Semantic determinism and functional logic program properties[⋆]

José Miguel Cleva Francisco J. López-Fraguas

Dpto Sistemas Informáticos y Computación
Universidad Complutense de Madrid
e-mail {jcleva,fraguas}@sip.ucm.es

Abstract

In modern functional logic languages like Curry or Toy, programs are possibly non-confluent and non-terminating rewrite systems, defining possibly non-deterministic non-strict functions. Therefore, equational reasoning is not valid for deriving properties of such programs. In a previous work we showed how a mapping from *CRWL* –a well known logical framework for functional logic programming– into logic programming could be in principle used as logical conceptual tool for proving properties of functional logic programs. A severe problem faced in practice is that simple properties, even if they do not involve non-determinism, require difficult proofs when compared to those obtained using equational specifications and methods. In this work we improve our approach by taking into account determinism of (part of) the considered programs. This results in significant shortenings of proofs when we put in practice our methods using standard systems supporting equational reasoning like, e.g., Isabelle.

Keywords: program properties, functional-logic programming, semantic determinism

1 Introduction

A frequent claim about declarative languages is that the task of reasoning about programs is easier than in other programming paradigms because of the existence of an underlying logic providing more or less natural logical methods for that purpose. Although this assertion is essentially true, such logical methods do not come for free with the language, not even when it is provided with sound semantic foundations (e.g., logic or model theoretic) although these are of considerable help. Moreover, achieving effective methods in practice can be difficult.

In the case of modern functional logic programming (FLP, in short), realized in systems like Curry [9] or Toy [11], the main problem to face is that equational reasoning is not valid for reasoning about programs, which are constructor based rewrite systems possibly non-terminating and non-confluent. Semantically this leads to the

⋆ The authors have been partially supported by the Spanish projects TIN2005-09207-C03-03 'MERIT-FORMS' and S-0505/TIC/0407 'PROMESAS-CAM'.

presence of non-strict and non-deterministic functions, which have been shown to be quite useful for practical declarative programming.

When reasoning about functional logic programs, non-determinism precludes the direct use of well-known existing methods and tools developed for equational specifications, like those coming with *Isabelle* [13] or *Coq* [2], which usually require also termination. Rewriting logic in the sense of [12] and related verification tools [4] cannot be applied directly, since the semantics for non-determinism of rewriting logic is *run-time choice*, instead of the *call-time choice* criterion adopted in FLP [8].

In a previous work [5] we started what is, up to our knowledge, the first general framework for the verification of program properties for FLP with non-deterministic functions. Our work was based on *CRWL* [1] [7,8], a well-established semantic framework for FLP. The idea was to map *CRWL* into first order logic (FOL) in the following sense: the *CRWL*-semantics of a program P, given by a reduction relation $\rightarrow$, is expressed by means of a FOL theory, actually a logic program P_L, whose least model corresponds closely to the *CRWL*-initial model of P. Then FOL methods can be used to prove the properties of interest, which are those valid in the least model of the program.

In practice, large parts of programs are made of 'classical', deterministic, even terminating, functions. In the approach of [5], no benefit is taken from this knowledge, since the *CRWL* framework itself does not make any distinction between these well-behaved functions and the rest: in the reduction relation of *CRWL*, all functions are implicitly considered as potentially non-strict and non-deterministic. An unpleasant consequence is that the proofs of simple properties concerning deterministic functions are much more complex than the corresponding proofs using equational methods. For instance, the commutativity of the addition of natural numbers requires within *CRWL* a long proof in Isabelle (more than two pages) or ITP, while it is almost automatic using an equational specification of addition.

To overcome this problem we refine here the *CRWL* logic to take into account that certain fragments of a program can be deterministic. We prove the technical soundness of the refinement by means of an equivalence theorem with respect to the original logic. Therefore it allows us to specify equationally the deterministic parts of a program, resulting in much shorter proofs when using tools supporting equational reasoning, like Isabelle or ITP.

The remainder of the paper is organized as follows. The next section presents some preliminaries about *CRWL*. Section 3 is the core of the paper where we give semantic notions related to determinism, we propose a suitable refinement of *CRWL* related to them, and we prove an equivalence result for the two versions of *CRWL*. In Section 4 we discuss the application to the verification of program properties. Finally, Section 5 summarizes some conclusions. Proofs can be found in `http://gpd.sip.ucm.es/fraguas/rule06long.pdf`.

[1] *CRWL* stands for 'Constructor based ReWriting Logic'.

$0 + Y$	$\to Y$	$coin$	$\to 0$	$loop$	$\to loop$
$s(X) + Y$	$\to s(X + Y)$	$coin$	$\to s(0)$	$g(0)$	$\to 0$
$double(X)$	$\to X + X$	$f(X)$	$\to 0$	$g(X)$	$\to s(g(X))$

Fig. 1. *CRWL* sample program *Coin*

2 *CRWL* programs and their logical semantics

We recall the essential notions about *CRWL* needed for this work. See [8] for details and [5] for a discussion about slight changes in our presentation of *CRWL* with respect to the original one.

We assume a signature $\Sigma = CS_\Sigma \cup FS_\Sigma$ where $CS_\Sigma = \bigcup_{n\in\mathbb{N}} CS^n_\Sigma$ is a set of *constructor* symbols and $FS_\Sigma = \bigcup_{n\in\mathbb{N}} FS^n_\Sigma$ is a set of *function* symbols, all of them with associated arity and such that $CS_\Sigma \cap FS_\Sigma = \emptyset$. 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 CS_Σ and $\mathcal{V}$. The subindex Σ will usually be omitted. Expressions intend to represent possibly reducible expressions, while c-terms represent not further reducible data values.

The signature $\Sigma_\perp$ results of extending Σ with the new constant (0-arity constructor) $\perp$, that plays the role of the undefined value. The sets $Exp_\perp$ and $CTerm_\perp$ of (partial) expressions and (partial) c-terms respectively are built up using $\Sigma_\perp$. 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, s, t for c-terms, and $\bar{x}$ for tuples of x's. In all cases, primes (') and subindices can be used. Expressions can be compared by the *approximation ordering* $\sqsubseteq$, defined as the least partial ordering verifying: $\perp \sqsubseteq e$ and $e_1 \sqsubseteq e'_1 \wedge \ldots \wedge e_n \sqsubseteq e'_n \Rightarrow h(e_1, \ldots, e_n) \sqsubseteq h(e'_1, \ldots, e'_n)$, for $h \in CS^n \cup FS^n$.

We will use the sets of substitutions $CSubst = \{\theta : \mathcal{V} \to CTerm\}$ and $CSubst_\perp = \{\theta : \mathcal{V} \to CTerm_\perp\}$.

2.1 *The Proof Calculus for* CRWL

Along this paper a *CRWL-program* 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 $\perp$ does not occur in programs. We write P_f for the set of defining rules of f in P.

The *CRWL*-program in Fig. 1 will be used as an example to illustrate several points throughout the paper. It uses the data constructors 0 and s to represent natural numbers. Notice that the program is non-confluent (see *coin* and g) and non-terminating (see *loop* and g).

From a given program P, the proof calculus for *CRWL* can derive *reduction* or *approximation statements* of the form $e \to t$, with $e \in Exp_\perp$ and $t \in CTerm_\perp$. The intended meaning of such statement is that e can be reduced to t, where reduction may be done by applying rewriting rules of P or by replacing subterms of e by

$$\textbf{(BT)}\quad \frac{}{e \to \bot} \quad \text{for any } e \in \mathit{Exp}_\bot$$

$$\textbf{(CS)}\quad \frac{e_1 \to t_1 \;...\; e_n \to t_n}{c(e_1,...,e_n) \to c(t_1,...,t_n)} \quad c \in CS^n,\ t_i \in \mathit{CTerm}_\bot,\ e_i \in \mathit{Exp}_\bot$$

$$\textbf{(FR)}\quad \frac{e_1 \to t_1 \;...\; e_n \to t_n \quad e \to t}{f(e_1,...,e_n) \to t} \quad \text{if } f(t_1,...,t_n) \to e \in [P]_\bot$$

Fig. 2. The *CRWL* proof calculus

$\bot$. We write $P \vdash_{CRWL} e \to t$ to express derivability, and define the *denotation* of $e \in \mathit{Exp}_\bot$ as $[\![e]\!]^P = \{t \in \mathit{CTerm}_\bot : P \vdash_{CRWL} e \to t\}$. The superscript P is usually omitted.

When using a function rule R to derive statements, the calculus uses the so called *c-instances* of R, defined as $[R]_\bot = \{R\theta | \theta \in \mathit{CSubst}_\bot\}$. We write$[P]_\bot$ for 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 2 shows the proof calculus for *CRWL*. The rule (FR) allows to use c-instances of program rules to prove approximations. These c-instances may contain $\bot$ and by rule (BT) any expression can be reduced to $\bot$. This reflects a non-strict semantics, allowing non-terminating programs to have a meaning different from $\bot$. The use of c-instances in rule (FR) instead of general instances corresponds to *call-time choice* semantics for non-determinism [10,7,8]. In the example, it is possible to build a *CRWL*-proof for the reduction $double(coin) \to 0$ and also for $double(coin) \to s(s(0))$, but not for $double(coin) \to s(0)$. In contrast, $coin + coin$ can be reduced to $0, s(0)$ and $s(s(0))$. Call-time choice is related to *sharing*, a well known operational technique considered essential for the effective implementation of lazy functional languages like Haskell, and also adopted in existing FLP languages like Curry or Toy. 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 [1].

From the point of view of verifying properties of FLP programs, non-determinism and call-time choice semantics imply 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 rewriting allows to obtain $double(coin) \to s(0)$, which is not valid with call-time choice.

CRWL is provided also with a model-theoretic semantics, in which every program has a least Herbrand model, which is an initial model. See [8] for details.

3 An improved *CRWL*-calculus for deterministic program fragments

We remark that the *CRWL* calculus is a way of fixing the logical semantics of a program, determining formally the set of possible (partial) values of a given expression, but it is not meant as an operational procedure. As a matter of fact, the calculus has a certain degree of non-determinism other than that coming from program rules. For instance, using the program of Fig. 1, there are two *CRWL*-derivations for $coin+coin \to s(0)$, but this is natural since each *coin* can be reduced independently to 0 and $s(0)$. But we have also that $f(double(s(s(0))) \to 0$ has 152 (!) different *CRWL*-derivations, despite of the fact that in this case all the involved functions are deterministic. This fact causes no harm to the original purposes of *CRWL*, but it is a source of practical problems when trying to reason about properties of programs.

This section presents an improvement on the *CRWL* calculus to deal equationally with deterministic parts of a program. The first thing to do is to determine which notion of determinism is adequate.

3.1 Preliminary semantic concepts about determinism

We recall the definition $[\![e]\!] = \{t \in CTerm_\perp : e \to t\}$. It is an easy fact that for any e, $[\![e]\!]$ is not empty ($\perp \in [\![e]\!]$) and downward closed (i.e., $t \in [\![e]\!] \wedge t \sqsupseteq t' \Rightarrow t' \in [\![e]\!]$).

We need first some additional definitions about denotations of expressions.

Definition 3.1 Let $e \in Exp_\perp$.

(a) The *total denotation* of e is defined as $[\![e]\!]^T = \{t \in CTerm : t \in [\![e]\!]\}$. Trivially $[\![e]\!]^T \subseteq [\![e]\!]$.

(b) The expression e is *finite* iff $[\![e]\!]$ is a finite set.

(c) The expression e is (semantically) *totally defined* iff $[\![e]\!] = [\![e]\!]^T\downarrow$, where the downwards closure $S\downarrow$ of $S \subseteq CTerm_\perp$ is defined as $S\downarrow = \{t \in CTerm_\perp : t \sqsubseteq t'$ for some $t' \in S\}$.

Some examples follow, using the program of Fig. 1: for the expression $double(coin)$ we have $[\![double(coin)]\!] = \{\perp, 0, s(\perp), s(s(\perp)), s(s(0))\}$, while $[\![double(coin)]\!]^T = \{0, s(s(0))\}$. Hence $double(coin)$ is finite and totally defined. Denotations, even total denotations, can be infinite. For instance, $[\![g(0)]\!]^T = \{0, s(0), s(s(0)), \ldots\}$. Therefore $g(0)$ is infinite, and it is easy to see that it is also totally defined. Rather different is the case of $g(s(0))$, which is infinite since its denotation is $[\![g(s(0))]\!] = \{\perp, s(\perp), s(s(\perp)), \ldots\}$, but it is not totally defined, since $[\![g(s(0))]\!]^T = \emptyset$.

We give now a first notion of determinism of expressions and functions.

Definition 3.2

(a) An expression $e \in Exp_\perp$ is *deterministic* iff $[\![e]\!]$ is a directed set, that is, given $t, t' \in [\![e]\!]$ there exists $t'' \in [\![e]\!]$ such that $t \sqsubseteq t''$ and $t' \sqsubseteq t''$. A function $f \in FS$ is deterministic if for each $\bar{t} \in CTerm_\perp$, $f(\bar{t})$ is a deterministic expression.

(b) An expression $e \in Exp_\perp$ is *strongly deterministic* iff e is deterministic, finite and totally defined. A function $f \in FS$ is strongly deterministic if for every $\bar{t} \in CTerm$, $f(\bar{t})$ is also strongly deterministic.

According to these definitions, *coin*, *double*(*coin*) and $g(0)$ are examples of non-deterministic expressions; *double*($s(0)$) is strongly deterministic, as happens with $f(coin)$, despite of the presence of *coin*; *loop* and $g(s(0))$ are deterministic, but not strongly deterministic, because they are not totally defined. With respect to functions, $+$, *double* and f are strongly deterministic, *loop* is deterministic (but not strongly) and *coin* and g are not deterministic.

Notice that the property of being deterministic, as stated in (a), has nothing to do with non-termination and partiality. Those conditions also cause problems for equational reasoning and are typically forbidden in the equational part of proof assistants, like Isabelle, Coq or ITP. Although we do not need to use the formal notion of non-termination in our work, the notion of strong determinism intuitively tries to avoid it, as well as partiality.

We remark also that strong determinism is somehow related to the notion of confluence, but does not coincide with it for several reasons: first, confluence in the sense of ordinary rewriting is not adequate for *CRWL* due to call-time choice semantics, and there is no obvious alternative way of defining it. But even if we ignore this, strong determinism might hold for an expression in absence of confluence in the classical sense: consider for example a 0-ary function h defined by the rules $h \to 0$ and $h \to fail$, where there is no rule for $fail$; then the expression h is strongly deterministic but its set of rules is not confluent.

An interesting consequence of strong determinism is:

Proposition 3.3 *If an expression e is strongly deterministic then $[\![e]\!]^T$ is a unitary set, that is, $[\![e]\!]^T = \{t\}$. Such t is called* the value *of e.*

Notice that the opposite does not hold. Consider for example a 0-ary function h defined by the rules $h \to s(s(loop))$ and $h \to s(0)$. Then $[\![h]\!]^T = \{s(0)\}$ but h is not strongly deterministic as it is not deterministic, because $s(0), s(s(\bot)) \in [\![h]\!]$, but they have no common upper bound in $\sqsubseteq$.

The refinement of *CRWL* we are looking for will try to prove $e \to t$ by equational means, where e is strongly deterministic and t is its value. Strongly deterministic will play indeed an important role in the refinement, but still it is not sufficient for it, since strongly deterministic functions might use in their definitions non-deterministic functions. As a simple example, consider the function l defined by the rules $l(0) \to 0$, $l(s(0)) \to 0$, $l(s(s(X))) \to s(0)$, and the 0-ary $k \to l(coin)$. It is easy to see that k is strongly deterministic and its value is 0, but $k \to 0$ cannot be proved only by equational means, due to presence of *coin* in its definition.

For this reason, we strength the notion of strong determinism to the following one:

Definition 3.4 Let P be a *CRWL*-program, and $D \subseteq FS$ a set of function symbols verifying that all functions in D are strongly deterministic and all their defining rules use only function symbols from D. We say that $e \in Exp$ is *D-globally deterministic* if every function symbol of e is from D.

For a fixed P we assume also a fixed D, and the mention to D will be usually omitted.

In our example, we can set $D = \{+, double, f\}$, and therefore the globally deterministic expressions are those not containing function symbols other than those. Notice that globally deterministic expressions do not contain $\perp$. Notice also that if $e = f(e_1, \ldots, e_n)$ is globally deterministic then each e_i is also globally deterministic.

We will need the following natural result, though surprisingly technically not trivial:

Proposition 3.5 *If an expression $e \in Exp$ is globally deterministic then e is strongly deterministic.*

We are now prepared for presenting the announced refinement of *CRWL*.

3.2 A refined CRWL *proof calculus for equational reasoning*

The rules for the new calculus $CRWL_E$ can be found in Fig. 3. As it is apparent, the calculus consists of two sets of rules, one defining the relation $e \rightarrow t$, which is still the 'top-level' relation and has the same meaning as before, and the other defining the auxiliary relation $e = t$, reserved for globally deterministic expressions e, and with the meaning 't is the value of e'. The rule **(EQ)** connects both relations, by stating that the only way of deriving in $CRWL_E$ a reduction $e \rightarrow t$ for a globally deterministic expression e is to derive the value t' of e and then decrease t' in the ordering $\sqsupseteq$ to obtain t. This latter step is arguable: it is needed to guarantee the strong equivalence result given below; but if we admit a weaker correspondence between the two calculi, we could leave $e \rightarrow t'$ as the only possible reduction for the globally deterministic expression e. Something similar, but limited to the special case of c-terms, was done in [5], and contributes to further reducing the space of derivations.

We remark, as was already done for *CRWL*, that the refinement $CRWL_E$ should not be understood as an operational procedure, nor it is intended to achieve efficiency in the execution of deterministic parts. It does not even pursue to obtain shorter *CRWL*-derivations: for instance, at least one of the 152 *CRWL*-derivations for $f(double(s(s(0)))) \rightarrow 0$ –namely, that reducing $double(s(s(0)))$ to $\perp$, since f is not strict– is shorter than the only $CRWL_E$-derivation existing for such reduction, which requires to reduce $double(s(s(0)))$ to its value $s(s(s(s(0))))$. The purpose of $CRWL_E$ is to simplify the space of derivations and therefore the reasoning about programs, by using equations as much as possible.

We notice also that the symbol $=$ and their rules in the proof calculus are related to the joinability relation $\bowtie$ of [8] (strict equality in the systems Curry or Toy), but there are important differences, apart from their different purposes ($=$ is not a program construct). For instance, $coin \bowtie 0$ can be proved with the rules of [8], but $coin = 0$ is not provable in $CRWL_E$.

We finally remark that the condition of being globally deterministic is of semantic nature and, therefore, typically undecidable. To investigate sufficient decidable criteria would be of clear interest in practice, but it is out of the scope of this paper.

3.3 Relation between CRWL *and* $CRWL_E$

In this section we give the main results relating the reductions obtained from the original calculus *CRWL* and those obtained in $CRWL_E$.

Fig. 3. Proof calculus *CRWL*$_E$

(BT) $\dfrac{}{e \rightarrow \bot}$ e is not globally deterministic

(CS) $\dfrac{e_1 \rightarrow t_1 \; ... \; e_n \rightarrow t_n}{c(e_1, ..., e_n) \rightarrow c(t_1, ..., t_n)}$ $c \in CS^n,\ t_i \in CTerm_\bot$

and $c(\bar{e})$ is not globally deterministic

(FR) $\dfrac{e_1 \rightarrow t_1 \; ... \; e_n \rightarrow t_n \quad e \rightarrow t}{f(e_1, ..., e_n) \rightarrow t}$ if $t \not\equiv \bot, f(t_1, ..., t_n) \rightarrow e \in [P]_\bot$ and $f(\bar{e})$ is not globally deterministic

(EQ) $\dfrac{e = t'}{e \rightarrow t}$ if e globally deterministic and $t \sqsubseteq t'$

(CSE) $\dfrac{e_1 = t_1 \; ... \; e_n = t_n}{c(e_1, ..., e_n) = c(t_1, ..., t_n)}$ $c \in CS^n,\ t_i \in CTerm$

and $c(\bar{e})$ is globally deterministic

(FRE) $\dfrac{e_1 = t_1 \; ... \; e_n = t_n \quad e = t}{f(e_1, ..., e_n) = t}$ if $f(t_1, ..., t_n) \rightarrow e \in [P]$ and $f(\bar{e})$ is globally deterministic

The following lemma relates the reductions obtained in the equational part of *CRWL*$_E$ calculus with the original *CRWL* calculus.

Lemma 3.6 *Let P be a* CRWL *program, $e \in Exp$ a globally deterministic expression and $t \in CTerm$. Then*

$$P \vdash_{CRWL} e \rightarrow t \Leftrightarrow P \vdash_{CRWL_E} e = t$$

In other terms, the lemma ensures that the equational part of *CRWL*$_E$ exactly proves $e = t$ for the value t of e, as it was intended.

The next result shows the strong equivalence between *CRWL*$_E$ and *CRWL*, since the refinement preserves the reduction relation $\rightarrow$ of *CRWL*, for arbitrary partial expressions and c-terms.

Proposition 3.7 *Let P be a* CRWL *program, $e \in Exp_\bot$ and $t \in CTerm_\bot$. Then*

$$P \vdash_{CRWL_E} e \rightarrow t \Leftrightarrow P \vdash_{CRWL} e \rightarrow t$$

4 Application to the verification of CRWL program properties

The previous calculus can be used for verification of properties of functional logic programs. Two essential questions arise in this sense: which is the language of the properties of interest? what means validity for a given property?

For verification purposes, the properties we are interested in are those concerning the possible reductions of expressions in the $CRWL_E$ calculus. Then the properties are specified as formulae over the relations $\rightarrow$ and $=$. In many cases, we want the quantifiers to range over a restricted universe as we are interested in properties valid only for $CTerm$ or $CTerm_\perp$; therefore we also include two predicates in the language of properties to deal with such restriction, namely *tot* and *term* respectively. We can also be interested in properties that have to do with globally deterministic expressions. For that reason, we introduce another predicate *gd* for those expressions. Summarizing, properties are expressed as first order logic (FOL) formulae over the relations $\{\rightarrow, =, tot, term, gd\}$.

With respect to validity, in [5] we translated the *CRWL* calculus into FOL by associating a logic program to a *CRWL*-program, and we verified properties of the *CRWL* program in the least model of the associated logic program. We follow here a similar approach for $CRWL_E$: defining a logic program and proving properties in the least model of the logic program. For this purpose we consider the logic program P_L associated to a program P of *CRWL*. Notice that, although programs do not change when moving from *CRWL* to $CRWL_E$, the associated logic programs do, because the logic has changed. In this case we need also to define the three auxiliary predicates mentioned above to distinguish between different kinds of expressions, another predicate *ngd* for expressions non globally deterministic and the relation for the *CRWL* approximation ordering $\sqsubseteq$, called *approx* in the logic program. The logic program P_L for every *CRWL* program P is obtained using the rules of Figure 4, where *ngd* and *term* are defined in a similar way as *gd* and *tot* respectively. The implication symbol in clauses is written as $\Leftarrow$.

Since in this approach validity of a given property of a *CRWL*-program P, expressed as a FOL formula φ, means validity of φ in the least model of the corresponding logic program P_L, it is important to ensure that the logic of P_L (given by FOL) and the logic of P (given by *CRWL* and $CRWL_E$) have a good correspondence. The following results relate both.

Proposition 4.1 *Let P be a CRWL-program and P_L its corresponding logic program. Then, for any expression e and term t,*
(i) $P_L \models e = t \Leftrightarrow P \vdash_{CRWL_E} e = t.$
(ii) $P_L \models e \rightarrow t \Leftrightarrow P \vdash_{CRWL_E} e \rightarrow t$
(iii) $P_L \models term(e) \Leftrightarrow e \in \mathrm{CTerm}_\perp$
(iv) $P_L \models tot(e) \Leftrightarrow e \in \mathrm{CTerm}$
(v) $P_L \models gd(e) \Leftrightarrow e$ *is globally deterministic*
(vi) $P_L \models approx(e, e') \Leftrightarrow e \sqsubseteq e'$

Therefore, by propositions 3.6 and 3.7 we have the following corollary:

$X \to \perp \Leftarrow \mathit{ngd}(X)$
$X \to T \Leftarrow \mathit{gd}(X) \wedge X = T' \wedge \mathit{approx}(T, T')$
For every $c \in \mathit{CS}$:
$c(E_1, \ldots, E_n) \to c(T_1, \ldots, T_n) \Leftarrow E_1 \to T_1 \wedge \ldots \wedge E_n \to T_n \wedge \mathit{ngd}(c(E_1, \ldots, E_n))$
For every $f \in \mathit{FS}$ and every rule $f(t_1, \ldots, t_n) = e \in P$:
$f(E_1, \ldots, E_n) \to T \Leftarrow E_1 \to t_1 \wedge \ldots \wedge E_n \to t_n \wedge e \to T \wedge \mathit{ngd}(f(E_1, \ldots, E_n))$
For every $c \in \mathit{CS}$:
$c(E_1, \ldots, E_n) = c(T_1, \ldots, T_n) \Leftarrow E_1 = T_1 \wedge \ldots \wedge E_n = T_n \wedge \mathit{gd}(c(E_1, \ldots, E_N))$
For every $f \in \mathit{FS}$ and every rule $f(t_1, \ldots, t_n) = e \in P$:
$f(E_1, \ldots, E_n) = T \Leftarrow E_1 = t_1 \wedge \ldots \wedge E_n = t_n \wedge e = T \wedge \mathit{gd}(f(E_1, \ldots, E_n))$
For every $c \in \mathit{CS}$:
$\mathit{gd}(c(E_1, \ldots, E_n)) \Leftarrow \mathit{gd}(E_1) \wedge \ldots \wedge \mathit{gd}(E_n)$
For every $f \in \mathit{GDFS}$:
$\mathit{gd}(f(E_1, \ldots, E_n)) \Leftarrow \mathit{gd}(E_1) \wedge \ldots \wedge \mathit{gd}(E_n)$
For every $c \in \mathit{CS}$:
$\mathit{tot}(c(E_1, \ldots, E_n)) \Leftarrow \mathit{tot}(E_1) \wedge \ldots \wedge \mathit{tot}(E_n)$
$\mathit{approx}(\perp, X)$
For every $c \in \mathit{CS}$:
$\mathit{approx}(c(E_1, \ldots, E_n), c(E'_1, \ldots, E'_n)) \Leftarrow \mathit{approx}(E_1, E'_1) \wedge \ldots \wedge \mathit{approx}(E_n, E'_n)$

Fig. 4. Logic program obtained from CRWL_E

Corollary 4.2 *Let P be a CRWL-program and P_L its corresponding logic program. Then, for any $e \in Exp_\perp$ and $t \in CTerm_\perp$,*
(i) $P_L \models e \to t \Leftrightarrow P \vdash_{\mathit{CRWL}} e \to t$.
(ii) *If e is globally deterministic, and $t \in CTerm$, then $P_L \models e = t \Leftrightarrow P \vdash_{\mathit{CRWL}} e \to t \Leftrightarrow t$ is the value of e.*

We conclude that the reductions obtained from the logic program are the same as those for the original *CRWL* program. Also we have that when we refer to an equation $e = t$ it is because t is the value of e in *CRWL*.

4.1 Practical aspects of the approach

The approximation of [5], translating *CRWL* into a logic program for the verification of properties was tested in various existing theorem provers. We have tested the new approximation in the theorem prover Isabelle [13]. The translation of the reduction process into Isabelle is done in two steps. First we define a function `red` for the equational part of the program, thus all globally deterministic functions define a unique reduction via the program rules. Such program rules for the collection of globally deterministic functions form the definition of the function `red`. We also define as functions the predicates *term*, *tot*, *gd*, *ngd* and the relation *approx* that appear in the logic program translation. Based on this translation procedure we can transform *CRWL* programs into Isabelle specifications. Consider the *CRWL* example on Figure 1 restricted to the functions $+$, *double* and *coin*. The corresponding Isabelle specification of the associated logic program is partially shown in Figure 5.

The formulas to specify properties are also transformed when considering this Isabelle specification. As the relation $=$ is now specified as a function `red` and it is only defined for globally deterministic expressions, a statement of the form $e = t$ in

```
inductive arrow
intros
bt [intro]: "ngd(x) ==> (x, bottom) : arrow"
dcs [intro]: "[|ngd(s x) ; (x, t):arrow|] ==> ((s x), (s t)):arrow"
fcoin1 [intro]: "(zero, t):arrow ==> (coin, t):arrow"
fcoin2 [intro]: "(s(zero), t):arrow ==> (coin, t):arrow"
sum1 [intro]: "[|ngd(sum x y) ; (x, zero):arrow ; (y, t):arrow|]
              ==> (suma x y, t):arrow"
sum2 [intro]: "[|ngd(sum x y) ; (x, s(t1)):arrow ; (y,t2):arrow ;
              (s(sum t1 t2), t):arrow|] ==> (sum x y , t):arrow"
double [intro]: "[|ngd(double x) ; (x, t1):arrow ; (sum t1 t1,t):arrow|]
                 ==> (double(x), t):arrow"
eq [intro]: "[|gd(x) ; red(x)=r ; approx(r, t)|] ==>(x, t):arrow"

recdef  red "measure number"
"red(zero) = zero"
"red(s x) = s(red x)"
"red(sum zero y) = red y"
"red(sum (s x) y) = s(red(sum x y))"
"red(double x) = red(sum x x)"
```

Fig. 5. Part of Isabelle specification for *Coin*

the logic is formulated as `gd(e) & red(e)=t` For instance, the formula:

$$\forall X, Y, T.(tot(X) \wedge tot(Y) \wedge X + Y = T \Rightarrow Y + X = T) \qquad (1)$$

is transformed into:

$$\forall X, Y, T.(tot(X) \wedge tot(Y) \wedge gd(X + Y) \wedge \texttt{red}(X + Y) = T \Rightarrow gd(Y + X) \wedge \texttt{red}(Y + X) = T)$$

With the Isabelle specification of Figure 5 the proof of this property is very short, similar to the proof that one could obtain if the program was functional. This contrasts with the results of using *CRWL* instead of $CRWL_E$, as done in [5]. The first thing to note is that the formula (1) is not expressible, since within *CRWL* the relation = simply does not exist. The closest formula would be:

$$\forall X, Y, T.(tot(X) \wedge tot(Y) \wedge X + Y \to T \Rightarrow Y + X \to T) \qquad (2)$$

which requires a rather long proof in an Isabelle specification of *CRWL*. To be more fair, we could compare the complexity of the proof of the property (2) with our refinement. Again, the use of $CRWL_E$ is successful, since the resulting Isabelle proof is three times shorter than in the case of using *CRWL*.

5 Conclusions

In this paper we have made some progress towards achieving effective methods for verifying properties of functional logic programs where non-deterministic functions are permitted.

The work was motivated by the following fact: to specify the underlying logic (*CRWL* [8]) of functional logic programming in other formalisms like first order logic (as in [5]) or rewriting logic (as in [6]), is a good conceptual starting point for verifying properties of those programs, but it is not enough in practice, since simple properties might require complex proofs. This happens because the possibility of non-determinism spreads over the whole logic, even if large parts of a program are purely deterministic. A typical example of such situation is commutativity of the addition of natural numbers, whose proof, if equationally specified, is almost automatic in most proof assistants, but requires a long proof in the approach of [5].

Let us give a succinct summary of our contributions to overcome this problem:

- We have identified, within the *CRWL* framework, a notion of semantic determinism appropriate to our purposes.
- We have refined the *CRWL* logic in such a way that reduction statements involving only deterministic expressions can be derived with equational-like reasoning, thus reducing enormously the indeterminism inherent to derivations in the original *CRWL* calculus.
- We have proved the correctness of the refinement through an equivalence result.
- We have applied the refined logic to our main aim, the proof of properties of *CRWL*-programs, following a similar scheme to that of [5]: the (refined) *CRWL*-logical semantics of a program is specified as a logic program; the properties to verify are first order formulae over the involved relations, and validity of a property means validity in the least model of that logic program.
- We have used Isabelle [13] to check our ideas in practice. In particular, we obtain a much shorter proof in the example of commutativity of addition.

Our improvement seems practical enough to continue in several ways. First, determining effective sufficient conditions ensuring determinism; in this sense, maybe the techniques in [14,3] could be useful. We are interested also in investigating weaker (but still applicable to our purposes) notions of determinism that will enlarge the deterministic part of the program in which proving properties will be more effective. Finally, we also plan to develop a set of non-trivial case studies for a better evaluation of our methods. This was almost impossible prior to this work due to the complexity of proofs of previous approaches.

Acknowledgements

We thank an anonymous reviewer for pointing out some technical problems in the preliminary version of this paper.

References

[1] S. Antoy. *Optimal Non-deterministic Functional Logic Computations*, Proc. Algebraic and Logic Prog. (ALP'97), Springer LNCS 1298, pp. 16–30, 1997.

[2] Y. Bertot, P. Casteran. *Interactive Theorem Proving and Program Development CoqArt: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science. Springer, 2004.

[3] B. Braßel, M. Hanus. *Nondeterminism Analysis of Functional Logic Programs*, Proc. Int. Conf. on Logic Programming (ICLP'05), Springer LNCS 3668, pp. 265–279, 2005.

[4] M. Clavel, M. Palomino. *A quick ITP tutorial*, Proc. V Jornadas sobre Programación y Lenguajes (PROLE'05), Thomson, pp. 159–172, 2005. An extended version will appear in Journal of Universal Computer Science.

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

[6] J.M. Cleva, I. Pita. *An approach to the verification of CRWL programs with rewritng logic*. Proc. V Jornadas sobre Programación y Lenguajes (PROLE'05), Thomson, pp. 138–148, 2005. An extended version will appear in Journal of Universal Computer Science.

[7] 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.

[8] 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.

[9] M. Hanus (ed.), *Curry: an Integrated Functional Logic Language*, Version 0.8.2, March 28, 2006. `http://www-i2.informatik.uni-kiel.de/∼curry/`.

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

[11] F.J.López Fraguas, J. Sánchez Hernández. *$\mathcal{TOY}$: A Multiparadigm Declarative System*. Proc. Rewriting Techniques and Applications (RTA'99), Springer LNCS 1631, pp 244–247, 1999.

[12] J. Meseguer. *Conditional Rewriting Logic as a Unified Model of Concurrency*. Theoretical Computer Science 96, pp. 73–155, 1992.

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

[14] R. Peña-Marí, Clara Segura. *Non-determinism analyses in a parallel-functional language*, Journal of Functional Programming 15 (1), pp. 67–100, 2005.