Verification of CRWL programs with rewriting logic

José Miguel Cleva
(Universidad Complutense de Madrid, Spain
jcleva@sip.ucm.es)

Isabel Pita
(Universidad Complutense de Madrid, Spain
ipandreu@sip.ucm.es)

Abstract: We present a novel approach to the verification of functional-logic programs. For our verification purposes equational reasoning is not valid due to the presence of non-deterministic and partial functions. Our approach transforms FLP programs into Maude theories and then uses the Rewriting Logic logical framework to verify properties of the transformed programs. We propose an inductive proving method based on the length of the computation on the Rewriting Logic framework to cope with the non-deterministic and non-terminating aspects of the programs. We illustrate the application of the method on various examples, where we analyze the sequence of steps to be performed by the proof in order to get expertise for the automatization of the process. Then, since the proposed transformation process is also amenable of automatization, we will obtain a tool for proving properties of CRWL programs. Another advantage of our methodology, that distinguish it from other approaches, is that it does not confuse the original functional-logic program with the subjects we want to talk about in the properties, but it allows the equational definition of observations on top of the transformed programs which simplifies the obtained proofs.
Key Words: Functional logic programming, Rewriting logic, Maude, Verification
Category: F.3.1, D.1.1, D.1.6

1 Introduction

Functional-logic (FL) languages like Toy [López and Sánchez 1999] or Curry [Hanus 2003] combine the use of non-strict and non-deterministic functions, which are quite useful for practical declarative programming. However, for verification purposes such non-deterministic and non-terminating functions make equational reasoning not valid for this purpose.

For this reason the *CRWL* logic has been proposed [González et al. 1996, González et al. 1999] as a suitable framework for FL programs. In this calculus we can derive reducibility relations $e \to t$ between evaluable expressions and constructed terms. There exist some extensions of this calculus to cope with HO, objects, failure, etc. [Rodríguez 2001], but in this paper we restrict our proposal to first-order functional-logic programming.

CRWL as a logical framework for rewrite systems shows many similarities with the rewriting logic (RL) proposed by Meseguer [Bruni and Meseguer 2003,

Meseguer 1992]. Rewriting logic has been implemented in the specification and programming language Maude [Clavel et al. 1999]. There are also some verification tools developed for Maude programs, like the ITP prover [Clavel 2001], the LTL model checker [Clavel et al. 1999] or the VLRL logic [Martí et al. 2005] for proving modal and temporal properties. In this paper we explore how to transform a CRWL program into an *equivalent* Maude program so that we can use the existing tools to prove properties of functional-logic programs.

The goal of our paper is to improve the verification methods for FL programs proposed in other works. In [Cleva et al. 2004], verification of FL programs is done by transforming them into logic programs in different ways. But in many cases the method is not as effective as one would desire. One of the problems arising there is that we have to translate the original program first into a logic program and then we need a translation between such logic program and the language accepted by the proving tool. In this paper we propose a transformation between CRWL programs and Maude programs. This transformation process is straightforward and it can be automated easily as we take advantage of the properties of Maude as a semantic framework for the specification; in this sense there are many works where Maude is used as the semantic framework to represent other logics [Martí and Meseguer 1999]. We also propose a proof mechanism that seems to be simpler in many interesting cases and that can be easily automated. We propose an inductive method for proving properties based on the length of the computation. This method complements the ITP prover and can be used in coordination with it. This coordination is done adapting the observations used in [Martí et al. 2005] for our purposes. Therefore, in order to prove any property we distinguish between the CRWL program itself and the observations we could define over it. Such observations correspond to the equational part of the resulting Maude specification and they distinguish between the program over which we can prove a given property and the property itself.

The rest of the paper is structured as follows. In the next section we give a brief introduction to rewriting logic. In Section 3 we present the CRWL calculus and we give the transformation process between CRWL and Maude programs. In section 4, we define the initial model of our programs and we give the language in which properties are expressed. Section 5 introduces the inductive method to prove the properties and illustrates its use on various examples. Finally, Section 6 analyzes this proof method and presents some conclusions.

2 Rewriting Logic and Maude

First, we outline some basic notions of rewriting logic and its implementation in the Maude language. For more information on the subject see [Clavel et al. 1999, Meseguer 1992, Meseguer 1993].

A *rewrite theory* $\mathcal{R}$ is defined as a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$ where (Σ, E) is an equational theory, L is a set of labels, and R is a set of possibly conditional labeled *rewrite rules*, $t \rightarrow t'$ that are applied modulo the equations E. Intuitively, the signature (Σ, E) of a rewrite theory describes a particular structure for the states of a system, and the rewrite rules describe which elementary local transitions are possible in the distributed state by concurrent local transformations.

The version of rewriting logic used in this paper is the one that selects membership equational logic (MEL), a generalization of *order-sorted* equational logic, as the underlying equational logic [Bruni and Meseguer 2003]. MEL supports sorts (e.g., `Bool`, `Nat`) via *many-sorted* signatures and the relation of sorts via *order-sorted* signatures (e.g. `NzNat < Nat`).

A MEL signature is a triple (K, Σ, S), where K represents a set of *Kinds*, $\Sigma = \{\Sigma_{\delta,k}\}_{(\delta,k)\in K^* \times K}$ a many-kinded signature and $S = \{S_k\}_{k\in K}$ a K-kinded family of disjoint sets of sorts. A MEL Σ-algebra A contains a set A_k for each kind $k \in K$, a function $A_f : A_{k_1} \times \ldots \times A_{k_n} \rightarrow A_k$ for each operator $f \in \Sigma_{k_1 \ldots k_n, k}$ and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$, with the meaning that elements in sorts are well-defined, while elements without sort are *errors*. We write $T_{\Sigma,k}$ and $T_\Sigma(X)_k$ to denote respectively the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in X, where $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ is a set of kinded variables.

Given a MEL signature, we may define equations of the form $t = t'$ or memberships of the form $t : s$, with $t, t' \in T_\Sigma(X)_k$ and $s \in S_k$. Order sorted notation $s_1 < s_2$ can be used to abbreviate the conditional membership $(\forall x : k)\ x : s_2$ *if* $x : s_1$.

The deduction rules for a RL theory are shown in Table 1. The rules are given for the unconditional case, which is the one used in the paper. The extended rules for the conditional case can be found in [Bruni and Meseguer 2003].

Systems in Maude are built out of basic elements called modules. A *functional module* specifies a theory (Σ, E) in membership equational logic. Equations of functional modules are assumed to be Church Rosser and terminating; however, equational attributes, like `assoc` and `comm` are allowed for declaring certain kinds of equational axioms, that would be in other case non-terminating equations. The initial model of a functional module is the initial algebra of the theory $T_{\Sigma/E}$. The specific syntax of a Maude functional module is

`fmod` *module-name* `is` *Declarations and statements* `endfm`.

where declarations include the importation of functional modules, sort, subsort and operator declarations, while statements include equational and membership axioms.

A *system module* specifies a rewrite theory. The initial model of a system module $\mathcal{T}_R$ is in essence an algebraic transition system, with additional operations, including a sequential composition operation for labelled transitions. The

Table 1: Rules of deduction for an RL-theory

Reflexivity	$\dfrac{t \in T_\Sigma(X)_k}{[t] \to [t]}$
Congruence	$\dfrac{f \in \Sigma_{k_1,\ldots,k_n,k}, \quad t_i, t_i' \in T_\Sigma(X)_k \text{ for } i \in [1 \ldots n] \quad [t_1] \to [t_1'] \ldots [t_n] \to [t_n']}{[f(t_1, ..., t_n)] \to [f(t_1', ..., t_n')]}$
Replacement	$\dfrac{r : [t(\overline{x})] \to [t'(\overline{x})] \quad [w_1] \to [w_1'] \ldots [w_n] \to [w_n']}{[t(\overline{w}/\overline{x})] \to [t'(\overline{w'}/\overline{x})]}$
Transitivity	$\dfrac{[t] \to [t'] \; [t'] \to [t'']}{[t] \to [t'']}$

data in the states of the system are provided by the underlying initial algebra $T_{\Sigma/E}$. The labelled state transitions are the concurrent rewrites possible in the system by application of the rules R. System modules are written in Maude by

`mod` *module-name* `is` *Declarations and statements* `endm`.

where the declarations and statements include now importation of system modules and rule statements.

3 Expressing CRWL programs in Maude

3.1 The Proof Calculus for CRWL

In this section we give a brief survey on the CRWL calculus [González et al. 1996, González et al. 1999] including the main characteristics needed in this paper.

We assume a signature $\Sigma = DC_\Sigma \cup FS_\Sigma$ where $DC_\Sigma = \bigcup_{n \in \mathbb{N}} DC_\Sigma^n$ is a set of *constructor* symbols and $FS_\Sigma = \bigcup_{n \in \mathbb{N}} FS_\Sigma^n$ is a set of *function* symbols, all of them with associated arity and such that $DC_\Sigma \cap FS_\Sigma = \emptyset$. We also assume a countable set $\mathcal{V}$ of *variable* symbols. We write Exp_Σ for the set of (total) *expressions* built up with Σ and $\mathcal{V}$ in the usual way, and we distinguish the subset $CTerm_\Sigma$ of (total) constructor terms or (total) *c-terms*, which only make use of DC_Σ symbols and variables from $\mathcal{V}$. The subindex Σ will usually be omitted. C-terms represent not further reducible data values.

Table 2: Rules for *CRWL*-provability

(BT) Bottom	$\dfrac{}{e \to \bot}$	if $e = f(e_1, \ldots, e_n)$
(MN) Monotonicity	$\dfrac{e_1 \to e'_1, ..., e_n \to e'_n}{h(e_1, ..., e_n) \to h(e'_1, ..., e'_n)}$	$h \in CS^n \cup FS^n$
(RF) Reflexivity	$\dfrac{}{e \to e}$	if e is ground
(R) Function reduction	$\dfrac{}{l \to r}$	if $l \to r \in [P]_\bot$
(TR) Transitivity	$\dfrac{e \to e' \; e' \to e''}{e \to e''}$	

We extend the signature Σ with a new constant $\bot$ representing the undefined value. The extended signature is denoted by $\Sigma_\bot$. $Exp_\bot$ and $CTerm_\bot$ denote respectively the sets of (partial) expressions and the set of (partial) c-terms. Partial c-terms represent the result of partially evaluated expressions; thus, they can be seen as approximations to the value of expressions.

As usual notation we will write $X, Y, Z, ...$ for variables, c, d for constructor symbols, f, g for functions, e for expressions and s, t for c-terms. In all cases, primes (') and subindices can be used. We will use the sets of substitutions $CSubst = \{\theta : \mathcal{V} \to CTerm\}$ and $CSubst_\bot = \{\theta : \mathcal{V} \to CTerm_\bot\}$. We denote by $e\theta$ the result of applying θ to e.

In this paper a *CRWL-program* $\mathcal{P}$ is a finite set of rewrite rules of the form $f(t_1, ..., t_n) \to e$ where $f \in FS^n$, $(t_1, ..., t_n)$ is a linear tuple (each variable in it occurs only once) of c-terms, and e is an expression. Notice that e may contain variables not occurring in $f(t_1, ..., t_n)$.

From a given program $\mathcal{P}$, the proof calculus considered in this work for *CRWL* is a simplification of the Basic Rewrite Calculus [González et al. 1999] and can derive *reduction statements* of the form $e \to e'$, with $e, e' \in Exp_\bot$. The intended meaning of such statement is that e can be reduced to e'. For our purposes of verifying properties of *CRWL* programs we are only interested on reductions from expressions to possibly partial c-terms ($e \to t$).

When a function rule R is applied to derive statements, the calculus uses the so called *c-instances* of R, defined as $[R]_\bot = \{R\theta \mid \theta \in CSubst_\bot\}$. We write $[P]_\bot$ to denote the set of c-instances of all the rules of a program P. Parameter passing in function calls are expressed by means of these c-instances in the proof calculus.

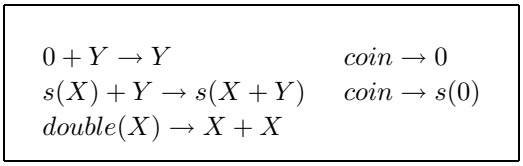

Figure 1: CRWL sample program *Coin*

Table 2 shows the proof calculus for *CRWL*. We write $\mathcal{P} \vdash_{CRWL} \varphi$ for expressing that the statement φ is provable from the program $\mathcal{P}$ with respect to this calculus. This calculus reflects a non-strict semantics, allowing non-terminating programs to be meaningful. The conditions imposed in the (**BT**) and (**R**) rules and the absence of joinability statements (which are present in the original CRWL calculus) is justified in [Cleva et al. 2004]; where a different proof calculus for *CRWL* is used, which is proved to be equivalent in the final reduction process to the calculus considered here [González et al. 1999].

A distinguished feature of *CRWL* (shared by concrete systems like Curry or Toy) is that programs can be non-confluent, defining thus *non-deterministic functions*. As a typical example, consider the program (called *Coin* for future references) in Fig.1, which assumes the constructors 0 and s for natural numbers. Notice that *coin* is a non-deterministic function, for which the previous calculus can derive the statements $coin \to 0$ and $coin \to s(0)$. The use of c-instances in rule (R) instead of general instances corresponds to *call time choice* semantics for non-determinism [Hussmann 1992, González et al. 1999]). In the example, it is possible to build a *CRWL*-proof for $double(coin) \to 0$ and also for $double(coin) \to s(s(0))$, but not for $double(coin) \to s(0)$. Call-time choice is related to *sharing*, a well known operational technique considered essential for the effective implementation of lazy functional languages like Haskell. Existing FLP languages like Curry or Toy also use sharing and call-time choice semantics. The above described behaviour for the reduction of $double(coin)$ corresponds exactly with what happens in those systems. Run-time choice, an alternative semantics for non-determinism with which $double(coin)$ can be reduced also to $s(0)$ is investigated for the FLP setting in [Antoy 1997].

From the point of view of verifying properties of FLP programs, nondeterminism and call-time choice semantics have the unpleasant consequence that equational reasoning is not valid for CRWL-programs. In the previous example, if the rules for *coin* were understood as the equalities $coin = 0$ and $coin = s(0)$, then we could deduce $0 = s(0)$, which is not intended. Call-time choice implies that not only equational reasoning, but also ordinary rewriting is invalid since, from the point of view of rewriting, the rule $double(X) \to X + X$ should be appli-

cable to *any* X, and not only to c-terms. Hence, we would have $double(coin) \rightarrow coin + coin$, and from this, $double(coin) \rightarrow s(0)$, which is not valid with call-time choice.

3.2 CRWL programs in Maude

Rewriting logic and CRWL exhibit clear similarities in their proof calculi, that allow us to simulate one logic with the other [Palomino 2003]. Nevertheless, using directly the rules of RL we cannot obtain the call-time choice semantics for nondeterminism, since an important difference between both calculus is that in the CRWL calculus the function reduction rule (**R**) is applied only to c-instances of program rules while the RL Replacement rule is applied to any RL term. In [Palomino 2003] this problem is solved with the use of explicit functions that distinguish c-terms from general expressions. We adopt in this paper a different approach by taking membership equational logic as the underlying logic of RL.

Consider the following transformation method:

1. Each CRWL program defines an RL theory which is written as a Maude system module.
2. We define three sorts, `Expr`, `Term` and `TTerm`, that represent the CRWL expressions ($Exp_\perp$), possibly partial c-terms ($CTerm_\perp$), and total c-terms ($CTerm$) respectively, with the subsort relation `TTerm` $<$ `Term` $<$ `Expr`. The sort `FApp` captures the CRWL expressions of the form $f(e_1, \ldots, e_n)$ where $e_i \in Exp_\perp$ and $f \in FS^n$ and it is declared as a subsort of `Expr`. The sort is used to avoid the application of the CRWL **BT** rule on c-terms.
3. We represent the undefined value $\perp$ of CRWL as a constant `bottom` of sort `Term`. The **BT** rule of CRWL defines a rule `rl [l] : x:FApp => bottom`.
4. Every $c \in DC^0$ is declared as a constant of sort `TTerm`, `op c : -> TTerm .`
5. Every $c \in DC^n$ with $n > 0$ defines an operation:
   `op c : Expr ... Expr -> Expr .`
6. For every $c \in DC^n$ with $n > 0$ we define the following membership relations:
   `c(X1:TTerm,...,Xn:TTerm):TTerm`
   `c(X1:Term,...,Xn:Term):Term`
7. Every $f \in FS^n$ with $n \geq 0$ defines an operation:
   `op f : Expr ... Expr -> FApp .`
8. Every CRWL program rule $l(\bar{X}) \rightarrow r(\bar{X})$, where $\bar{X}$ represents all the variables occurring both in l and r, defines a rewrite rule: $l(\bar{X}) \Rightarrow r(\bar{X})$ where for all i, X_i is of sort `Term`.

For example, the CRWL program of Figure 1 is transformed into the following Maude system module:

```
mod COIN is
  sorts TTerm Term Expr .
  subsort TTerm < Term < Expr .
  sort FApp .
  subsort FApp < Expr .
  op 0 : -> TTerm .
  op s : Expr -> Expr .
  op coin : -> FApp .
  op _+_ : Expr Expr -> FApp .
  op double : Expr -> FApp .
  op bottom : Term .

  mb s(X:TTerm) : TTerm .
  mb s(X:Term) : Term .
  rl [bot] : X:FApp => bottom .
  rl [coin0] : coin => 0 .
  rl [coin1] : coin => s(0) .
  rl [sum0] : 0 + X:Term => X:Term .
  rl [sum1] : s(X:Term) + Y:Term => s(X:Term + Y:Term) .
  rl [dob] : double(X:Term) => X:Term + X:Term .
endm
```

The following lemma shows the relation between the CRWL expressions and the sort of the associate Maude terms of the transformed program

Lemma 1. *Let t, e be ground CRWL expressions and the related*[1] *Maude terms t, e in the transformation process. We have the following equivalences:*

1. $t \in \mathit{CTerm}_\perp \Leftrightarrow t : \texttt{Term}$
2. $t \in \mathit{CTerm} \Leftrightarrow t : \texttt{TTerm}$
3. $e \in \mathit{Exp}_\perp \Leftrightarrow t : \texttt{Expr}$
4. $e = f(e_1, \ldots, e_n) \Leftrightarrow e : \texttt{FApp}$ *for any* $f \in \mathit{FS}^n$ *and* $e_i \in \mathit{Exp}_\perp$.

Proof. 1. ($\Rightarrow$) We proceed by induction over $t \in \mathit{CTerm}_\perp$. If $t = \perp$ by the transformation rule (3) we obtain $t : \texttt{Term}$. If $t = c$ for some $c \in \mathit{CS}^0$, by (4)

[1] We will use the same name for CRWL terms and their transformed Maude terms when there is no ambiguity since syntactically both terms are identical except for the $\perp$

and the subsort declaration (2) `TTerm < Term` we obtain t : `Term`. For the inductive case we consider $c \in DC^n$ and $t_i : Cterm_\perp$ then $t = c(t_1, \ldots, t_n)$. By induction hypothesis $t_i : CTerm_\perp \Rightarrow t_i$: `Term` and applying step (6) of the transformation process we obtain t : `Term`.

($\Leftarrow$) We proceed by induction over $t \in$ `Term`. We distinguish three possible cases. If $t =$ `bottom` : `Term`, it should have been obtained through step (3) so it represents the undefined value $\perp$ of CRWL and $\perp \in CTerm_\perp$. If $t = c(e_1, \ldots, e_n)$: `Term` and $c \in DC^n$ it should have been obtained through steps (4) or (5) depending whether $n = 0$ or $n > 0$, for the first case we obtain $c \in CTerm_\perp$. For the second case, if $n > 0$ t : `Term` can only be obtained by the membership of step (6), therefore for every i, $e_i \in$ `Term` and by induction hypothesis $e_i \in Cterm_\perp$. Now applying the definition of $CTerm_\perp$ we obtain $t \in CTerm_\perp$. The final case, $t = f(e_1, \ldots, e_n)$ where $f \in FS^n$, is impossible as there are no membership axioms to restrict the sort to such expression.

2-4. The rest of the cases can be proved in the same way. □

The main result of this section relates the reductions obtained using the CRWL calculus and the reductions for the RL transformed program. The equivalence of both programs is based on the restriction of the application of the RL rules to partial c-terms in the transformation method.

Proposition 2. *Let P be a CRWL program and $\hat{P}$ the Maude transformed program. For any ground expressions e and e' we have:*

$$P \vdash_{CRWL} e \to e' \Leftrightarrow \hat{P} \vdash_{RL} e \to e'$$

Proof. [$\Rightarrow$] Assume $P \vdash_{CRWL} e \to e'$. We proceed by induction on the rules of the CRWL calculus.

1. Suppose that $e \to e'$ is obtained by applying the **BT** rule of CRWL, then e is of the form $f(e_1, \ldots, e_n)$. Therefore $f \in FS^n$ with $n > 0$ and by the transformation step (7) we obtain an RL term $f(e_1, \ldots, e_n)$ of sort `FApp`. $\hat{P} \vdash_{RL} e \to e'$ is obtained by applying the RL Replacement rule with the rewrite rule `bot` defined in step (3) of the transformation method.

2. If $e \to e'$ is obtained by applying the **MN** rule of CRWL. Then $e = h(e_1, \ldots e_n)$ where $h \in DC^r \cup FS^n$, and by induction hypothesis, $\hat{P} \vdash_{RL} e_i \to {e'}_i$ for $i = 1 \ldots n$. Therefore, $\hat{P} \vdash_{RL} e \to e'$ is derived with the RL Congruence rule.

3. Suppose that $e \to e'$ is obtained by applying the **RF** rule of CRWL. Then e' is syntactically equal to e and $\hat{P} \vdash_{RL} e \to e'$ is derived with the RL Reflexivity rule.

4. Suppose that $e \to e'$ is obtained by applying the **TR** rule of CRWL. Then, there is an expression e'' such that $P \vdash_{CRWL} e \to e''$ and $P \vdash_{CRWL} e'' \to e'$. By induction hypothesis $\hat{P} \vdash_{RL} e \to e''$ and $\hat{P} \vdash_{RL} e'' \to e'$. Therefore, $\hat{P} \vdash_{RL} e \to e'$ is derived with the RL Transitivity rule.

5. If $e \to e'$ is obtained by applying the **R** rule of CRWL, then $e \to e'$ is a c-instance of a CRWL program rule $l \to r$. Let $\theta = \bar{w}/\bar{x}$ be the substitution used to obtain the c-instance of the rule. By lemma 1 each CRWL c-term w_i is transformed into an RL term w_i of sort `TTerm`, we can apply the RL replacement rule to the rewrite rule obtained from $l \to r$ by the transformation step (8) with the substitution $\bar{w}/\bar{x}$. Notice that by the RL Reflexivity rule we always have $[w_i] \to [w_i]$.

[$\Leftarrow$] Assume $\hat{P} \vdash_{RL} e \to e'$. We proceed by induction on the rules of the RL calculus.

1. If $e \to e'$ is obtained by applying the RL Reflexivity rule then as in the transformed program $\hat{P}$ there are no equational rules, e' is syntactically equal to e and $P \vdash_{CRWL} e \to e'$ is derived with the **RF** CRWL rule.

2. Suppose that $e \to e'$ is obtained by applying the RL Congruence rule. Then, $e = h(e_1, \ldots e_n)$ and by induction hypothesis $P \vdash_{CRWL} e_i \to e'_i$ for any $i \in 1 \ldots n$. Therefore, $P \vdash_{CRWL} e \to e'$ is derived with the **MN** CRWL rule.

3. Suppose that $e \to e'$ is obtained by applying the RL Transitivity rule. Then, there is an expression e'' such that $\hat{P} \vdash_{RL} e \to e''$ and $\hat{P} \vdash_{RL} e'' \to e'$ and by induction hypothesis $P \vdash_{CRWL} e \to e''$ and $P \vdash_{CRWL} e'' \to e'$. Therefore, $P \vdash_{CRWL} e \to e'$ is derived with the **TR** CRWL rule.

4. If $e \to e'$ is obtained by applying the RL Replacement rule of deduction we distinguish two cases:

   - If the RL Replacement rule is applied with the `bot` rewrite rule, we have $e \to \bot$ with e of sort `FApp`. Since the transformation method does only define operators of sort `FApp` from CRWL functions $f \in FS^n$ we have that e is of the form $f(e_1, \ldots, e_n)$. Then $P \vdash_{CRWL} e \to t$ can be derived using the **BT** CRWL deduction rule.

   - Suppose the RL Replacement rule is applied with a rule different from `bot`. This rewrite rule must have been obtained from the CRWL program by step (8) of the transformation process; then, variables can only have been instantiated by terms w_i of sort `Term`.

   Now, by lemma 1 terms of sort `Term` are always obtained from CRWL terms, and we can derive by induction hypothesis $P \vdash_{CRWL} w_i \to w'_i$, for $i \in 1 \ldots n$

Then, we apply the **R** CRWL deduction rule and derive $P \vdash_{CRWL} t(\bar{w}/\bar{x}) \rightarrow t'(\bar{w}/\bar{x})$.

Finally, by the repeated application of the **MN** CRWL deduction rule, we can derive $P \vdash_{CRWL} t(\bar{w}/\bar{x}) \rightarrow t'(\bar{w'}/\bar{x})$.

□

4 Expressing abstract properties of CRWL programs

In this section we give the logical framework in which properties of the functional-logic system will be specified and proved.

4.1 Formal specification of properties

We are interested in expressing properties of all possible terms reachable from an initial expression. For example, we may want to prove a property about the *Coin* program, defined in [Section 3.2], stating that from *double(T)* we can only obtain an even number.

First, we need to define the subjects we will talk about in the properties, like, for example, that a natural number is *even*. We use operations, called *observations*, over the state of the system, that are declared in an extension of the system module, as is done in [Martí et al. 2005]. In this way the user can decide the means by which he will observe the system without interfering with the system specification. Since we are interested in expressing properties of transformed CRWL terms we take observations as operations of type `Term` $\rightarrow s$ or `TTerm` $\rightarrow s$ if the CRWL term we want to observe is a total term. Then we provide their algebraic specification. The definition of the observations requires an extension of the signature of the system in order to have available the observations and perhaps some sorts and auxiliary operations that are related to the observations but are not part of the original program. We denote by (Σ^+, E^+) the extended signature.

For example, the previous observation about even numbers will be an operation of type $even$: `TTerm` $\rightarrow$ $Bool$. The extended signature includes the observation, but in this case we don't need to extend the RL signature with new sorts, since the CRWL original program does already define the *Boolean values*.

The logic we will use to express our properties is first order equational logic with predicates that express sequences of rewrites.

$$\varphi ::= true \mid t_1 = t_2 \mid s_1 \rightarrow^i s_2 \mid s_1 \rightarrow s_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \neg\varphi \mid \forall x : s.\varphi$$

where $t_1, t_2 \in T_{\Sigma^+}(X)$, are two terms of the same sort over the extended signature, $s_1, s_2 \in T_{\Sigma}(X)$ are two terms of the same sort over the original signature, $\_ \rightarrow^i \_$ represents a collection of binary operators, one for each possible $i \in \mathbb{N}$,

and s is the sort `Term` or the sort `TTerm`. For example the previous property about the *Coin* program would be expressed as

$$\forall T, T' : \texttt{TTerm}.(double(T) \rightarrow T' \Rightarrow even(T'))$$

We will make use of the usual derived propositional connectives for conjunction ($\wedge$), disjunction ($\vee$), and logical equivalence ($\equiv$). The existential quantifier is also defined in terms of the universal quantifier as usual. Free variables are considered universally quantified, that is, our properties are expressed as closed formulae.

4.2 Model semantics

The models we use to interpret our formulae are the transition systems defined by the Maude extended rewrite theories obtained from the CRWL programs.

- The set of states is the set $T_{\Sigma,Expr}$ of ground terms of sort `Expr`.
- The family of state transitions is given by the *one-step sequential rewrites* of the rewrite theory. Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, a (Σ, E)-sequent $[t] \rightarrow [t']$ is called a *one-step sequential rewrite* [Meseguer 1993] iff it can be derived from $\mathcal{R}$ by finite application of the rules 1–3 of RL [see Section 2], with exactly one application of rule 3.

We have to provide an interpretation I which is a family of $I_{Obs}: T_{\Sigma,E,Term} \rightarrow T_{\Sigma^+,E^+,s}$ for the observations of the system. Each interpretation of I is defined axiomatically through sets of equations over the extended signature Σ^+.

The satisfaction relation is now defined for a given transition system, a given observation interpretation and a given initial state.

The term language is interpreted in T_{Σ^+,E^+} as follows:

- $[\![obs(t)]\!]^I = I_{Obs}(t)$, where t is of sort `Term`,
- $[\![f(t_1, \ldots, t_m)]\!]^I = f([\![t_1]\!]^I, \ldots, [\![t_m]\!]^I)$,
  for each operation $f : s_1 \ldots s_m \rightarrow s$ in Σ^+.

Satisfaction of formulae for a given transition system $\mathcal{K}$, observation interpretation I and initial state e_0 is

- $\mathcal{K}, I, e_0 \models true$,
- $\mathcal{K}, I, e_0 \models t_1 = t_2$ iff $[\![t_1]\!]^I = [\![t_2]\!]^I$,
- $\mathcal{K}, I, e_0 \models s_0 \rightarrow^1 s_1$ iff $[s_0] = [e_0]$ and $[s_0] \rightarrow [s_1] \in \mathcal{K}$,
- $\mathcal{K}, I, e_0 \models s_0 \rightarrow^n s_n$ iff there are $s_1, \ldots s_{n-1} \in T_\Sigma$ and $\mathcal{K}, I, e_0 \models s_0 \rightarrow^1 s_1$ and $\mathcal{K}, I, s_i \models s_i \rightarrow^1 s_{i+i}$ for $i = 1 \ldots n-1$,

- $\mathcal{K}, I, e_0 \models s_0 \rightarrow s_n$ iff there is some $i \in \mathbb{N}^+$ such that $\mathcal{K}, I, e_0 \models s_0 \rightarrow^i s_n$
- $\mathcal{K}, I, e_0 \models \neg\varphi$ iff it is not the case that $\mathcal{K}, I, e_0 \models \varphi$,
- $\mathcal{K}, I, e_0 \models \varphi_1 \Rightarrow \varphi_2$ iff $\mathcal{K}, I, e_0 \models \varphi_1$ implies $\mathcal{K}, I, e_0 \models \varphi_2$,
- $\mathcal{K}, I, e_0 \models \forall x : s.\varphi$ iff for all substitutions $x \leftarrow t'$ we have $\mathcal{K}, I, e_0 \models \varphi[x \leftarrow t']$

where $t_1, t_2 \in T_{\Sigma^+,E^+}$ and $s_0, \ldots, s_n \in T_\Sigma$.

Notice that $\mathcal{K}, I, s_0 \models s_0 \rightarrow^i s_i \rightarrow^j s_n$ iff $\mathcal{K}, I, s_0 \models s_0 \rightarrow^i s_i$ and $\mathcal{K}, I, s_i \models s_i \rightarrow^j s_n$

A formula is valid in the initial model of the extended signature if the initial model satisfies the formula for any initial state.

4.3 Examples of abstract properties

The framework presented in this section allows us to specify and proof properties about semantic features of a program. For that purpose we consider the approximation relation of partial c-terms presented in CRWL [González et al. 1999]. The approximation relation is defined as the least partial ordering over $\text{CTerm}_\perp$ satisfying: $\perp \sqsubseteq t$ for every $t \in \text{CTerm}_\perp$, and for every $c \in DC$ and $t_i, t'_i \in \text{CTerm}_\perp$, $c(t_1, \ldots, t_n) \sqsubseteq c(t'_1, \ldots, t'_n)$ if $t_i \sqsubseteq t'_i$. This definition can be specified as an observation of the system and thus equationally. Therefore, some properties of the original program can be stated:

- We can specify that a function is deterministic by the formula:

  $$\forall X\ T\ T' : Term. f(X) \rightarrow T \wedge f(X) \rightarrow T' \Rightarrow \exists\ T'' : Term\ .\ T \sqsubseteq T'' \wedge T' \sqsubseteq T''$$

- A function is strict if it satisfies the formula:

  $$\forall\ T : Term\ .\ f(\perp) \rightarrow T \Rightarrow T = \perp$$

- And finally we can show that a function is totally defined proving the formula:

  $$\forall X : TTerm\ .\ \forall T : Term\ .\ f(X) \rightarrow T \Rightarrow \exists T' : TTerm. T \sqsubseteq T' \wedge f(X) \rightarrow T'$$

5 Inductive methods for proving properties

We can distinguish between two types of properties, those that are written using only the first-order equational part of the logic and those that are written using sentences of rewriting logic. The first type of properties may be proved with Maude's Inductive Theorem Prover (ITP) [Clavel 2001], that proves theorems

over inductive models of functional Maude modules. The ITP implements an structural induction proving method based on the structure of the system.

We propose using induction on the length of the computation for proving the second type of properties. The inductive proof method is as follows: to prove that the property $(t \rightarrow t') \Rightarrow \varphi$ is valid in the extended initial model it is sufficient to show:

- $(t \rightarrow^1 t') \Rightarrow \varphi$ is valid for all possible one-step sequential rewrites.
- Assuming $(t \rightarrow^k t') \Rightarrow \varphi$ for $k \leq n$ and $n \geq 1$ is valid, it must be shown that $(t \rightarrow^{n+1} t') \Rightarrow \varphi$ is valid for all possible $n+1$-step sequential rewrites.

The correctness of the inductive method is obtained from the semantics of the $\rightarrow^i$ connectives.

We illustrate the application of these ideas on the following examples.

5.1 Example: Even numbers

A CRWL program to obtain any even number is the following:

$$aneven \rightarrow 0$$
$$aneven \rightarrow s(s(aneven))$$

We could be interested in proving a property stating that every possible total reduction of *aneven* produces an even number. First we obtain the Maude program following the transformation process of [Section 3.2]:

```
mod EVEN-NUMBERS is
  sorts Expr Term TTerm .
  subsort TTerm < Term < Expr .
  sort FApp .
  subsort FApp < Expr .

  op 0 : -> TTerm .
  op aneven : -> FApp .
  op s_ : Expr -> Expr .
  op bottom : -> Term .

  mb s(X:TTerm) : TTerm .
  mb s(X:Term) : Term .

  rl [bot] : X:FApp => bottom .
  rl [init] : aneven => 0 .
  rl [next] : aneven => s s aneven .
endm
```

We need to observe if a natural number is even to prove our property.

```
mod EVEN-NUMBERS-OBSERVED is
  protecting EVEN-NUMBERS .
  op even : TTerm -> Bool .

  eq even(0) = true .
  eq even(s 0) = false .
  eq even(s s T:TTerm) = even(T:TTerm) .
endm
```

The property is formalized as follows:

$$\forall T : \texttt{TTerm}.((\texttt{aneven} \rightarrow T) \Rightarrow (\texttt{even}(T) = \texttt{true}))$$

For proving the property we need the following lemma about the preservation of constructors on the rewrite process:

Lemma 3. *Given a Maude program $\hat{P}$ obtained from a CRWL program P, an expression $e \in$ `Expr` and a function symbol c obtained from a CRWL expression e and a CRWL constructor symbol $c \in DC$, we have:*

$$\forall n : I\!N.\ \forall e' : \texttt{Expr}.\ (c(e) \rightarrow^n e' \Rightarrow \exists e'' : \texttt{Expr}.(e \rightarrow^n e'' \wedge e' = c(e'')))$$

Proof. (Sketch of the proof) The proof is done by induction over the length of the computation. The result uses the fact that since the Maude program is obtained from a CRWL program, the rewrite process cannot affect constructors symbols because the program does not provide any rewrite rule for such symbols.

Now, we prove the property by induction on the length of the computation:

- Base case: $\forall T : \texttt{TTerm}.((\texttt{aneven} \rightarrow^1 T) \Rightarrow (\texttt{even}(T) = \texttt{true}))$.

  We can apply the rewrite rules of the program related with the function `aneven`. We have three rewrite rules in the system.

  1. If we apply the rule `bot` we obtain a state $T =$ `bottom` and then the property is true because we have reached an expression that is not of the required sort.
  2. If we apply rule `init` we obtain a state $T =$ `0` and it can be proved with the ITP tool that `even(0)` = `true`.
  3. If we apply rule `next` we obtain a state `s s aneven` $\notin$ `TTerm`.

- Inductive case ($n > 1$): As before, we have three possible rewrite rules in the system: `bot`, `init` and `next` which can constitute a one-step sequential rewrite. We will show what happens when these rules are applied to the initial state `aneven` and then the inductive hypothesis is applied for the rest of the computation.

1. If we apply the rule `bot` then we have `aneven` $\rightarrow^1$ `bottom` and then from `bottom` we cannot apply more rewrite rules, therefore there is not a $n+1$-sequential rewrite.

2. If we apply rule `init` then we obtain `aneven` $\rightarrow^1$ `0` and at this point there is no rewrite rule to be applied. As in the previous case there is not a $n+1$-sequential rewrite.

3. If we apply rule `next`, we have to prove:

   $\forall T : \texttt{TTerm}.((\texttt{aneven} \rightarrow^1 \texttt{s s aneven} \rightarrow^n T) \Rightarrow (\texttt{even}(T) = \texttt{true}))$.

   Therefore by lemma 3, $T = \texttt{s s } T'$ for some T' such that `aneven` $\rightarrow^n T'$. We apply the induction hypothesis and we obtain `even`(T') and therefore as $T =$ `s s` (T') applying the rules of the observation part we obtain `even`(T).

5.2 Example: Lists of naturals

Now, we translate the following CRWL specification of lists of naturals

$$\begin{array}{ll} insert(X, nil) & \rightarrow X : nil \\ insert(X, Y : Xs) & \rightarrow X : (Y : Xs) \\ insert(X, Y : Xs) & \rightarrow Y : insert(X, Xs) \\ permute(nil) & \rightarrow nil \\ permute(X : Xs) & \rightarrow insert(X, permute(Xs)) \end{array}$$

into a Maude module:

```
mod NAT-LIST is
  sorts TTerm Term Expr .
  subsort TTerm < Term < Expr .
  sort FApp .
  subsort FApp < Expr .

  op bottom : -> Term .

  op 0 : -> TTerm .
  op s : Expr -> Expr .

  op nil : -> TTerm .
  op _:_ : Expr Expr -> Expr .
  op permute : Expr -> FApp .
  op insert : Expr Expr -> FApp .

  mb (s (X:TTerm)) : TTerm .
  mb (s (X:Term)) : Term .

  mb ((X:TTerm) : (Xs:TTerm)) : TTerm .
  mb ((X:Term) : (Xs:Term)) : Term .
```

```
  rl [bot] : X:FApp => bottom .
  rl [insert1] : insert(X:Term, nil) => X:Term : nil .
  rl [insert2] : insert(X:Term, Y:Term : Xs:Term)
                 => X:Term : (Y:Term : Xs:Term) .
  rl [insert3] : insert(X:Term, Y:Term : Xs:Term)
                 => Y:Term : insert(X:Term, Xs:Term) .
  rl [permute1] : permute(nil) => nil .
  rl [permute2] : permute(X:Term : Xs:Term)
                  => insert(X:Term, permute(Xs:Term)) .
endm
```

We define an observation *member*, that says if a given natural number belongs to a list. Observe that the observation, like in the previous example, is only defined on the `TTerm` sort.

```
mod NAT-LIST-OBSERVED is
  protecting NAT-LIST .

  op member : TTerm TTerm -> Bool .

  vars X Y : TTerm .
  var Xs : TTerm .

  eq member(X, nil) = false .
  eq member(X, X : Xs) = true .
  ceq member(X, Y : Xs) = member(X, Xs) if X =/= Y .
endm
```

The property we want to prove is:

$$\forall X : TTerm\ \forall L\ L' : TTerm.$$
$$(permute(L) \rightarrow L' \Rightarrow (member(X, L) = true \Rightarrow member(X, L') = true))$$

For proving the property, we need the following auxiliary lemma:

Lemma 4. *For all $x, t : TTerm$ and $T, T' : TTerm$ we have:*

1. $insert(t, T) \rightarrow T' \Rightarrow member(t, T')$.
2. $insert(t, T) \rightarrow T' \Rightarrow (member(x, T) = true \Rightarrow member(x, T') = true)$.

Proof. (Proof sketch)

1. We proceed by induction on the length of the computation, distinguishing cases between the rewrite rules applied at the one step sequential rewriting. The obtained cases are proved straightforward by pattern matching with the program rules `bot`, `insert1`, `insert2` and `insert3` and equational reasoning for the observations.

2. We also proceed by induction on the length of the computation, and like the above case, the proof is obtained straightforward. □

Now, to prove the property, we also proceed by induction on the length of the computations.

1. Base case.

$$\forall X : Nat\ \forall L\ L' : Term.(permute(L) \rightarrow^1 L'(member(X, L) = true \Rightarrow member(X, L') = true))$$

The rewrite theory defines 5 rewrite rules, but only two of them can be applied in a state $permute(L)$. Notice that since L is of sort *Term*, it cannot be of the form `insert(x,Xs)` or `permute(Xs)`. The two possible rewrite rules are applied *at the top*, that is, the *congruence* rule of the logic is not necessary. *Reflexivity* can always be applied in a one-step sequential rewrite, but it does not affect the result of the execution, $permute(L) \rightarrow^1 L'$. The *replacement* rule of the logic gives raise to the following two cases:

(a) if we can apply the rule

```
rl [permute1] : permute(nil) => nil
```

to the state *permute(L)* is because $L =$ `nil`.

Then, since $member(x, nil) = false$ the property is fulfilled because its premise is false.

(b) if we can apply the rule

```
rl [permute2] : permute(X : Xs) => insert(x, permute(Xs))
```

we obtain the state `insert(X, permute(Xs))` which is not of sort *Term* as it is required by the left-hand side of the property.

2. For the inductive case, assume

$$\forall X : Nat\ , L\ L' : Term.(member(X, L) \wedge permute(L) \rightarrow^n L' \Rightarrow member(X, L'))$$

and prove

$$\forall X : Nat\ , L\ L' : Term.(member(X, L) \wedge permute(L) \rightarrow^{n+1} L' \Rightarrow member(X, L'))$$

We will apply the induction hypothesis in the last part of the computation, that is, we will explore the computations

$$permute(L) \rightarrow^1 T \rightarrow^n L'$$

Like in the base case, we can only apply the system rewrite rules in the top, hence *congruence* is not applied. Let's see what happens when *replacement* is applied to the two possible system rewrite rules.

(a) When we apply the system rewrite rule

```
rl [permute1] : permute(nil) => nil
```

to the state *permute(L)* we obtain the state *nil*. Since there is no possible one-step sequential rewrite defined for this state, we are in the base case.

(b) Lets now consider the second system rewrite rule

```
rl [permute2] : permute(X : Xs) => insert(X, permute(Xs)) .
```

When this rule is applied we obtain the state `insert(l, permute(L))`, where the initial list is of the form $l : L$. We can now apply the *replacement* rule of the logic only to the subterm: `permute(L)`. and we obtain an execution of the form

$$permute(l : L) \rightarrow^1 insert(l, permute(L)) \rightarrow^m insert(l, L') \rightarrow^{n-m} L''$$

with $m < n$, since the rules `insert1`, `insert2` and `insert3` can only be applied over terms of sort `Term`. Then, by induction hypothesis:

$$\forall X : Nat.(member(X, L) \Rightarrow member(X, L')).$$

Now, by Lemma 1(b) we have:

$$\forall X : Nat.(member(X, L') \Rightarrow member(X, L'')).$$

And by Lemma 1(a) we have $member(l, L'')$.

Since $member(x, l : L) \equiv x == l \vee member(x, L)$ we have the result.

5.3 Repeating elements

As a final example we consider a function building lists formed by the repetition of one element. The CRWL program is the following:

$$repX \rightarrow X : rep(X)$$

The semantics of call-time choice produces that the possible reductions from `rep(coin)` should be `0:0:0...` or `1:1:1...` and it is impossible to obtain any other reduction containing 0's and 1's at the same time. One property to be proved could be the following: Only cero is a member of the list obtained from `rep(0)`. To prove such property we proceed as in the former cases considering the translation of the CRWL program into a Maude module.

```
mod LIST-REP is
  sorts TTerm Term Expr .
  subsort TTerm < Term < Expr .
  sort FApp .
  subsort FApp < Expr .

  op bottom : -> Term .
  op 0 : -> TTerm .
  op s : Expr -> Expr .
  op nil : -> TTerm .
  op _:_ : Expr Expr -> Expr .
  op rep : Expr -> FApp .

  mb (s(X:TTerm)) : TTerm .
  mb (s(X:Term)) : Term .

  mb ((X:TTerm) : (Xs:TTerm)) : TTerm .
  mb ((X:Term) : (Xs:Term)) : Term .

  rl [bot] : X:FApp => bottom .
  rl [repeat] : rep(X:Term) => X:Term : rep(X:Term) .
endm
```

We define the *member* observation for the *TTerm* sort.

```
mod LIST-REP-OBSERVED is
  protecting LIST-REP .

  op member : TTerm Term -> Bool .

  vars X Y : TTerm .
  var Xs : Term .

  eq member(X, bottom) = false .
  eq member(X, nil) = false .
  eq member(X, X : Xs) = true .
  ceq member(X, Y : Xs) = member(X, Xs) if X =/= Y .
endm
```

We want to prove the following property:

$$\forall N : TTerm\ L : Term\ .\ (rep(0) \rightarrow L \Rightarrow (member(N,L) \Rightarrow (N == 0)))$$

We prove it by induction on the length of the computation

1. Base case:

   $$\forall N : TTerm\ L : Term\ .\ (rep(0) \rightarrow^1 L \Rightarrow (member(N,L) \Rightarrow (N == 0))).$$

   We have two rewrite rules in the system.

   (a) If we apply the `bot` rule we have $rep(0) \rightarrow^1$ `bottom` and therefore the property is trivially true.

(b) If we apply `repeat` we obtain the state `0 : rep(0)`, since it is not of sort *Term* the property is fulfilled.

2. Inductive case: For the inductive step we assume that $\forall N : TTerm\ L : Term\ .\ (rep(0) \rightarrow^n L \Rightarrow (member(N,L) \Rightarrow (N == 0)))$. and we have to prove that $\forall N : TTerm\ L : Term\ .\ (rep(0) \rightarrow^{n+1} L \Rightarrow member(N,L) \Rightarrow (N == 0))$.

   – We can apply the two program rules.

   • If the `bot` rule is applied, we obtain $rep(0) \rightarrow^1 bottom$ and we are in the base case.

   • If we apply the `repeat` rule, we obtain $rep(0) \rightarrow^1 0 : rep(0)$. From this expression we can only rewrite the *rep(0)* term and we get $rep(0) \rightarrow^n L'$. We are only interested in the states such that L' is of sort *Term*, since in other case the property is true. By induction hypothesis, we have $member(N, L') \Rightarrow (N == 0)$. Therefore, $rep(0) \rightarrow^1 0 : rep(0) \rightarrow^n 0 : L'$

   We need to prove $member(N, 0 : L') \Rightarrow (N == 0)$, but since $member(N, 0 : L') \equiv N == 0 \vee member(N, L')$ we can derive the result.

5.4 Representing sequential rewrites in first order equational logic

In some cases, the representation of sequential rewrites in first order equational logic is possible and it allows the use of existing theorem provers to proof the properties.

The process to obtain the representation is the following: first we define the sort `Comp` representing sequential rewrites as a list of the states of the system in the rewriting process. We use two operators to generate the computation:

```
op U: Exp -> Comp .
op C: Exp Comp -> Comp .
```

Then, we identify one-step sequential rewrites by means of a function `red` declared as:

```
op red: Exp Exp -> Bool .
```

that mirrors the rewrite rules of the system.

The function `cvalid` checks if a given computation is possible in the system. The declaration is:

```
op cvalid: Comp -> Bool .
```

and it is specified as:

```
eq cvalid(U(E)) = true .
eq cvalid(C(E1, U(E2)) = red(E1, E2) .
ceq cvalid(C(E1, C(E2, P))) = cvalid(C(E2, P))
                              if red(E1, E2)=true .
ceq cvalid(C(E1, C(E2, P))) = false if red(E1, E2)=false .
```

this function uses `red` to verify if the one-step sequential rewrite `E1 -> E2` is valid in the system.

Now we can express our properties in first order equational logic. For example the formula of the example in [section 5.1] is specified as:

$$\forall P : Comp.(\mathit{first}(P) = \mathit{aneven} \wedge \mathit{tterm}(\mathit{last}(P)) = \mathit{true} \wedge \mathit{cvalid}(P) \Rightarrow \mathit{even}(\mathit{last}(P)))$$

where *first* and *last* are functions to obtain the first and last expressions from a sequential rewrite, and *tterm* checks if a given expression is of sort `TTerm`.

This goal has been proved by the authors with the ITP tool v.013 and the Isabelle theorem prover [Nipkow et al.2002].

Although the proofs follows the steps of the inductive method proposed in [section 5.1], the proof is complicated by the use of the above auxiliary functions. Besides, the definition of the `red` operation may not be as direct as it would be desirable.

6 Conclusions

This paper continues the research initiated by M. Palomino in [Palomino 2003] of relating the CRWL logic with RL. However, our process for simulating CRWL programs with RL theories uses membership equational logic as the underlying logic for RL bringing off clearer Maude programs.

Nevertheless, our main goal behind simulating CRWL programs with Maude programs has been to study the benefits of applying the verification framework of RL to the non-terminating and non-deterministic CRWL programs. The obtained results are encouraging, in the sense that, they seem to be easily automatizable, specially the transformation process from CRWL programs to Maude theories; and the possibility of integrating the proposed proving method with the ITP tool would increment a lot the power of the verification process. The paper illustrates the proposed methodology on non-terminating and non-deterministic examples, taking into account the laziness of the functions.

On the other hand, the use of observations on the properties allows the user to decide the means by which he will observe the system without interfering with the system specification, and facilitates the verification of the properties.

We have in mind to implement the transformation process and the inductive method, in such a way that the use of RL will be transparent for the user. Furthermore we will explore new methods for automatize the given inductive proving method in a more direct way. We are also proving other kind of properties to study the potential of the proposed inductive method and to compare it with other approaches to the verification of nonterminating programs such as the different variants of temporal logics.

Acknowledgements.

Research supported by the Spanish Projects MELODIAS TIC2002-01167 and MIDAS TIC2003-01000. We are very grateful to Narciso Martí Oliet, Francisco López Fraguas and Miguel Palomino for all their comments to previous versions of this paper and to Manuel García Clavel for his help on the use of the ITP tool.

References

[Antoy 1997] S. Antoy. *Optimal Non-deterministic Functional Logic Computations*, Proc. ALP/HOA 1997, Springer LNCS 1298, pp. 16–30, 1997.

[Bruni and Meseguer 2003] R. Bruni and J. Meseguer, *Generalized rewrite theories*, ICALP 2003, LNCS 2719, pp 256–266 2003.

[Clavel et al. 1999] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude manual (version 2.1)* `http://maude.cs.uiuc.edu`, March 2004.

[Clavel 2001] M. Clavel. *The ITP tool.* In A. Nepomuceno, J. F. Quesada, and J. Salguero, editors, Logic, Language and Information. Proc. of the 1st Workshop on Logic and Language, Kronos, 55–62, 2001. System available at `http://www.ucm.es/info/dsip/clavel/itp`.

[Cleva et al. 2004] J.M. Cleva, J. Leach, F.J.López-Fraguas. *A logic programming approach to the verification of functional-logic programs.* Proc. Principles and Practice of Declarative Programming (PPDP'04), ACM Press, pp. 9–19, 2004.

[González et al. 1996] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas and M. Rodríguez-Artalejo. *A Rewriting Logic for Declarative Programming.* Proc. European Symp. on Programming (ESOP'96), Springer LNCS 1058, pp. 156–172, 1996.

[González et al. 1999] 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(1), pp. 47–87, 1999.

[Hanus 2003] M. Hanus (ed.), *Curry: an Integrated Functional Logic Language*, Version 0.8, April 15, 2003. `http://www-i2.informatik.uni-kiel.de/ ~curry/`.

[Hussmann 1992] H. Hussmann. *Nondeterministic Algebraic Specifications and Nonconfluent Term Rewriting.* Journal of Logic Programming 12, pp. 237–255, 1992.

[López and Sánchez 1999] F.J.López-Fraguas and J. Sánchez-Her-nández. *$\mathcal{TOY}$: A Multiparadigm Declarative System.* Proc. RTA'99, Springer LNCS 1631, pp 244–247, 1999.

[Meseguer 1992] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, *Theoretical Computer Science* 96, 1992, pages 73–155.

[Meseguer 1993] J. Meseguer, *A logical theory of concurrent objects and its realization in the Maude language*, in: G. Agha, P. Wegner, and A. Yonezawa (eds.), *Research Directions in Concurrent Object-Oriented Programming*, The MIT Press, 1993, pages 314–390.

[Martí and Meseguer 1999] N. Martí-Oliet and J. Meseguer. *Action and change in Rewriting Logic*. In Dynamic Worlds: From the frame problem to knwoledge management, R. Pareschi and B. Frohöfer (eds.), Applied Logic Series, Vol. 12, pp. 1–53, Kluwer Academic Publishers, 1999.

[Martí et al. 2005] N. Martí-Oliet, I. Pita, J.L. Fiadeiro, J. Meseguer and T. Maibaum. *A verification Logic for Rewriting Logic*. *Journal of Logic and Computation*. Vol. 15:3 , pp. 317–352, June 2005.

[Nipkow et al.2002] T. Nipkow, L.C. Paulson, M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higer-Order Logic*. Springer LNCS 2283, 2002.

[Palomino 2003] M. Palomino. *Comparing Meseguer's Re-writing Logic with the Logic CRWL*. International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001). Selected Papers. Elsevier ENTCS 66, 2001.

[Rodríguez 2001] M. Rodríguez-Artalejo. *Functional and Constraint Logic Programming*. in H. Comon, C. Marché and R. Treinen (eds.), *Constraints in Computational Logics, Theory and Applications*, Revised Lectures of the International Summer School CCL'99, Springer LNCS 2002, Chapter 5, pp. 202–270, 2001.