op approx : Expr Term  -> ApproxStatement [ctor] .
op empty-approx : -> ProducedPart [ctor] .
op __ : ProducedPart ProducedPart -> ProducedPart
                                  [ctor assoc comm id: empty-approx] .

subsort JoinStatement < JoinPart .
op join : Expr Expr -> JoinStatement [ctor comm] .
op empty-join : -> JoinPart [ctor] .
op __ : JoinPart JoinPart -> JoinPart [ctor assoc comm id: empty-join] .

subsort JoinStatement < JoinPart .
op join : Expr Expr -> JoinStatement [comm] .
op empty-join : -> JoinPart .
op __ : JoinPart JoinPart -> JoinPart [assoc comm id: empty-join] .

op zero : -> Term [ctor] .
op s : Expr -> Expr [ctor] .
op s : Term -> Term [ctor] .
ops + * : Expr Expr -> Expr [ctor] .

vars E E' : Expr .
vars E1 E2 A1 A2 B1 B2 : Expr .
var T T' : Term .
var SP : SolvedPart .
var AS : ApproxStatement .
var PP : ProducedPart .
var JS : JoinStatement .
var JP : JoinPart .
var UVars : VarSet .

op variables : Expr -> VarSet .
op variables : ProducedPart -> VarSet .
op variables : JoinPart -> VarSet .
eq variables(V) = V .
eq variables(zero) = empty-var .
eq variables(s(E)) = variables(E) .
eq variables(+(E,E')) = variables(E) U variables(E') .
eq variables(*(E,E')) = variables(E) U variables(E') .
eq variables(empty-approx) = empty-var .
eq variables(approx(E,E') PP) = variables(E) U variables(E') U
                                variables(PP) .
eq variables(empty-join) = empty-var .
eq variables(join(E,E') JP) = variables(E) U variables(E') U
```

```
                              variables(JP) .

op safeVar : Expr -> VarSet .
eq safeVar(V) = V .
eq safeVar(zero) = empty-var .
eq safeVar(s(E)) = safeVar(E) .
eq safeVar(+(E,E')) = empty-var .
eq safeVar(*(E,E')) = empty-var .

op producedVar : ProducedPart -> VarSet .
eq producedVar(empty-approx) = empty-var .
eq producedVar(approx(E,E') PP) = variables(E') U producedVar(PP) .

op demandVar : JoinPart -> VarSet .
eq demandVar(empty-join) = empty-var .
eq demandVar(join(V,V') JP) = V U V' U demandVar(JP) .
ceq demandVar(join(V,E') JP) = V U demandVar(JP) if not (E' : Var) .
ceq demandVar(join(E,E') JP) = demandVar(JP)
    if not (E : Var) and not (E' : Var) .

op subst : Expr Var Term -> Expr .
op subst : SolvedPart Var Term -> SolvedPart .
op subst : ProducedPart Var Term -> ProducedPart .
op subst : JoinPart Var Term -> JoinPart .
ceq subst(V',V,T) = T if V' == V .
ceq subst(V',V,T) = V' if V' =/= V .
eq subst(zero,V,T) = zero .
eq subst(s(E),V,T) = s(subst(E,V,T)) .
eq subst(+(E,E'),V,T) = +(subst(E,V,T),subst(E',V,T)) .
eq subst(*(E,E'),V,T) = *(subst(E,V,T),subst(E',V,T)) .
eq subst(empty-eq,V,T) = empty-eq .
eq subst(eq(V',T') SP,V,T) = eq(V',subst(T',V,T)) subst(SP,V,T) .
eq subst(empty-approx,V,T) = empty-approx .
eq subst(approx(E,E') PP,V,T) =
     approx(subst(E,V,T),subst(E',V,T)) subst(PP,V,T) .
eq subst(empty-join,V,T) = empty-join .
eq subst(join(E,E') JP,V,T) =
     join(subst(E,V,T),subst(E',V,T)) subst(JP,V,T) .


*** rules for joinability
rl [DC1] : goal(UVars,SP,PP,join(zero,zero) JP) =>
           goal(UVars,SP,PP,JP) .
rl [DC1] : goal(UVars,SP,PP,join(s(A1),s(B1)) JP) =>
           goal(UVars,SP,PP,join(A1,B1) JP) .

crl [ID] : goal(UVars,SP,PP,join(V,V) JP) => goal(UVars,SP,PP,JP)
```

```
             if not (V in producedVar(PP)) .

  crl [BD] : goal(UVars,SP,PP,join(V,T) JP) =>
             goal(UVars,eq(V,T) subst(SP,V,T),subst(PP,V,T),subst(JP,V,T))
             if intersection(variables(T),producedVar(PP)) == empty-var
                and not V in (variables(T) U producedVar(PP)) .

  crl [IM] : goal(UVars,SP,PP,join(V,s(E)) JP) =>
             goal(new(UVars) U UVars,
                   eq(V,s(new(UVars))) subst(SP,V,s(new(UVars))),
                   subst(PP,V,s(new(UVars))),
                   subst(join(new(UVars),E) JP,V,s(new(UVars))))
             if (not (s(E) : Term) or
                 intersection(variables(s(E)),producedVar(PP)) =/=
                 empty-var)
                and not (V in producedVar(PP))
                and not (V in safeVar(s(E))) .

  rl [NR1] : goal(UVars,SP,PP,join(+(E1,E2),E) JP) =>
             goal(new(UVars) U UVars,SP,
                  approx(E1,new(UVars)) approx(E2,zero) PP,
                  join(new(UVars),E) JP) .
  rl [NR1] : goal(UVars,SP,PP,join(+(E1,E2),E) JP) =>
             goal(new(new(UVars) U UVars) U UVars,SP,
                  approx(E1,new(UVars))
                      approx(E2,s(new(new(UVars) U UVars))) PP,
                  join(s(+(new(UVars),new(new(UVars) U UVars))),E) JP) .
  rl [NR1] : goal(UVars,SP,PP,join(*(E1,E2),E) JP) =>
             goal(new(UVars) U UVars,SP,
                  approx(E1,new(UVars)) approx(E2,zero) PP,
                  join(zero,E) JP) .
  rl [NR1] : goal(UVars,SP,PP,join(*(E1,E2),E) JP) =>
             goal(new(new(UVars) U UVars) U UVars,SP,
                  approx(E1,new(UVars))
                       approx(E2,s(new(new(UVars) U UVars))) PP,
                  join(+(*(new(UVars),new(new(UVars) U UVars)),
                         new(UVars)),E) JP) .

  *** rules for reduction

  rl [DC2] : goal(UVars,SP,approx(zero,zero) PP,JP) =>
             goal(UVars,SP,PP,JP) .
  rl [DC2] : goal(UVars,SP,approx(s(E),s(T)) PP,JP) =>
             goal(UVars,SP,approx(E,T) PP,JP) .

  crl [OB1] : goal(UVars,SP,approx(V,T) PP,JP) =>
              goal(UVars,eq(V,T) subst(SP,V,T),subst(PP,V,T),
```

```
                 subst(JP,V,T))
            if (not T : Var) and not (V in producedVar(PP)) .
crl [OB2] : goal(UVars,SP,approx(V,T) PP,JP) =>
            goal(UVars,SP,subst(PP,V,T),subst(JP,V,T))
            if (not T : Var) and (V in producedVar(PP)) .

rl [IB] : goal(UVars,SP,approx(T,V) PP,JP) =>
          goal(UVars,SP,subst(PP,V,T),subst(JP,V,T)) .

crl [IIM] : goal(UVars,SP,approx(s(E),V) PP,JP) =>
            goal(new(UVars) U UVars,SP,
                 subst(approx(E,new(UVars)) PP,V,s(new(UVars))),
                 subst(JP,V,s(new(UVars))))
            if not (s(E) : Term) and (V in demandVar(JP)) .

crl [EL] : goal(UVars,SP,approx(E,V) PP,JP) =>
           goal(UVars,SP,PP,JP)
           if not (V in (variables(PP) U variables(JP))) .

crl [NR2] : goal(UVars,SP,approx(+(E1,E2),T) PP,JP) =>
            goal(new(UVars) U UVars,SP,
                 approx(E1,new(UVars)) approx(E2,zero)
                                       approx(new(UVars),T) PP, JP)
            if (not T : Var) or (T in demandVar(JP)) .
crl [NR2] : goal(UVars,SP,approx(+(E1,E2),T) PP,JP) =>
            goal(new(new(UVars) U UVars) U UVars,SP,
                 approx(E1,new(UVars))
                 approx(E2,s(new(new(UVars) U UVars)))
                 approx(s(+(new(UVars),new(new(UVars) U UVars))),T) PP,
                 JP)
            if (not T : Var) or (T in demandVar(JP)) .
crl [NR2] : goal(UVars,SP,approx(*(E1,E2),T) PP,JP) =>
            goal(new(UVars) U UVars,SP,
                approx(E1,new(UVars)) approx(E2,zero) approx(zero,T) PP,
                JP)
            if (not T : Var) or (T in demandVar(JP)) .
crl [NR2] : goal(UVars,SP,approx(*(E1,E2),T) PP,JP) =>
            goal(new(new(UVars) U UVars) U UVars,SP,
                 approx(E1,new(UVars))
                 approx(E2,s(new(new(UVars) U UVars)))
                 approx(+(*(new(UVars),new(new(UVars) U UVars)),
                          new(UVars)),T) PP,
                 JP)
             if (not T : Var) or (T in demandVar(JP)) .

*** failure rules
```

```
  rl [CF1] : goal(UVars,SP,PP,join(zero,s(E)) JP) => FAIL .

  crl [CY] : goal(UVars,SP,PP,join(V,E) JP) => FAIL
             if (V =/= E) and (V in safeVar(E)) .

  rl [CF2] : goal(UVars,SP,approx(zero,s(E)) PP,JP) => FAIL .
  rl [CF2] : goal(UVars,SP,approx(s(E),zero) PP,JP) => FAIL .

endm
```

§B. The Module MAP-PHI

This code already includes the metarepresentation of the sort `Answer` and its associated operations mentioned on page 96.

```
(fmod MAP-PHI is
  protecting META-LEVEL .
  protecting MACHINE-INT .
  protecting BOOL .

  sorts Constructor ConstructorL Function FunctionL
        RuleCRWL RuleCRWLL CRWLTheory
        PairTerm PairTermL .

  op cSymbol : Qid MachineInt -> Constructor [ctor].
  op nilC : -> ConstructorL [ctor] .
  op consC : Constructor ConstructorL -> ConstructorL [ctor] .

  op fSymbol : Qid MachineInt -> Function [ctor] .
  op nilF : -> FunctionL [ctor] .
  op consF : Function FunctionL -> FunctionL [ctor] .

  op <_;_> : Term Term -> PairTermL [ctor] .
  op nilPT : -> PairTermL [ctor] .
  op consPT : PairTerm PairTermL -> PairTermL [ctor] .
  op _->_<=_ : Term Term PairTermL -> RuleCRWL [ctor] .
  op nilR : -> RuleCRWLL [ctor] .
  op consR : RuleCRWL RuleCRWLL -> RuleCRWLL [ctor] .

  op crwlth : ConstructorL FunctionL RuleCRWLL -> CRWLTheory [ctor] .
  op phi : CRWLTheory -> Module .

  var CL : ConstructorL .  var FL : FunctionL .
  var R : RuleCRWL .       var RL : RuleCRWLL .
```

```
var ODS : OpDeclSet .    var VDS : VarDeclSet .
var ES : EquationSet .
var Q Q0 : Qid .         var M N N0 : MachineInt .
var T T1 T2 : Term .     var TL TL1 TL2 : TermList .

op sortCRWL : -> SortDecl .
eq sortCRWL = (sorts 'Var ; 'VarSet ; 'Term ; 'Expr ; 'Equation ;
                     'SolvedPart ; 'ApproxStatement ; 'ProducedPart ;
                     'JoinStatement ; 'JoinPart ; 'Goal ; 'Answer .) .

op subsortCRWL : -> SubsortDeclSet .
eq subsortCRWL = (subsort 'Var < 'VarSet .) (subsort 'Var < 'Term .)
           (subsort 'Term < 'Expr .) (subsort 'Equation < 'SolvedPart .)
           (subsort 'ApproxStatement < 'ProducedPart .)
           (subsort 'JoinStatement < 'JoinPart .) .

op opCRWL : -> OpDeclSet .
eq opCRWL =
     (op 'solution : nil -> 'Answer [ctor] .)
     (op 'maybe-sol : nil -> 'Answer [ctor] .)
     (op 'no-solution : nil -> 'Answer [ctor] .)
     (op 'ok : 'Goal -> 'Answer [none] .)
     (op 'v : 'MachineInt -> 'Var [ctor] .)
     (op 'empty-var : nil -> 'VarSet [ctor] .)
     (op '_U_ : 'VarSet 'VarSet -> 'VarSet
                       [ctor assoc comm id({'empty-var}'VarSet)] .)
     (op 'intersection : 'VarSet 'VarSet -> 'VarSet [none] .)
     (op '_in_ : 'Var 'VarSet -> 'Bool [none] .)
     (op 'new : 'VarSet -> 'Var [none] .)
     (op 'maxIndex : 'VarSet -> 'MachineInt [none] .)
     (op 'goal : 'VarSet 'SolvedPart 'ProducedPart 'JoinPart ->
                                                   'Goal [ctor] .)
     (op 'FAIL : nil -> 'Goal [ctor] .)
     (op 'eq : 'Var 'Term -> 'Equation [ctor] .)
     (op 'empty-eq : nil -> 'SolvedPart [ctor] .)
     (op '__ : 'SolvedPart 'SolvedPart -> 'SolvedPart
                    [ctor assoc comm id({'empty-eq}'SolvedPart)] .)
     (op 'approx : 'Expr 'Term -> 'ApproxStatement [ctor] .)
     (op 'empty-approx : nil -> 'ProducedPart [ctor] .)
     (op '__ : 'ProducedPart 'ProducedPart -> 'ProducedPart
               [ctor assoc comm id({'empty-approx}'ProducedPart)] .)
     (op 'join : 'Expr 'Expr -> 'JoinStatement [ctor comm] .)
     (op 'empty-join : nil -> 'JoinPart [ctor] .)
     (op '__ : 'JoinPart 'JoinPart -> 'JoinPart
                     [ctor assoc comm id({'empty-join}'JoinPart)] .)
     (op 'variables : 'Expr -> 'VarSet [none] .)
     (op 'variables : 'ProducedPart -> 'VarSet [none] .)
```

```
      (op 'variables : 'JoinPart -> 'VarSet [none] .)
      (op 'safeVar : 'Expr -> 'VarSet [none] .)
      (op 'producedVar : 'ProducedPart -> 'VarSet [none] .)
      (op 'demandVar : 'JoinPart -> 'VarSet [none] .)
      (op 'subst : 'Expr 'Var 'Term -> 'Expr [none] .)
      (op 'subst : 'Term 'Var 'Term -> 'Term [none] .)
      (op 'subst : 'SolvedPart 'Var 'Term -> 'SolvedPart [none] .)
      (op 'subst : 'ProducedPart 'Var 'Term -> 'ProducedPart [none] .)
      (op 'subst : 'JoinPart 'Var 'Term -> 'JoinPart [none] .) .

  op repeatL : Qid MachineInt -> QidList .
  op addC : ConstructorL OpDeclSet -> OpDeclSet .
  op addF : FunctionL OpDeclSet -> OpDeclSet .

  eq repeatL(Q, N) = if N > 0 then Q repeatL(Q, _-_(N,1)) else nil fi .
  eq addC(nilC, ODS) = ODS .
  eq addC(consC(cSymbol(Q,N),CL),ODS) =
     addC(CL,(op Q : repeatL('Expr,N) -> 'Expr [ctor] .)
             (op Q : repeatL('Term,N) -> 'Term [ctor] .) ODS) .
  eq addF(nilF,ODS) = ODS .
  eq addF(consF(fSymbol(Q,N),FL),ODS) =
     addF(FL,(op Q : repeatL('Expr,N) -> 'Expr [ctor] .) ODS) .

  op varCRWL : -> VarDeclSet .
  eq varCRWL = (var 'N : 'MachineInt .) (var 'V : 'Var .)
        (var 'V0 : 'Var .) (var 'UVars : 'VarSet .)
        (var 'VS : 'VarSet .) (var 'VS0 : 'VarSet .)
        (var 'EQ : 'Equation .) (var 'SP : 'SolvedPart .)
        (var 'PP : 'ProducedPart .)
        (var 'JP : 'JoinPart .)
        (var 'E : 'Expr .) (var 'E0 : 'Expr .)
        (var 'T : 'Term .) (var 'T0 : 'Term .)   .

  op max : MachineInt MachineInt -> MachineInt .
  op maxC : ConstructorL -> MachineInt .
  op maxF : FunctionL -> MachineInt .
  op addVar : MachineInt VarDeclSet -> VarDeclSet .

  eq max(M,N) = if M > N then M else N fi .
  eq maxC(nilC) = 0 .
  eq maxC(consC(cSymbol(Q,N),CL)) = max(N,maxC(CL)) .
  eq maxF(nilF) = 0 .
  eq maxF(consF(fSymbol(Q,N),FL)) = max(N,maxF(FL)) .
  eq addVar(N,VDS) = if N > 0
        then addVar(_-_(N,1), (var index('A,N) : 'Expr .)
                              (var index('B,N) : 'Expr .)
                              (var index('E,N) : 'Expr .)
```

```
                            (var index('T,N) : 'Term .) VDS)
      else VDS fi .

op eqCRWL : -> EquationSet .
eq eqCRWL =
     (eq 'ok[{'FAIL}'Goal] = {'no-solution}'Answer .)
     (eq 'ok['goal['UVars,'SP,{'empty-approx}'ProducedPart,
                   {'empty-join}'JoinPart]] = {'solution}'Answer .)
     (eq '_U_['V,'V] = 'V .)
     (eq '_in_['V,{'empty-var}'VarSet] = {'false}'Bool .)
     (eq '_in_['V,'_U_['V0,'VS]] = '_or_['_==_['V,'V0],'_in_['V,'VS]] .)
     (eq 'maxIndex[{'empty-var}'VarSet] = {'0}'MachineInt .)
     (eq 'maxIndex['_U_['v['N],'VS]] =
         'if_then_else_fi['_>_['N,'maxIndex['VS]],'N,'maxIndex['VS]] .)
     (eq 'new['VS] = 'v['_+_['maxIndex['VS],{'1}'MachineInt]] .)
     (eq 'intersection['VS,{'empty-var}'VarSet] = {'empty-var}'VarSet .)
     (eq 'intersection['VS,'_U_['V,'VS0]] =
        'if_then_else_fi['_in_['V,'VS],'_U_['V,'intersection['VS,'VS0]],
                         'intersection['VS,'VS0]] .)
     (eq '__['EQ,'EQ] = 'EQ .)
     (eq 'variables['V] = 'V .)
     (eq 'variables[{'empty-approx}'ProducedPart] =
         {'empty-var}'VarSet .)
     (eq 'variables['__['approx['E,'E0],'PP]] =
         '_U_['variables['E],'variables['E0],'variables['PP]] .)
     (eq 'variables[{'empty-join}'JoinPart] = {'empty-var}'VarSet .)
     (eq 'variables['__['join['E,'E0],'JP]] =
         '_U_['variables['E],'variables['E0],'variables['JP]].)
     (eq 'safeVar['V] = 'V .)
     (eq 'producedVar[{'empty-approx}'ProducedPart] =
         {'empty-var}'VarSet .)
     (eq 'producedVar['__['approx['E,'E0],'PP]] =
         '_U_['variables['E0],'producedVar['PP]] .)
     (eq 'demandVar['__['join['V,'V0],'JP]] =
         '_U_['V,'V0,'demandVar['JP]] .)
     (ceq 'demandVar['__['join['V,'E0],'JP]] =
          '_U_['V,'demandVar['JP]]
          if 'not_['E0 : 'Var] = {'true}'Bool .)
     (ceq 'demandVar['__['join['E,'E0],'JP]] = 'demandVar['JP]
          if '_and_['not_['E : 'Var],'not_['E0 : 'Var]]= {'true}'Bool .)
     (eq 'demandVar[{'empty-join}'JoinPart] = {'empty-var}'VarSet .)
     (ceq 'subst['V0,'V,'T] = 'T if '_==_['V0,'V] = {'true}'Bool .)
     (ceq 'subst['V0,'V,'T] = 'V0 if '_=/=_['V0,'V] = {'true}'Bool .)
     (eq 'subst[{'empty-eq}'SolvedPart,'V,'T]= {'empty-eq}'SolvedPart .)
     (eq 'subst['__['eq['V0,'T0],'SP],'V,'T] =
         '__['eq['V0,'subst['T0,'V,'T]],'subst['SP,'V,'T]] .)
     (eq 'subst[{'empty-approx}'ProducedPart,'V,'T] =
```

```
          {'empty-approx}'ProducedPart .)
      (eq 'subst['__['approx['E,'T0],'PP],'V,'T] =
          '__['approx['subst['E,'V,'T],'subst['T0,'V,'T]],
              'subst['PP,'V,'T]] .)
      (eq 'subst[{'empty-join}'JoinPart,'V,'T] =
          {'empty-join}'JoinPart .)
      (eq 'subst['__['join['E,'E0],'JP],'V,'T] =
          '__['join['subst['E,'V,'T],'subst['E0,'V,'T]],
              'subst['JP,'V,'T]] .) .

  op genArgs : Qid MachineInt -> TermList .
  op auxEq1 : Qid MachineInt -> TermList .
  op auxEq2 : MachineInt -> TermList .
  op addEqC : ConstructorL EquationSet -> EquationSet .
  op addEqF : FunctionL EquationSet -> EquationSet .

  eq genArgs(Q,N) = if N > 1 then genArgs(Q,_-_(N,1)),index(Q,N)
                              else index(Q,1) fi .
  eq auxEq1(Q,N) = if N > 0 then auxEq1(Q,_-_(N,1)),Q[index('E,N)]
                             else {'empty-var}'VarSet fi .
  eq auxEq2(N) = if N > 1 then auxEq2(_-_(N,1)),'subst[index('E,N),'V,'T]
                           else 'subst['E1,'V,'T] fi .
  eq addEqC(nilC,ES) = ES .
  eq addEqC(consC(cSymbol(Q,N),CL),ES) = if N == 0
       then addEqC(CL, (eq 'variables[{Q}'Term] = {'empty-var}'VarSet .)
                       (eq 'safeVar[{Q}'Term] = {'empty-var}'VarSet .)
                       (eq 'subst[{Q}'Term,'V,'T] = {Q}'Term .) ES)
       else addEqC(CL, (eq 'variables[Q[genArgs('E,N)]] =
                           '_U_[auxEq1('variables,N)] .)
                       (eq 'safeVar[Q[genArgs('E,N)]] =
                           '_U_[auxEq1('safeVar,N)] .)
                       (eq 'subst[Q[genArgs('E,N)],'V,'T] =
                           Q[auxEq2(N)] .) ES) fi .
  eq addEqF(nilF,ES) = ES .
  eq addEqF(consF(fSymbol(Q,N),FL),ES) = if N == 0
       then addEqF(FL, (eq 'variables[{Q}'Term] = {'empty-var}'VarSet .)
                       (eq 'safeVar[{Q}'Term] = {'empty-var}'VarSet .)
                       (eq 'subst[{Q}'Term,'V,'T] = {Q}'Term .) ES)
       else addEqF(FL, (eq 'variables[Q[genArgs('E,N)]] =
                           '_U_[auxEq1('variables,N)] .)
                       (eq 'safeVar[Q[genArgs('E,N)]] =
                           {'empty-var}'VarSet .)
                       (eq 'subst[Q[genArgs('E,N)],'V,'T] =
                           Q[auxEq2(N)] .) ES) fi .

  op ruleCRWL : -> RuleSet .
  eq ruleCRWL =
```

```
  (crl['ID] : 'goal['UVars,'SP,'PP,'__['join['V,'V],'JP]] =>
              'goal['UVars,'SP,'PP,'JP]
              if 'not_['_in_['V,'producedVar['PP]]] = {'true}'Bool .)
  (crl['BD] : 'goal['UVars,'SP,'PP,'__['join['V,'T],'JP]] =>
              'goal['UVars,'__['eq['V,'T],'subst['SP,'V,'T]],
                           'subst['PP,'V,'T],'subst['JP,'V,'T]]
              if '_and_['_==_['intersection['variables['T],
                                           'producedVar['PP]],
                               {'empty-var}'VarSet],
                         'not_['_in_['V,'_U_['variables['T],
                                           'producedVar['PP]]]]] =
                 {'true}'Bool .)
  (crl['OB1] : 'goal['UVars,'SP,'__['approx['V,'T],'PP],'JP] =>
               'goal['UVars,'__['eq['V,'T],'subst['SP,'V,'T]],
                            'subst['PP,'V,'T],'subst['JP,'V,'T]]
               if '_and_['not_['T : 'Var],
                          'not_['_in_['V,'producedVar['PP]]]] =
                  {'true}'Bool .)
  (crl['OB2] : 'goal['UVars,'SP,'__['approx['V,'T],'PP],'JP] =>
               'goal['UVars,'SP,'subst['PP,'V,'T],'subst['JP,'V,'T]]
               if '_and_['not_['T : 'Var],
                          '_in_['V,'producedVar['PP]]] = {'true}'Bool .)
  (rl['IB] : 'goal['UVars,'SP,'__['approx['T,'V],'PP],'JP] =>
             'goal['UVars,'SP,'subst['PP,'V,'T],'subst['JP,'V,'T]] .)
  (crl['EL] : 'goal['UVars,'SP,'__['approx['E,'V],'PP],'JP] =>
              'goal['UVars,'SP,'PP,'JP]
              if 'not_['_in_['V,'_U_['variables['PP],'variables['JP]]]] =
                 {'true}'Bool .)

  (crl['CY] : 'goal['UVars,'SP,'PP,'__['join['V,'E],'JP]] =>
              {'FAIL}'Goal
              if '_and_['_=/=_['V,'E],'_in_['V,'safeVar['E]]] =
                 {'true}'Bool
  .) .

op _joinTerms_ : TermList TermList -> TermList .
op _approxTerms_ : TermList TermList -> TermList .
op genNewVars : Term MachineInt -> TermList .
op ruleDC1 : ConstructorL -> RuleSet .
op ruleIM : ConstructorL -> RuleSet .
op ruleDC2 : ConstructorL -> RuleSet .
op ruleIIM : ConstructorL -> RuleSet .
op ruleCF1 : ConstructorL -> RuleSet .
op ruleCF2 : ConstructorL -> RuleSet .

eq T1 joinTerms T2 = 'join[T1,T2] .
eq ( T1,TL1 ) joinTerms ( T2,TL2 ) = 'join[T1,T2], (TL1 joinTerms TL2) .
```

```
eq T1 approxTerms T2 = 'approx[T1,T2] .
eq ( T1,TL1 ) approxTerms ( T2,TL2 ) =
     'approx[T1,T2], (TL1 approxTerms TL2) .
eq genNewVars(Q,N) =
      if N > 1 then 'new[Q],genNewVars('_U_[Q,'new[Q]],_-_(N,1))
               else 'new[Q] fi .

eq ruleDC1(nilC) = none .
eq ruleDC1(consC(cSymbol(Q,N),CL)) = if N > 0
  then rl['DC1] : 'goal['UVars,'SP,'PP,
                        '__['join[Q[genArgs('A,N)],
                                  Q[genArgs('B,N)]],'JP]] =>
                  'goal['UVars,'SP,'PP,
                        '__[genArgs('A,N) joinTerms genArgs('B,N),
                        'JP]] .
       ruleDC1(CL)
  else rl['DC1] : 'goal['UVars,'SP,'PP,'__['join[{Q}'Term,
                                                 {Q}'Term],'JP]] =>
                  'goal['UVars,'SP,'PP,'JP] .
       ruleDC1(CL)
  fi .
eq ruleIM(nilC) = none .
eq ruleIM(consC(cSymbol(Q,N),CL)) = if N > 0 then
    crl['IM]: 'goal['UVars,'SP,'PP,'__['join['V,Q[genArgs('E,N)]],'JP]]
     =>
     'goal['_U_[genNewVars('UVars,N),'UVars],
           '__['eq['V,Q[genNewVars('UVars,N)]],
               'subst['SP,'V,Q[genNewVars('UVars,N)]]],
           'subst['PP,'V,Q[genNewVars('UVars,N)]],
           'subst['__[genArgs('E,N) joinTerms genNewVars('UVars,N),'JP],
                  'V,Q[genNewVars('UVars,N)]]]
     if '_and_['_or_['not_[Q[genArgs('E,N)] : 'Term],
                     '_=/=_['intersection['variables[Q[genArgs('E,N)]],
                                          'producedVar['PP]],
                            {'empty-var}'VarSet]],
               '_and_['not_['_in_['V,'producedVar['PP]]],
                      'not_['_in_['V,'safeVar[Q[genArgs('E,N)]]]]]]] =
        {'true}'Bool .
    ruleIM(CL)
  else ruleIM(CL) fi .
eq ruleDC2(nilC) = none .
eq ruleDC2(consC(cSymbol(Q,N),CL)) = if N > 0
  then rl['DC2] : 'goal['UVars,'SP,
                        '__['approx[Q[genArgs('E,N)],Q[genArgs('T,N)]],
                            'PP],
                        'JP] =>
                  'goal['UVars,'SP,
```

```
                       '__[genArgs('E,N) approxTerms genArgs('T,N),'PP],
                        'JP] .
       ruleDC2(CL)
  else rl['DC2] : 'goal['UVars,'SP,'__['approx[{Q}'Term,{Q}'Term],'PP],
                        'JP] =>
                  'goal['UVars,'SP,'PP,'JP] .
       ruleDC2(CL)
  fi .
eq ruleIIM(nilC) = none .
eq ruleIIM(consC(cSymbol(Q,N),CL)) = if N > 0 then
    crl['IIM]: 'goal['UVars,'SP,
                     '__['approx[Q[genArgs('E,N)],'V],'PP],'JP] =>
               'goal['_U_[genNewVars('UVars,N),'UVars],'SP,
                     'subst['__[genArgs('E,N) approxTerms
                                genNewVars('UVars,N),'PP],
                                'V,Q[genNewVars('UVars,N)]],
                     'subst['JP,'V,Q[genNewVars('UVars,N)]]]
               if '_and_['not_[Q[genArgs('E,N)] : 'Term],
                         '_in_['V,'demandVar['JP]]] = {'true}'Bool .
    ruleIIM(CL)
  else ruleIIM(CL) fi .

sorts PairC PairCL .
subsort PairC < PairCL .
op <_`,_> : Constructor Constructor -> PairC [ctor].
op nilP : -> PairCL [ctor].
op consP : PairCL PairCL -> PairCL [ctor assoc id: nilP] .
op addPairs : Constructor ConstructorL -> PairCL .
op pairsL : ConstructorL -> PairCL .
op createTerm : Qid Qid MachineInt -> Term .

eq pairsL(nilC) = nilP .
eq pairsL(consC(cSymbol(Q,N),CL)) =
     consP(addPairs(cSymbol(Q,N),CL),pairsL(CL)) .
eq addPairs(cSymbol(Q,N),nilC) = nilP .
eq addPairs(cSymbol(Q,N),consC(cSymbol(Q0,N0),CL)) =
     consP(< cSymbol(Q,N), cSymbol(Q0,N0) >,addPairs(cSymbol(Q,N),CL)) .
eq createTerm(Q, Q0, N) = if N > 0 then Q[genArgs(Q0,N)]
                                   else {Q}'Term fi .

var PCL : PairCL .
op ruleCF1Aux : PairCL -> RuleSet .
op ruleCF2Aux : PairCL -> RuleSet .
eq ruleCF1(CL) = ruleCF1Aux(pairsL(CL)) .
eq ruleCF1Aux(nilP) = none .
eq ruleCF1Aux(consP(< cSymbol(Q,N), cSymbol(Q0,N0) >,PCL)) =
   rl['CF1]: 'goal['UVars,'SP,'PP,
```

```
                 '__['join[createTerm(Q,'A,N), createTerm(Q0,'B,N0)],
                    'JP]]
          => {'FAIL}'Goal .
   ruleCF1Aux(PCL) .
eq ruleCF2(CL) = ruleCF2Aux(pairsL(CL)) .
eq ruleCF2Aux(nilP) = none .
eq ruleCF2Aux(consP(< cSymbol(Q,N), cSymbol(Q0,N0) >,PCL)) =
  (rl['CF2]: 'goal['UVars,'SP,
                 '__['approx[createTerm(Q,'E,N),
                            createTerm(Q0,'T,N0)],'PP],
                 'JP]
          => {'FAIL}'Goal .)
  (rl['CF2]: 'goal['UVars,'SP,
                 '__['approx[createTerm(Q0,'E,N0),
                            createTerm(Q,'T,N)],'PP],
                 'JP]
          => {'FAIL}'Goal .)
   ruleCF2Aux(PCL) .

var PTL PTL1 : PairTermL .

op findMatch : Term PairTermL -> Term . *** the term is in the list
eq findMatch( T1, consPT(< T1 ; T2 >,PTL)) = T2 .
ceq findMatch( T, consPT(< T1 ; T2 >,PTL)) = findMatch(T, PTL)
    if T =/= T1 .

op length : TermList -> MachineInt .
eq length(T) = 1 .
eq length( (T,TL) ) = 1 + length(TL) .

sort TermSet .
subsort Term < TermSet .

var TS : TermSet .

op empty-term : -> TermSet [ctor] .
op _U_ : TermSet TermSet -> TermSet [ctor assoc comm id: empty-term] .
op delete : Term TermSet -> TermSet .
eq T U T = T .
eq delete(T,empty-term) = empty-term .
eq delete(T1,T1 U TS) = delete(T1,TS) .
ceq delete(T1,T2 U TS) = T2 U delete(T1,TS) if T1 =/= T2 .

op varsInR : RuleCRWL -> TermSet .
op varsInT : Term -> TermSet .
op varsInTL : TermList -> TermSet .
op varsInCond : PairTermL -> TermSet .
```

```
eq varsInR( T1 -> T2 <= PTL ) = varsInT(T1) U varsInT(T2)
                                             U varsInCond(PTL) .
eq varsInT(Q) = empty-term .
eq varsInT({Q}Q0) = empty-term .
eq varsInT(Q[TL]) = if (Q == 'v) and (length(TL) == 1) then Q[TL]
                    else varsInTL(TL) fi .
eq varsInTL(T) = varsInT(T) .
eq varsInTL( (T,TL) ) = varsInT(T) U varsInTL(TL) .
eq varsInCond(nilPT) = empty-term .
eq varsInCond(consPT(< T1 ; T2 >, PTL)) = varsInT(T1) U varsInT(T2) U
                                          varsInCond(PTL) .

sort PairSubst .
op genSubst : TermSet Term -> PairSubst .
op pairS : PairTermL Term -> PairSubst [ctor].
op extractSubst : PairSubst -> PairTermL .
op extractNewVars : PairSubst -> Term .
eq extractSubst(pairS(PTL,T)) = PTL .
eq extractNewVars(pairS(PTL,T)) = T .
eq genSubst(empty-term,T) = pairS(nilPT,T) .
eq genSubst(T1 U TS,T2) =
   pairS(consPT(< T1 ; 'new[T2] >,
             extractSubst(genSubst(delete(T1,TS),
                                   '_U_['new[T2],T2]))),
         extractNewVars(genSubst(delete(T1,TS),'_U_['new[T2],T2]))) .

op applySubstInR : RuleCRWL PairTermL -> RuleCRWL .
op applySubstInT : Term PairTermL -> Term .
op applySubstInTL : TermList PairTermL -> TermList .
op applySubstInCond : PairTermL PairTermL -> PairTermL .
eq applySubstInR( T1 -> T2 <= PTL, PTL1) =
   applySubstInT(T1,PTL1) -> applySubstInT(T2,PTL1) <=
   applySubstInCond(PTL,PTL1) .
eq applySubstInT(Q,PTL) = Q .
eq applySubstInT({Q}Q0,PTL) = {Q}Q0 .
eq applySubstInT(Q[TL],PTL) = if (Q == 'v) and (length(TL) == 1)
     then findMatch(Q[TL],PTL)
     else Q[applySubstInTL(TL,PTL)] fi .
eq applySubstInTL(T,PTL) = applySubstInT(T,PTL) .
eq applySubstInTL( (T,TL), PTL ) =
     (applySubstInT(T,PTL) , applySubstInTL(TL,PTL)) .
eq applySubstInCond(nilPT,PTL) = nilPT .
eq applySubstInCond(consPT(< T1 ; T2 > , PTL),PTL1) =
    consPT(< applySubstInT(T1,PTL1) ; applySubstInT(T2,PTL1) >,
             applySubstInCond(PTL,PTL1)) .

op condToJoin : PairTermL -> TermList .
```

```
  eq condToJoin(nilPT) = {'empty-join}'JoinPart .
  eq condToJoin(consPT(< T1 ; T2 >, PTL)) =
       ('join[T1,T2] , condToJoin(PTL)) .

  op ruleNR1 : RuleCRWLL -> RuleSet .
  op ruleNR1Aux : RuleCRWL Term -> RuleSet .
  op ruleNR2 : RuleCRWLL -> RuleSet .
  op ruleNR2Aux : RuleCRWL Term -> RuleSet .

  eq ruleNR1(nilR) = none .
  eq ruleNR1(consR(R,RL)) =
    ruleNR1Aux(applySubstInR(R,extractSubst(genSubst(varsInR(R),'UVars))),
               extractNewVars(genSubst(varsInR(R),'UVars)))
     ruleNR1(RL) .
  eq ruleNR1Aux(Q[TL] -> T2 <= PTL, T) =
     (rl['NR1] : 'goal['UVars,'SP,'PP,
                       '__['join[Q[genArgs('E,length(TL))],'E],'JP]] =>
                 'goal[T,'SP,'__[genArgs('E,length(TL)) approxTerms TL,
                                 'PP],
                       '__['join[T2,'E],condToJoin(PTL),'JP]] .) .
  eq ruleNR1Aux({Q}Q0 -> T2 <= PTL, T) =
     (rl['NR1] : 'goal['UVars,'SP,'PP,'__['join[{Q}Q0,'E],'JP]] =>
                 'goal[T,'SP,'PP,'__['join[T2,'E],condToJoin(PTL),
                       'JP]] .) .

  eq ruleNR2(nilR) = none .
  eq ruleNR2(consR(R,RL)) =
    ruleNR2Aux(applySubstInR(R,extractSubst(genSubst(varsInR(R),'UVars))),
               extractNewVars(genSubst(varsInR(R),'UVars)))
    ruleNR2(RL) .
  eq ruleNR2Aux(Q[TL] -> T2 <= PTL, T) =
    (crl['NR2] : 'goal['UVars,'SP,
                       '__['approx[Q[genArgs('E,length(TL))],'T],'PP],
                       'JP] =>
                 'goal[T,'SP,
                       '__[genArgs('E,length(TL)) approxTerms TL,
                           'approx[T2,'T],'PP],
                       '__[condToJoin(PTL),'JP]]
                 if '_or_['_in_['T,'demandVar['JP]],'not_['T : 'Var]] =
                    {'true}'Bool .) .
  eq ruleNR2Aux({Q}Q0 -> T2 <= PTL, T) =
    (crl['NR2] : 'goal['UVars,'SP,'__['approx[{Q}Q0,'T],'PP],'JP] =>
                 'goal[T,'SP,'__['approx[T2,'T],'PP],
                       '__[condToJoin(PTL),'JP]]
                 if '_or_['_in_['T,'demandVar['JP]],'not_['T : 'Var]] =
                    {'true}'Bool .) .
```

```
  eq phi(crwlth(CL,FL,RL)) =
     (mod 'CRWLTHEORY is
       including 'MACHINE-INT .
       including 'BOOL .
       sortCRWL
       subsortCRWL
       addC(CL,addF(FL,opCRWL))
       addVar(max(maxC(CL),maxF(FL)),varCRWL)
       none
       addEqC(CL,addEqF(FL,eqCRWL))
       (ruleCRWL ruleDC1(CL) ruleIM(CL) ruleNR1(RL)
                 ruleDC2(CL) ruleIIM(CL) ruleNR2(RL)
                 ruleCF1(CL) ruleCF2(CL))
     endm) .
endfm)
```

Comparing Meseguer's Rewriting Logic with the Logic CRWL*

Miguel Palomino Tarjuelo

Dpto. Sistemas Informáticos y Programación
Facultad de Matemáticas, Universidad Complutense de Madrid, Spain
`miguelpt@eucmos.sim.ucm.es`

Abstract. Meseguer's rewriting logic and the rewriting logic CRWL are two well known approaches to rewriting as logical deduction that, despite some clear similarities, were designed with different objectives. Here we study the relationships between them, both at a syntactic and at a semantic level. It is not possible to establish an entailment system map between them, but both can be naturally simulated in each other. Semantically, there is no embedding between the corresponding institutions. We also use the syntactic results to prove reflective properties of CRWL and to extend those already known for Meseguer's rewriting logic.

1 Introduction

The aim of this paper is to study in detail, and to clarify to some extent, the relationships between two well known approaches to rewriting as logical deduction, namely, José Meseguer's rewriting logic [12], and the constructor-based rewriting logic (CRWL) developed by Mario Rodríguez Artalejo's research group in Madrid [9].

The first of these was proposed as a logical framework wherein to represent other logics, and also as a semantic framework for the specification of languages and systems. The experience accumulated throughout the last years has come to support that original intention; in particular, it has been shown that rewriting logic is a very flexible framework in which many other logics, including first-order logic, intuitionistic logic, linear logic, Horn logic with equality, as well as any other logic with a sequent calculus, can be represented. An important characteristic of these representations that should be stressed is that they are usually quite simple and natural, so that their mathematical properties are often straightforward to derive.

On the other hand, the goal of the constructor-based rewriting logic is to serve as a logical basis for declarative programming languages involving lazy evaluation, offering support, in addition, to non-strict and possibly non-deterministic functions.

---

* Supported by a postgraduate fellowship from the Spanish Ministry for Education, Culture and Sports.

Despite these differences there is a clear resemblance between both logics, namely, the fact that logical deduction is based on rewriting. It seems natural, then, to ask about the relationships between deduction in these logics, and to extend the question so as to encompass whether the corresponding models are also related.

A suitable framework in which to carry out this study is the theory of *general logics* developed by Meseguer [10]. There, a logic is described in a very abstract manner and two separated components are distinguished in it: an *entailment system* and an *institution*, corresponding with the syntactic and the semantic parts of the logic, respectively.

We will begin by studying derivability and, for that, we will try to associate entailment systems to both logics. Unfortunately, it will be proven that there is none corresponding to deduction in CRWL, and so we will be forced to leave the formal framework and undertake more informal simulations of the logics in each other. Although such simulations could be possible by making use of suitable down-level encodings, relying on the analogies between both logics, our interest resides in finding natural and simple simulations which at least show that their expressive power is the same. In addition, these results will be used to study reflective properties of both logics.

After the comparison at the syntactic level, the next step is the study of the corresponding models. Now we will be able to associate an institution to each logic, so this study will take place within the formal framework of the theory of institutions. The main result we will obtain is that models in these logics bear no relation at all, and through its way we will clarify some subtle points regarding the definition of models in Meseguer's rewriting logic.

A detailed account of all the results presented here can be found in [15].

Notation

We will write RL for Meseguer's rewriting logic; more precisely, for the version which uses unsorted and unconditional equational logic as its underlying logic. A signature in RL is a pair (Σ, E) with Σ a ranked alphabet of function symbols and E a set of Σ-equations. Rewriting operates on equivalence classes of terms modulo the set of equations E. We denote by $T_\Sigma(\mathcal{X})$ the Σ-algebra of Σ-terms with variables in a set $\mathcal{X}$, and by $[t]_E$ or just $[t]$ the E-equivalence class of $t \in T_\Sigma(\mathcal{X})$. To indicate that $\{x_1, \ldots, x_n\}$ is the set of variables occurring in t we write $t(x_1, \ldots, x_n)$. Given $t(x_1, \ldots, x_n)$, and terms $u_1, \ldots, u_n$, $t(u_1/x_1, \ldots, u_n/x_n)$ denotes the term obtained from t by *simultaneously substituting* u_i for x_i, $i = 1, \ldots, n$. To simplify notation we denote a sequence of objects $a_1, \ldots, a_n$ by $\overline{a}$.

An RL-theory $\mathcal{R}$ is a 4-tuple $\mathcal{R} = (\Sigma, E, L, \Gamma)$, where (Σ, E) is a signature and Γ is a set of rewrite rules, *labelled* with elements of L, of the form

$$r : [t] \to [t'] \textit{ if } [a_1] \to [b_1] \wedge \ldots \wedge [a_m] \to [b_m].$$

For a complete exposition of RL we refer the reader to [12].

CRWL uses signatures with constructors $\Sigma = C_\Sigma \cup F_\Sigma$, where C_Σ and F_Σ are disjoint sets of constructor and defined function symbols, respectively. $\Sigma_\perp$ refers to the signature which is obtained from Σ by adding a new constructor $\perp$ of arity 0. Given a set $\mathcal{X}$ of variables, we will write $\mathit{Expr}(\Sigma, \mathcal{X})$ for the set of expressions which can be built with Σ and $\mathcal{X}$, and $\mathit{Term}(\Sigma, \mathcal{X})$ for those terms which only make use of C_Σ and $\mathcal{X}$. $\mathit{Expr}_\perp(\Sigma, \mathcal{X})$ and $\mathit{Term}_\perp(\Sigma, \mathcal{X})$, the sets of partial expressions and terms, are defined analogously using $\Sigma_\perp$.

A CRWL-theory is a pair (Σ, Γ), where Σ is a signature with constructors, and Γ is a set of conditional rewrite rules of the form

$$f(t_1, \dots, t_n) \to r \Leftarrow a_1 \bowtie b_1, \dots, a_n \bowtie b_n \quad (m \geq 0),$$

with f a function symbol, and $t_1, \dots, t_n \in \mathit{Term}(\Sigma, \mathcal{X})$. From a given theory two kinds of sentences are derived: reduction statements of the form $a \to b$, and joinability statements $a \bowtie b$ (meaning that there exists a total term to which both a and b reduce). Again, we refer to [9] for a presentation of the CRWL-calculus and its model-theoretic semantics. (Note that in [9] the names “term” and “constructor term” are used instead of “expression” and “term”.)

2 Entailment Systems for RL and CRWL

We begin by studying the notion of derivation in RL and CRWL. This notion is abstractly captured by Meseguer’s *entailment systems*, which are given by a category **Sign** of signatures together with, for each signature Σ, a set of sentences, and an entailment relation $\vdash_\Sigma$ between sets of sentences and sentences, satisfying some conditions. Among those conditions is transitivity: if $\Gamma \vdash_\Sigma \varphi_i$ for all $i \in I$, and $\Gamma \cup \{\varphi_i \mid i \in I\} \vdash_\Sigma \psi$, then $\Gamma \vdash_\Sigma \psi$. Our goal is to associate entailment systems to both RL and CRWL, and then to relate them by maps of entailment systems.

Assigning an entailment system to RL is a relatively straightforward task: either we restrict to unconditional rewrite rules and define $\vdash_\Sigma$ by means of derivation in the RL-calculus [12], or we also consider conditional rules, in which case the RL-calculus must be extended to be able to derive them [15]. The basic rewriting calculi for both logics are shown in Fig. 1.

At first sight, the same two possibilities hold for CRWL. However, a closer look reveals that derivation in the CRWL-calculus [9]. is not transitive. Consider, for example, a signature Σ with $c, d, h \in \Sigma$, function symbols of arities 0, 0, and 1, respectively. Then it can be proved that

$$\begin{aligned} \{\, c \to h(c), h(x) \to h(d), h(x) \to h(d) \Leftarrow x \bowtie x, \,\} &\vdash_{\mathrm{CRWL}} c \to h(d) \\ \{\, c \to h(c), h(x) \to h(d) \Leftarrow x \bowtie x \,\} &\vdash_{\mathrm{CRWL}} h(x) \to h(d), \end{aligned}$$

but

$$\{\, c \to h(c), h(x) \to h(d) \Leftarrow x \bowtie x \,\} \not\vdash_{\mathrm{CRWL}} c \to h(d).$$

The reason for this is that the CRWL-calculus is sound and complete with respect to validity in models *only* under totally defined valuations [9]. In particular, in

— For an RL-theory (Σ, E, L, Γ) —

$$\frac{}{[t]\rightarrow[t]} \qquad \frac{[t_1]\rightarrow[t'_1] \ \ldots \ [t_n]\rightarrow[t'_n]}{[f(\overline{t})]\rightarrow[f(\overline{t'})]} \qquad \frac{[t]\rightarrow[t'] \ \ [t']\rightarrow[t'']}{[t]\rightarrow[t'']}$$

$$\frac{\begin{array}{c}[w_1]\rightarrow[w'_1] \ \ldots \ [w_n]\rightarrow[w'_n] \\ [a_1(\overline{w}/\overline{x})]\rightarrow[b_1(\overline{w}/\overline{x})] \ \ldots \ [a_m(\overline{w}/\overline{x})]\rightarrow[b_m(\overline{w}/\overline{x})]\end{array}}{[t(\overline{w}/\overline{x})]\rightarrow[t'(\overline{w'}/\overline{x})]} \ \text{if } r{:}[t]\rightarrow[t'] \ \textit{if} \ \ [a_1]\rightarrow[b_1]\wedge\ldots\wedge[a_n]\rightarrow[b_n]\in\Gamma$$

— For a CRWL-theory (Σ, Γ) —

$$\frac{}{e\rightarrow\bot} \qquad \frac{}{e\rightarrow e} \qquad \frac{e_1\rightarrow e'_1 \ \ldots \ e_n\rightarrow e'_n}{h(\overline{e})\rightarrow h(\overline{e'})}$$

$$\frac{C}{l\rightarrow r} \ \text{for } l\rightarrow r\Leftarrow C \text{ an instance of a rule of } \Gamma \text{ with partial terms}$$

$$\frac{e\rightarrow e' \ \ e'\rightarrow e''}{e\rightarrow e''} \qquad \frac{a\rightarrow t \ \ b\rightarrow t}{a\bowtie b} \ \text{if } t \text{ is a total term}$$

Fig. 1. Rules of deduction of the RL and CRWL rewriting calculi

the second entailment above, $h(x) \rightarrow h(d)$ means that $h(t)$ rewrites to $h(d)$ just for those instances where a total term t is substituted for x. This proves that the relation $\vdash_{\mathrm{CRWL}}$ is not transitive and, therefore, we are not going to be able to build an entailment system based on the CRWL rewriting calculus.

3 Simulating CRWL in RL

As there does not exist an entailment system associated to the CRWL-calculus, we cannot define a map of entailment systems as intended. In the following we will be pleased just with presenting how entailment in CRWL can be simulated in RL. The set of labels of an RL-theory does not take part in the entailment process, and so it is omitted; the same convention will also be adopted in Sects. 4 and 5.

Of course, every CRWL-theory T can be trivially "simulated" in RL by means of an RL-theory T' with a constant c_t for each term (and each expression) t in T, and with axioms $c_t \rightarrow c_{t'}$ whenever $t \rightarrow t'$. But such a T' is not computable in general, and this is a property that will be required in Sect. 6 when we apply the simulation to the study of reflection in CRWL. And so we must look for another construction.

The idea is to associate to every CRWL-theory T a theory T' in RL (whose set of equational axioms will be empty) in which all the operations in T, together with a new constant $\bot$, are available, plus one rule for each axiom in T and, perhaps, some more rules coping with the rules of deduction of the CRWL-calculus. Since rules in CRWL can only be instantiated with terms and not expressions, we introduce a unary relation *pterm* (technically, a unary function) and a constant *true* to distinguish them in RL. One immediate rule defining *pterm* is $\mathit{pterm}(\bot) \rightarrow \mathit{true}$; however, how to express that variables are also partial terms? The obvious rule $\mathit{pterm}(x) \rightarrow \mathit{true}$ is clearly not valid: everything would be a partial term! This means that we must add the CRWL variables to the signature of T' as constants, and use a new set $\mathcal{X}$ of variables for RL. This way, assuming variables in CRWL belong to a set $\mathcal{V}$, the rules defining *pterm* will be

$$\begin{array}{l} \mathit{pterm}(\bot) \to \mathit{true} \\ \mathit{pterm}(v_i) \to \mathit{true} \quad (\forall v_i \in \mathcal{V}) \\ \mathit{pterm}(h(x_1, \ldots, x_n)) \to \mathit{true} \\ \qquad \mathit{if}\ \ \mathit{pterm}(x_1) \to \mathit{true} \wedge \ldots \wedge \mathit{pterm}(x_n) \to \mathit{true} \quad (\forall h \in C^n_{\Sigma}, n \in \mathbb{N}), \end{array}$$

In a similar way, two more predicates, *pexpr* and *tterm*, dealing with partial expressions and total terms, are defined [15].

As a side effect, rewriting in CRWL can no longer be simulated in RL directly through the rewriting relation. Consider, for example, the theory of natural numbers in CRWL, with 0 a constructor and + a function symbol. In RL, $\mathit{pterm}(0+0)$ should not rewrite to *true*; however, with the usual definitions, $0 + 0 \to 0$ and by congruence $\mathit{pterm}(0 + 0) \to \mathit{pterm}(0)$, and this last term must reduce to *true*. Therefore, a rewrite in CRWL will be simulated through a binary relation R so that $e \to e'$ in CRWL if and only if $R(e, e') \to \mathit{true}$ in RL. In a similar way, strict equalities $a \bowtie b$ will be simulated through a binary relation $\bowtie$.

It just remains to translate the rules of deduction of the CRWL-calculus, which is straightforward. For example, the bottom rule stating that every expression is reducible to $\bot$ is written

$$R(x, \bot) \to \mathit{true}\ \ \mathit{if}\, \mathit{pexpr}(x) \to \mathit{true},$$

whereas the joinability rule

$$\frac{a \to t \qquad b \to t}{a \bowtie b} \quad \text{if } t \text{ is a total term}$$

results in

$$x \bowtie y \to \mathit{true}\ \ \mathit{if}\, R(x, z) \to \mathit{true} \wedge R(y, z) \to \mathit{true} \wedge \mathit{tterm}(z) \to \mathit{true}.$$

Finally, to every rule $l(\overline{v}) \to r(\overline{v}) \Leftarrow a_1(\overline{v}) \bowtie b_1(\overline{v}), \ldots, a_m(\overline{v}) \bowtie b_m(\overline{v})$ in the CRWL-theory, we associate the following rule in RL

$$\begin{array}{l} R(l(\overline{x}), r(\overline{x})) \to \mathit{true} \\ \qquad \mathit{if}\ \ a_1(\overline{x}) \bowtie b_1(\overline{x}) \to \mathit{true} \wedge \ldots \wedge a_m(\overline{x}) \bowtie b_m(\overline{x}) \to \mathit{true} \wedge \\ \qquad\quad \mathit{pterm}(x_1) \to \mathit{true} \wedge \ldots \wedge \mathit{pterm}(x_n) \to \mathit{true}, \end{array}$$

where each CRWL variable v_i (a constant in the RL-theory) has been substituted by the variable x_i. This last rule corresponds with the reduction rule in the CRWL-calculus, and the condition that program rules in CRWL can only be instantiated with terms is taken care of by demanding $\mathit{pterm}(x) \to \mathit{true}$ for all the variables appearing in it.

We will write $\alpha(T)$ for the RL-theory associated to a CRWL-theory T. The following proposition ensures us that the translation is correct.

Proposition 1. *Given a CRWL-theory* $T = (\Sigma, \Gamma)$ *such that* $\alpha(T) = (\Sigma', \emptyset, \Gamma')$, *then, if* $l, r, a, b \in T_{\Sigma'}(\mathcal{X})$*:*

$$\begin{array}{l} l, r \in \mathit{Expr}_{\bot}(\Sigma, \mathcal{V})\ \ \mathit{and}\ \ T \vdash_{\mathrm{CRWL}} l \to r \iff \alpha(T) \vdash_{\mathrm{RL}} R(l, r) \to \mathit{true}; \\ a, b \in \mathit{Expr}_{\bot}(\Sigma, \mathcal{V})\ \ \mathit{and}\ \ T \vdash_{\mathrm{CRWL}} a \bowtie b \iff \alpha(T) \vdash_{\mathrm{RL}} a \bowtie b \to \mathit{true}. \end{array}$$

4 Simulating RL in CRWL

We now embark ourselves on findind the converse simulation of RL in CRWL. We are still interested in a computable and simple translation, and the idea for this is very similar to that of the previous section. Now, however, there are no terms and expressions to distinguish, and so predicates such as *pterm* are no longer necessary; as a consequence, we will be able to use the same set $\mathcal{X}$ of variables for both logics. The fact that only joinability statements are allowed to appear in the condition of a rewrite rule in CRWL forces us to represent, as in Sect. 3, the rewriting relation in RL through a binary relation R in CRWL, so that $t \to t'$ in RL if and only if $R(t,t') \to \mathit{true}$ in CRWL; rewriting modulo a set of equations will be handled by transforming each equation $t = t'$ into the rewrites $t \to t'$ and $t' \to t$.

More precisely, given a signature (Σ, E) in RL we associate to it a CRWL-theory over the signature Σ' with $C_{\Sigma'} = \Sigma \cup \{\mathit{true}\}$ and $F_{\Sigma'} = \{R\}$, with *true* and R of arities 0 and 2, respectively. The rules in the theory are

$$\begin{aligned} R(x_1, x_2) &\to \mathit{true} \Leftarrow x_1 \bowtie x_2, \\ R(x, y) &\to \mathit{true} \Leftarrow R(x, z) \bowtie \mathit{true}, R(z, y) \bowtie \mathit{true}, \end{aligned}$$

and, for each $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$R(f(x_1, \ldots, x_n), f(y_1, \ldots, y_n)) \to \mathit{true} \Leftarrow R(x_1, y_1) \bowtie \mathit{true}, \ldots, R(x_n, y_n) \bowtie \mathit{true},$$

mimicking the reflexivity, transitivity, and congruence rules in the RL-calculus, together with

$$R(t, t') \to \mathit{true} \qquad \text{and} \qquad R(t', t) \to \mathit{true},$$

for every $t = t' \in E$. The goal of the condition in the rule corresponding to reflexivity is to avoid instantiating it with terms containing $\perp$, which have no meaning in RL.

A conditional rewrite rule

$$[l] \to [r] \text{ if } [a_1] \to [b_1] \wedge \ldots \wedge [a_m] \to [b_m]$$

over (Σ, E) in RL is then translated to

$$R(l, r) \to \mathit{true} \Leftarrow R(a_1, b_1) \bowtie \mathit{true}, \ldots, R(a_m, b_m) \bowtie \mathit{true},$$

where l, r, a_i, b_i are arbitrary members of $[l]$, $[r]$, $[a_i]$ and $[b_i]$, respectively.

In fact, the previous definitions must be slightly modified due to some technical details. In a conditional rewrite rule $l \to r \Leftarrow C$ in CRWL, l must be linear, and it is obvious that with the above definitions this property is not ensured for the translation of equations and rewrite rules; therefore, those rules must be "linearised" (see [1, 15]).

In what follows, we write $\beta(T)$ for the CRWL-theory associated to a RL-theory T. The next proposition guarantees the correctness of the translation.

Proposition 2. *Given any RL-theory $T = (\Sigma, E, \Gamma)$, and $l, r \in T_\Sigma(\mathcal{X})$:*

$$\begin{aligned} T \vdash_{\mathrm{RL}} [l] \to [r] &\iff (\exists l' \in [l], \exists r' \in [r]) \;\; \beta(T) \vdash_{\mathrm{CRWL}} R(l', r') \to \mathit{true} \\ &\iff (\forall l' \in [l], \forall r' \in [r]) \;\; \beta(T) \vdash_{\mathrm{CRWL}} R(l', r') \to \mathit{true}. \end{aligned}$$

5 Simulating Rewriting Logic over Membership Equational Logic in CRWL

The results presented in Sect. 4 can be extended to the case in which the underlying equational logic for Meseguer's rewriting logic is membership equational logic [13]; we write MERL for it. As there, a binary function symbol R can be used to simulate the rewriting relation in CRWL: the goal in mind is having $t \to t'$ in MERL if and only if $R(t,t') \to \mathit{true}$ in CRWL. Now, in addition, a binary function symbol I is needed to represent equality in membership equational logic ($t = t' \iff I(t,t') \to \mathit{true}$), as well as another one M to simulate memberships ($t : s \iff M(t,s) \to \mathit{true}$).

A further distinction concerns the treatment of variables. CRWL has no types so, given the function symbols of a membership equational signature, we could use them to build more terms than those that we would have in the original logic and, moreover, the use of such terms would enlarge the set of provable statements in a theory. So it will be necessary to distinguish somehow the well-typed terms and, for that, we are forced again to represent variables in MERL as constants in CRWL. The task of recognising well-typed terms will be carried out by a binary function symbol *wtterm* in such a way that $\mathit{wtterm}(t,k) \to \mathit{true}$ if and only if the term t has kind k. We can use $\mathcal{V}$ for the set of variables in MERL and $\mathcal{X}$ for the variables in CRWL.

Let then $((K,\Sigma,S),E)$ be a MERL signature, i.e., a membership equational logic theory. We associate to it a CRWL-theory with signature Σ' such that $C_{\Sigma'} = \Sigma \cup \{\mathit{true}\} \cup K \cup S \cup \mathcal{V}$ (the elements of K, S and $\mathcal{V}$ are constants) and $F_{\Sigma'} = \{\mathit{wtterm}, R, I, M\}$. The rules defining *wtterm* are:

$$\begin{array}{ll} \mathit{wtterm}(v,k) \to \mathit{true} & (\forall v \in \mathcal{V} \text{ of kind } k \in K), \\ \mathit{wtterm}(c,k) \to \mathit{true} & (\forall c : k \in \Sigma), \end{array}$$

and, for all $f : k_1 \dots k_n \to k \in \Sigma$,

$$\begin{array}{l} \mathit{wtterm}(f(x_1,\dots,x_n),k) \to \mathit{true} \\ \qquad \Leftarrow \mathit{wtterm}(x_1,k_1) \bowtie \mathit{true}, \dots, \mathit{wtterm}(x_n,k_n) \bowtie \mathit{true}. \end{array}$$

The translation of the rules of deduction of the equational and rewriting calculus goes along the same lines as that of Sect. 4. For example, modus ponens for an equation

$$\begin{array}{l} (\forall \mathcal{W})\ t(\overline{v}) = t'(\overline{v}) \Leftarrow a_1(\overline{v}) = b_1(\overline{v}) \wedge \dots \wedge a_m(\overline{v}) = b_m(\overline{v}) \wedge \\ \qquad\qquad\qquad\qquad\qquad w_1(\overline{v}) : s_1 \wedge \dots \wedge w_p(\overline{v}) : s_p, \end{array}$$

with $\mathcal{W} \subseteq \mathcal{V}$ containing all variables in $\overline{v}$, is simulated by means of (the linearised version of) the rule

$$\begin{array}{rl} I(t(\overline{x}), t'(\overline{x})) \to \mathit{true} \Leftarrow & I(a_1(\overline{x}), b_1(\overline{x})) \bowtie \mathit{true}, \dots, I(a_m(\overline{x}), b_m(\overline{x})) \bowtie \mathit{true}, \\ & M(w_1(\overline{x}), s_1) \bowtie \mathit{true}, \dots, M(w_p(\overline{x}), s_p) \bowtie \mathit{true}, \\ & \mathit{wtterm}(x_1,k_1) \bowtie \mathit{true}, \dots, \mathit{wtterm}(x_n,k_n) \bowtie \mathit{true}, \end{array}$$

where k_i is the kind of v_i; and similarly (replacing I with M) for memberships.

The translation of a rewrite rule

$$[l(\overline{v})] \to [r(\overline{v})] \ \mathit{if} \ [a_1(\overline{v})] \to [b_1(\overline{v})] \wedge \ldots [a_m(\overline{v})] \to [b_m(\overline{v})]$$

is

$$\begin{aligned} R(l(\overline{x}), r(\overline{x})) \to \mathit{true} \Leftarrow & \; R(a_1(\overline{x}), b_1(\overline{x})) \bowtie \mathit{true}, \ldots, R(a_m(\overline{x}), b_m(\overline{x})) \bowtie \mathit{true}, \\ & \; \mathit{wtterm}(x_1, k_1) \bowtie \mathit{true}, \ldots, \mathit{wtterm}(x_n, k_n) \bowtie \mathit{true}, \end{aligned}$$

with k_i the kind of v_i.

In what follows, we will write $\delta(T)$ for the CRWL-theory associated to a MERL-theory T. We have the following two main results.

Proposition 3. *Let T be an RL-theory with signature $((K, \Sigma, S), E)$, $\delta(T) = (\Sigma', \Gamma')$, and let $t, t', s \in Expr_\perp(\Sigma', \mathcal{X})$:*

$$\delta(T) \vdash_{\mathrm{CRWL}} I(t, t') \to \mathit{true} \Longleftrightarrow (\exists k \in K)\; t, t' \in T_\Sigma(\mathcal{V})_k \ \mathit{and} \ E \vdash (\forall \mathcal{V})\; t = t';$$
$$\delta(T) \vdash_{\mathrm{CRWL}} M(t, s) \to \mathit{true} \Longleftrightarrow (\exists k \in K)\; t \in T_\Sigma(\mathcal{V})_k, s \in S_k \ \mathit{and} \ E \vdash (\forall \mathcal{V})\; t : s.$$

Proposition 4. *Given any RL-theory T with signature $((K, \Sigma, S), E)$ and set of rules Γ, and given $l, r \in T_\Sigma(\mathcal{V})$, the following are equivalent:*

1. $T \vdash_{\mathrm{MERL}} [l] \to [r]$;
2. $(\exists l' \in [l], \exists r' \in [r])\; \delta(T) \vdash_{\mathrm{CRWL}} R(l', r') \to \mathit{true}$;
3. $(\forall l' \in [l], \forall r' \in [r])\; \delta(T) \vdash_{\mathrm{CRWL}} R(l', r') \to \mathit{true}$.

6 Reflection in CRWL and in RL

Intuitively, a reflective logic is a logic in which important aspects of its metatheory, such as the concepts of theory and entailment, can be represented and reasoned about *in* the logic. A general axiomatic notion of reflective logic was proposed by Clavel and Meseguer [3, 5]. The notion is itself expressed in terms of the notion of an entailment system.

Given an entailment system $\mathcal{E}$ and a nonempty set of theories $\mathcal{C}$ in it, a theory U is $\mathcal{C}$-universal if there is a function, called a representation function,

$$\overline{(\_ \vdash \_)} : \bigcup_{T \in \mathcal{C}} (\{T\} \times \mathit{sen}(T)) \to \mathit{sen}(U),$$

such that for each $T \in \mathcal{C}$, $\varphi \in \mathit{sen}(T)$,

$$T \vdash \varphi \quad \text{iff} \quad U \vdash \overline{T \vdash \varphi}.$$

If, in addition, $U \in \mathcal{C}$, then the entailment system $\mathcal{E}$ is called $\mathcal{C}$-reflective. Finally, a reflective logic is a logic whose entailment system is $\mathcal{C}$-reflective for $\mathcal{C}$ the class of all finitely presentable theories in the logic. The condition that the representation function be computable and injective has also been required recently [6].

RL has been proved to be reflective in [3, 5, 6] and we will use this result to obtain an analogous one for CRWL. Strictly speaking, the notion of reflection does not apply to CRWL as it was observed in Sect. 3 that CRWL does not have an associated entailment system. Even in the case of RL, except for the original result in [3], subsequent generalizations do not immediately fit within the formal definition of reflection as they allow conditional sentences on the left of the entailment relation but not on the right. For this reason, in what follows we consider a "loose" definition of reflection (making use of some kind of "weak" entailment system) general enough to encompass all the cases just discussed.

Let $\mathcal{U}$ be a universal theory for RL; we will use it to prove that CRWL is also reflective. In Sects. 3 and 4 we defined mappings α and β that map a theory T in CRWL, respectively in RL, to a theory in RL, respectively in CRWL, which simulates the behaviour of T. By abuse of notation we will also use α and β to name the translation of sequents defined in those sections. (Recall that $\alpha(T)$ is *not* obtained by simply applying α to every $\varphi \in T$, and analogously for β.) Then, given an arbitrary CRWL-theory T and a statement φ, we have the following chain of equivalences:

$$\begin{aligned} T \vdash_{\mathrm{CRWL}} \varphi &\iff \alpha(T) \vdash_{\mathrm{RL}} \alpha(\varphi) \\ &\iff \mathcal{U} \vdash_{\mathrm{RL}} \overline{\alpha(T) \vdash \alpha(\varphi)} \\ &\iff \beta(\mathcal{U}) \vdash_{\mathrm{CRWL}} \beta(\overline{\alpha(T) \vdash \alpha(\varphi)}), \end{aligned}$$

which proves that $\beta(\mathcal{U})$ is universal in CRWL (and, since β was quite simple, no much more complicated than the original $\mathcal{U}$).

As it has been pointed out previously, the reflective results about RL have been proved only for special cases. The one we need here is that in which the underlying equational logic is unsorted and the rules are conditional [6]. Moreover, the results presented here allow us to extend the reflective results to the full power of MERL. For that we can consider that α yields theories in MERL, because unsorted equational logic is naturally embeddable in membership equational logic, and let δ be the function defined in Sect. 5 such that, for T a MERL-theory, $T \vdash_{\mathrm{MERL}} \varphi \iff \delta(T) \vdash_{\mathrm{CRWL}} \delta(\varphi)$. Then, for $\mathcal{W}$ any universal theory in CRWL (like $\beta(\mathcal{U})$, for example),

$$\begin{aligned} T \vdash_{\mathrm{MERL}} \varphi &\iff \delta(T) \vdash_{\mathrm{CRWL}} \delta(\varphi) \\ &\iff \mathcal{W} \vdash_{\mathrm{CRWL}} \overline{\delta(T) \vdash \delta(\varphi)} \\ &\iff \alpha(\mathcal{W}) \vdash_{\mathrm{RL}} \alpha(\overline{\delta(T) \vdash \delta(\varphi)}), \end{aligned}$$

which proves that $\alpha(W)$ is universal in MERL.

In Maude, a specification and programming language based on RL, reflection is exploited systematically to extend the language with program transformation methods and internal strategies [4]. In particular, a flexible and robust module algebra incorporating parameterisation and object-oriented features into the language has been built in [8]. In $\mathcal{TOY}$, an experimental language and system that implements the CRWL paradigm, up til now reflection has played no role at all. A possible reason could be that reflection on its own were not a sufficient condition for many results, but required the existence of other logical properties. On the other hand, it could simply be that more attention has to be paid to it, so that a closer look may reveal some possible applications.

7 An Institution for CRWL

In this section we leave behind our study of the entailment relation and turn our attention to models and satisfaction. Our interest consists in associating suitable institutions to both CRWL and RL and, thereafter, to relate them via maps of institutions with "good" properties.

A first step towards this direction is taken in [14], where an institution is associated to CRWL. This institution is defined with the goal of providing a basis for the semantics of modules in CRWL, and restricts its attention to a class of particular term algebras. Since our objective is more general, we do not place such a limitation and define $\mathcal{I}_{\text{CRWL}} = (\mathbf{Sign}, \mathit{sen}, \mathbf{Mod}, \models)$ as follows:

- **Sign**: the category of signatures with constructors and signature morphisms (for the definition of signature morphisms, see [14]);
- $\mathit{sen} : \mathbf{Sign} \to \mathbf{Set}$ the functor assigning to each signature Σ the set of all conditional rewrite rules over it, and to each signature morphism, its extension to rewrite rules;
- $\mathbf{Mod} : \mathbf{Sign}^{\text{op}} \to \mathbf{Cat}$ the functor assigning to each signature the category of CRWL-algebras and homomorphisms over it, and to each $\sigma : \Sigma \to \Sigma'$ the forgetful functor taking $\mathcal{A}' \in |\mathbf{Mod}(\Sigma')|$ to the CRWL-algebra $\mathcal{A}'_\sigma$ with the same underlying poset and such that $h^{\mathcal{A}'_\sigma} = \sigma(h)^{\mathcal{A}'}$ for all $h \in \Sigma$, and which is the identity over homomorphisms;
- $\models$ the satisfaction relation in CRWL.

Proposition 5. *$\mathcal{I}_{\text{CRWL}}$ is an institution.*

It can be proved that the category $\mathbf{Mod}(T)$ has products for every CRWL-theory T; it is not complete, however, as in Sect. 9 it is shown that, in general, $\mathbf{Mod}(T)$ does not have equalizers. $\mathcal{I}_{\text{CRWL}}$ is also a semiexact institution [15].

8 An Institution for RL

The task of assigning an institution to RL is harder than expected. The first and more natural idea is to define the category of signatures **Sign** as the category of equational theories and theory morphisms, and the functor *sen* to map any such theory to the set of conditional rewrite rules over it. Since there are also notions of model and satisfaction in RL [12], the desired institution seems to be at hand. However, when one tries to put together the various components of the institution, problems start to arise. In the first place, the notion of satisfaction in RL is defined only for unconditional rewrite rules, so our first task must be to extend its definition so as to encompass the conditional ones. Although there are at least two possible ways in which this could be done, this is a relatively minor problem which can be solved by mirroring the definition of $\mathcal{R}$-systems (see [15]).

A far more serious problem is posed by the functor $\mathbf{Mod} : \mathbf{Sign}^{\mathrm{op}} \to \mathbf{Cat}$, mapping signatures to models. The difficulty resides in the fact that, in RL, models are assigned directly to RL-theories instead of signatures, as it is customary in other logics. One obvious solution would be to consider a signature (Σ, E) as a theory $\mathcal{R} = (\Sigma, E, \emptyset, \emptyset)$ with empty set of axioms (and labels), and to map (Σ, E) to the category $\mathcal{R}$-**Sys** of models of $\mathcal{R}$ [12]. But this approach presents an important drawback. Up to this point in the paper, we have omitted any explicit mention of the set of labels of an RL-theory. Although this was a safe convention when talking about deduction, it is not longer the case when our interest shifts to models. Due to the set of labels L in an RL-theory $\mathcal{R} = (\Sigma, E, L, \Gamma)$, the elements of Γ become special, *labelled* rewrite rules. These rules force $\mathcal{R}$-systems to have a certain internal structure: not only $\mathcal{R}$-systems must satisfy them, but also must associate to them a *distinguished* interpretation (natural transformation) that must be preserved by homomorphisms[1]. When considering a signature as a theory with empty sets of axioms, we are not taking into account labelled rewrite rules. This way, homomorphisms are not subjected to preserve any rewrite rule and the categories $\mathbf{Mod}(\Gamma)$ and $\mathcal{R}$-**Sys** turn out to be different, in opposition to our original intention.

Keeping **Sign** as the category of equational theories, there are other possibilities as to how to define **Mod** (and even $\models$) but, since they cannot reflect the distinction between labelled rules belonging to RL-theories and unlabelled rules, all of them are bound to failure (see [15] for a detailed discussion). For this reason we are led to an institution in which the category **Sign** subsumes all the information of an RL-theory. More precisely, we define $\mathcal{I}_{\mathrm{RL}} = (\mathbf{Sign}, \mathit{sen}, \mathbf{Mod}, \models)$ where:

- **Sign** is the *discrete* category of RL-theories;
- $\mathit{sen} : \mathbf{Sign} \to \mathbf{Set}$ maps each RL-theory to its corresponding set of conditional rewrite rules;
- $\mathbf{Mod} : \mathbf{Sign}^{\mathrm{op}} \to \mathbf{Cat}$ maps an RL-theory $\mathcal{R}$ to the category $\mathcal{R}$-**Sys**;
- $\models$ the satisfaction relation conveniently extended to conditional rewrite rules as discussed above.

Since **Sign** is discrete, this trivially defines an institution. Admittedly, this restriction seems to be not justified. In fact, two types of morphisms of RL-theories are proposed in [11]. Basically, they are equational theory morphisms "preserving" the rules in the RL-theories; a complete study of the institution $\mathcal{I}_{\mathrm{RL}}$ extended with these morphisms will be undertaken in a future occasion. For our purposes, the present definition is general enough as it stands, and its extension would not modify the use we will make of it in the next section.

There exist other institutions associated to (varieties of) RL in the literature, e.g., [2, 7]; in these papers, the objects in the category of signatures are the sets

---

[1] In particular, the same rule can appear *twice* in an RL-theory $\mathcal{R}$ under two different labels. $\mathcal{R}$-systems are then forced to provide two, possibly different, interpretations for the same rule, each of them to be preserved by the homomorphisms.

of function symbols, without any rules. As a consequence of this simplicity and because of the reasons we have mentioned above, the general categorical models of RL must be somehow restricted, and the choice in these two papers is to require them to be preorders instead of general categories.

9 Searching for Embeddings

We would like to relate the institutions $\mathcal{I}_{\mathrm{CRWL}}$ and $\mathcal{I}_{\mathrm{RL}}$ by means of a map of institutions $(\Phi, \alpha, \beta) : \mathcal{I}_{\mathrm{CRWL}} \to \mathcal{I}_{\mathrm{RL}}$ having nice properties, in such a way that it indicated that $\mathcal{I}_{\mathrm{CRWL}}$ could be considered as a subinstitution of $\mathcal{I}_{\mathrm{RL}}$. The formal definition of subinstitution appeared originally in [10] and has been further generalized in subsequent articles. One of those extensions was introduced by Meseguer in [13], where it is called an *embedding*. The only requirement imposed on a map of institutions $(\Phi, \alpha, \beta) : \mathcal{I} \to \mathcal{I}'$ to be an embedding is that for each $T \in \mathbf{Th}_{\mathcal{I}}$, the functor $\beta_T : \mathbf{Mod}'(\Phi(T)) \to \mathbf{Mod}(T)$ must be an equivalence of categories.

We will show, however, that there is no embedding from $\mathcal{I}_{\mathrm{CRWL}}$ into $\mathcal{I}_{\mathrm{RL}}$. For that it will be enough to find a categorical property which is preserved by an equivalence of categories and a theory $T \in \mathbf{Th}_{\mathrm{CRWL}}$ such that $\mathbf{Mod}_{\mathrm{RL}}(\Phi(T))$, but not $\mathbf{Mod}_{\mathrm{CRWL}}(T)$, has the property.

Let Σ be a signature with constructors such that $C_\Sigma = \emptyset$ and F_Σ consists of just two constants f_1 and f_2, $\Gamma = \{f_2 \to x \Leftarrow f_1 \bowtie f_1\}$, and consider the CRWL-theory $T = (\Sigma, \Gamma)$. We define two CRWL-algebras over Σ: $\mathcal{A}$ given by the set $D^{\mathcal{A}} = \{\perp, a_1, a_2\}$ with partial order $\perp \sqsubseteq a_1 \sqsubseteq a_2$ and the cones $f_1^{\mathcal{A}} = \langle a_1 \rangle$ and $f_2^{\mathcal{A}} = \langle \perp \rangle$; and $\mathcal{B}$ with $D^{\mathcal{B}} = \{\perp, b_1\}$ and the cones $f_1^{\mathcal{B}} = f_2^{\mathcal{B}} = \langle \perp \rangle$. $\mathcal{A}, \mathcal{B} \in \mathbf{Mod}_{\mathrm{CRWL}}(T)$ trivially, because they do not satisfy the condition $f_1 \bowtie f_1$.

Let us now define two CRWL-homomorphisms $F, G : \mathcal{A} \to \mathcal{B}$, given by:

$$F(x) = \langle \perp \rangle \ \text{ and } \ G(x) = \begin{cases} \langle \perp \rangle \text{ if } x = \perp, a_1 \\ \langle b_1 \rangle \text{ if } x = a_2. \end{cases}$$

Clearly, F and G preserve both f_1 and f_2, so that they are actually homomorphisms. It can be shown that there does not exist the equalizer of F and G.

In contrast with what happens in CRWL, equalizers can be built in $\mathcal{R}$-**Sys** by mimicking the construction in **Cat**[15]. Since an equivalence of categories preserves limits, we have:

Proposition 6. *$\mathcal{I}_{\mathrm{CRWL}}$ is not embeddable in $\mathcal{I}_{\mathrm{RL}}$.*

What about the other way around? Can we embed $\mathcal{I}_{\mathrm{RL}}$ in $\mathcal{I}_{\mathrm{CRWL}}$? In order to prove that RL cannot be embedded in CRWL we have to find an RL-theory T such that $\mathbf{Mod}_{\mathrm{RL}}(T)$ has a categorical property that no category of models in CRWL has. For that, note that for any CRWL-theory T there exists a CRWL-algebra $\mathcal{A} \in |\mathbf{Mod}_{\mathrm{CRWL}}(T)|$ with an infinite number of automorphisms. Simply consider $\mathcal{A}$ given by $D^{\mathcal{A}} = \{\perp, a, b_1, b_2, \ldots\}$ with $\perp \sqsubseteq a$, $\perp \sqsubseteq b_1 \sqsubseteq b_2 \sqsubseteq \ldots$, the image of all functions associated to constructor symbols to be $\langle a \rangle$, and the corresponding one of defined function symbols to be $D^{\mathcal{A}}$. This way $\mathcal{A}$ is clearly a

CRWL-algebra, satisfies all conditional rewrite rules, and leaving $\perp$ and a fixed, and mapping all b_i to some fixed b_j we get an infinite family of automorphisms of $\mathcal{A}$. On the other hand, in RL, if $\mathcal{R}$ is the RL-theory given by $(\{c\}, \{x = c\}, \emptyset, \emptyset)$ then, for all $\mathcal{R}$-systems $\mathcal{S}$, the equality $id_{\mathcal{S}} = c_{\mathcal{S}}$, where $c_{\mathcal{S}}$ is a constant functor, forces $\mathcal{S}$ to be a category with just one object and one arrow, and no infinite family of homomorphisms can exist. Therefore (as an equivalence of categories is full and faithful), $\mathbf{Mod}_{\mathrm{RL}}(\mathcal{R})$ is not categorically equivalent to $\mathbf{Mod}_{\mathrm{CRWL}}(\Phi(\mathcal{R}))$, whatever Φ might be.

Proposition 7. *$\mathcal{I}_{\mathrm{RL}}$ is not embeddable in $\mathcal{I}_{\mathrm{CRWL}}$.*

Admittedly, although this counterexample formally solves the problem, it is so particular that it cannot be considered to truly reflect the real differences between RL and CRWL. Future work should concentrate on finding a more illustrative example.

10 Conclusions

The main outcome of the research carried out in this paper has been the clarification of the relationship between RL and CRWL. Both logics have been proved to be expressive enough to simulate deduction in each other in a simple way, though resorting to binary predicates. On the other hand, the results on institutions have shown that neither can RL be considered as a sublogic of CRWL, nor can CRWL with respect to RL.

During the preparation of this work we have been forced to take a close look at the notions of entailment system and institution, and the difficulties we have found have shown us that intuition can be misleading in this field. The conclusion we have reached is that it would be very convenient to develop some kind of generalization of these concepts. One reason supporting this claim is the fact that, although it seems clear that CRWL should fit within the frame of entailment systems, the lack of the transitivity property forbids it to be considered so. In addition, there have been several occasions wherein we have had to make a distinction between two types of sentences within the same logic. The most outstanding case was that of labelled and unlabelled rewrite rules in RL, but we should also emphasize that rules in CRWL-programs are a restricted class of the more general class of reduction statements, and that, when we talked about reflection, we had to "weaken" its definition in order to encompass some of the results about RL, due to conditional sentences not being treated like unconditional ones. What all these examples have in common is that sentences belonging to a theory are given a different treatment from the rest of sentences and, with the current definitions of entailment system and institution, there is no way of taking this distinction into account.

For future work, besides the generalizations just mentioned, it would be interesting to complete the definition of the institution $\mathcal{I}_{\mathrm{RL}}$ with signature morphisms (morphisms of RL-theories), as well as finding a more illustrative example showing the reasons why RL is not embeddable in CRWL.

Acknowledgments

The author is indebted to Narciso Martí-Oliet, José Meseguer, Mario Rodríguez-Artalejo and the anonymous referees, for their valuable comments.

References

1. P. Arenas-Sánchez and M. Rodríguez-Artalejo. A general framework for lazy functional logic programming with algebraic polymorphic types. *Theory and Practice of Logic Programming*, 1(2):185–245, Mar. 2001.
2. M. V. Cengarle. The rewriting logic institution. Technical Report 9801, Ludwig-Maximilians-Universität München, Institut für Informatik, May 1998.
3. M. Clavel. *Reflection in General Logics and in Rewriting Logic, with Applications to the Maude Language*. PhD thesis, Universidad de Navarra, Spain, Feb. 1998.
4. , M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic, *Theoretical Computer Science*, 2001. To appear.
5. M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Lecture Notes. CSLI Publications, 2000.
6. M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 2001. To appear.
7. R. Diaconescu and K. Futatsugi. Logical foundations of CafeOBJ. *Theoretical Computer Science*, 2001. To appear.
8. F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, Spain, Jun. 1999.
9. J. C. González-Moreno, M. T. Hortalá-González, F. J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.
10. J. Meseguer. General logics. In H.-D. Ebbinghaus, J. Fernández-Prida, M. Garrido, D. Lascar, and M. Rodríguez-Artalejo, editors, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.
11. J. Meseguer. Rewriting as a unified model of concurrency. Technical Report SRI-CSL-90-02, SRI International, Computer Science Laboratory, Feb. 1990. Revised June 1990.
12. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
13. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1998.
14. J. M. Molina-Bravo. *Modularidad en Programación Lógico-Funcional de Primer Orden*. PhD thesis, Universidad de Málaga, Spain, Dec. 2000.
15. M. Palomino Tarjuelo. Relating Meseguer's rewriting logic with the constructor-based rewriting logic. Master's thesis, Facultad de Matemáticas, Universidad Complutense de Madrid, May 2001.