Relating two semantic descriptions of functional logic programs

(Work in progress) [1]

F.J. López-Fraguas [2] J. Rodríguez-Hortalá [2]
J. Sánchez-Hernández [2]

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
Madrid, Spain

Abstract

A distinctive feature of modern functional logic languages like Toy or Curry is the possibility of programming non-strict and non-deterministic functions with call-time choice semantics. For almost ten years the CRWL framework [6,7] has been the only formal setting covering all these semantic aspects. But recently [1] an alternative proposal has appeared, focusing more on operational aspects. In this work we investigate the relation between both approaches, which is far from being obvious due to the wide gap between both descriptions, even at syntactical level.

Key words: Functional logic programming, Semantic equivalence

1 Introduction

In its origin functional logic programming (FLP) did not consider non-deterministic functions (see [8] for a survey of that era). Inspired in those ancestors and in Hussmann's work [12], the *CRWL* framework [6,7] was proposed in 1996 as a formal basis for *FLP* having as main notion that of non-strict non-deterministic function with call-time choice semantics. At the operational level, modern FLP has been mostly influenced by the notions of definitional trees [2] and needed narrowing [3].

Both approaches –*CRWL* and needed narrowing– coexist with success in the development of FLP (see [15,9] for recent respective surveys). It is tacitly accepted in the FLP community that they essentially speak of the same

[1] Work partially supported by the Spanish projects TIN2005-09207-C03-03 (MERIT-FORMS-UCM) and S-0505/TIC/0407 (PROMESAS-CAM).

[2] Email: {`fraguas,juan,jaime`}`@sip.ucm.es`

'programming stuff', realized by systems like Curry [11] or Toy [14], but up to now they remain technically disconnected. One of the reasons has been that the formal setting for needed narrowing is classical rewriting, which is known to be unsound for call-time choice, which requires sharing.

But recently [1] a new operational formal description of FLP has been proposed, coping with narrowing, residuation, laziness, non-determinism and sharing, for a language called here *FLC* for its proximity to *Flat Curry* [10].

There is a long distance in the formal aspects of the two approaches, each one having its own merit: *CRWL* provides a concise and clear way for giving logical semantics to programs, with a high level of abstraction and a syntax close to the user, while *FLC* and its semantics are closer to computations and concrete implementations with details about variable bindings representation.

The goal of our work is to relate both approaches in a technically precise manner. In this way, some known or future results obtained for one of them could be applied to the other.

The rest of the paper is organized as follows. Sections 2 and 3 present the essentials of *CRWL* and *FLC* needed to relate them. Section 4 sets some restrictions assumed in our work and gives an overview of the structure of our results. Section 5 relates *CRWL* to $CRWL_{FLC}$, a new intermediate formal description. Section 6 is the main part of the work and studies the relation between $CRWL_{FLC}$ and *FLC*. Section 7 gives some conclusions. Proofs are mostly omitted and some of them are still under development.

2 The *CRWL* Framework: a Summary

We assume a signature $\Sigma = CS \cup FS$, where CS (FS) is a set of constructor symbols (defined function symbols) each of them with an associated arity; we sometimes write CS^n (FS^n resp.) to denote the set of constructor (function) symbols of arity n. As usual notations write $c, d \ldots$ for constructors, $f, g \ldots$ for functions and $x, y \ldots$ for variables taken from a numerable set $\mathcal{V}$.

The set of expressions *Exp* is defined as usual: $e ::= x \mid h(e_1, \ldots, e_n)$, where $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. The set of *constructed* terms is defined analogously but with h restricted to CS, i.e., function symbols are not allowed. The intended meaning is that *Exp* stands for evaluable expressions while $CTerm$ are data terms. We will also use the extended signature $\Sigma_\perp = \Sigma \cup \{\perp\}$, where $\perp$ is a new constant (0-arity constructor) that stands for *undefined value*. Over this signature we build the sets $Exp_\perp$ and $CTerm_\perp$ in the natural way. The set $CSubst$ ($CSubst_\perp$ resp.) stands for substitutions or mappings from $\mathcal{V}$ to $CTerm$ ($CTerm_\perp$ resp.). Both kind of substitutions will be written as $\theta, \sigma \ldots$. The notation $\sigma\theta$ denotes the composition of substitutions in the usual way. The notation $\overline{o}$ stands for tuples of any of the previous syntactic constructions.

The original *CRWL* logic introduces strict equality as a built-in constraint and program-rules optionally contain a sequence of equalities as condition. In

$$\textbf{(B)}\ \frac{}{e \rightarrow \perp} \qquad \textbf{(RR)}\ \frac{}{x \rightarrow x} \quad x \in \mathcal{V}$$

$$\textbf{(DC)}\ \frac{e_1 \rightarrow t_1\ \ldots\ e_n \rightarrow t_n}{c(e_1, \ldots, e_n) \rightarrow c(t_1, \ldots, t_n)} \quad c \in CS^n,\ t_i \in CTerm_\perp$$

$$\textbf{(Red)}\ \frac{e_1 \rightarrow t_1\theta\ \ldots\ e_n \rightarrow t_n\theta \quad e\theta \rightarrow t}{f(e_1, \ldots, e_n) \rightarrow t} \quad \begin{array}{l}(f(t_1, \ldots, t_n) = e) \in \mathcal{P} \\ \theta \in CSubst_\perp\end{array}$$

Fig. 1. Rules of *CRWL*

the current work, as *FLC* does not consider built-in equality, we restrict the class of programs. Then a *CRWL*-program $\mathcal{P}$ is a set of rules of the form: $f(\bar{t}) = e$, where $f \in FS^n$, $\bar{t}$ is a linear (without multiple occurrences of the same variable) n-tuple of c-terms and $e \in Exp$. We write $\mathcal{P}_f$ for the sef of rules defining f.

Rules of *CRWL* (without equality) are presented in Figure 1. Rule **(B)** allows any expression to be undefined or not evaluated (non-strict semantics). Rule **(Red)** is a proper reduction rule: for evaluating a function call it uses a compatible program-rule, makes the parameter passing (by means of a substitution θ) and then reduces the body. This logic proves *approximation* statements of the form $e \rightarrow t$, where $e \in Exp_\perp$ and $t \in CTerm_\perp$. Given a program $\mathcal{P}$, the denotation of an expression e with respect to *CRWL* is defined as $[\![e]\!]^{\mathcal{P}}_{CRWL} = \{t \mid e \rightarrow t\}$.

3 The *FLC* Language and its Natural Semantics

The language *FLC* considered in [1] is a convenient –although somehow low-level– format to which functional logic programs like those of Curry or Toy can be transformed (not in a unique manner). This transformation embeds important aspects of the operational procedure of FLP languages, like are definitional trees and inductive sequentiality.

The syntax of *FLC* is given in Fig. 2. Notice that each function symbol f has exactly one definition rule $f(x_1, \ldots, x_n) = e$ with distinct variables $x_1, \ldots, x_n$ as formal parameters. All non-determinism is expressed by the use of *or* choices in right-hand sides and also all pattern matching has been moved to right-hand sides by means of nesting of *(f)case* expressions. *Let bindings* are a convenient way to achieve sharing.

An additional *normalization step* over programs is assumed in [1]. In normalized expressions each constructor o function symbol appears applied only to distinct variables. This can be achieved via let-bindings. The normalization of e is written as e^*.

In [1] two operational semantics are given to *FLC*: a natural (*big-step*) semantics in the style of Launchbury's semantics [13] for lazy evaluation (with

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textit{Programs}\colon\ P \ ::=\ D_1 \ldots D_m} \\
\multicolumn{3}{l}{\textit{Function definitions}\colon\ D \ ::=\ f(x_1, \ldots, x_n) = e} \\
\multicolumn{3}{l}{\textit{Expressions}} \\
e \ ::= & x & \text{(variable)} \\
\quad \mid & c(e_1, \ldots, e_n) & \text{(constructor call)} \\
\quad \mid & f(e_1, \ldots, e_n) & \text{(function call)} \\
\quad \mid & \mathit{case}\ e\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} & \text{(rigid case)} \\
\quad \mid & \mathit{fcase}\ e\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} & \text{(flexible case)} \\
\quad \mid & e_1\ \mathit{or}\ e_2 & \text{(disjunction)} \\
\quad \mid & \mathit{let}\ x_1 = e_1, \ldots, x_n = e_n\ \mathit{in}\ e & \text{(let binding)} \\
\multicolumn{3}{l}{\textit{Patterns}\colon\ p \ ::=\ c(x_1, \ldots, x_n)}
\end{array}
$$

Fig. 2. Syntax for FLC programs

sharing) for functional programming, and a small step semantics. CRWL itself being a big-step semantics, it seems more adequate to compare it to the natural semantics of [1], which is shown [3] in Fig. 3. It consists of a set of rules for a relation $\Gamma : e \Downarrow \Delta : v$, indicating that one of the possible evaluations of e ends up with the head normal form (variable or constructor rooted) v. Γ, Δ are *heaps* consisting of bindings $x \mapsto e$ for variables. An initial configuration has the form $[] : e$.

$$
\begin{array}{rll}
\textbf{(VarCons)} & \Gamma[x \mapsto t] : x \Downarrow \Gamma[x \mapsto t] : t & t \text{ constructor-rooted} \\[2ex]
\textbf{(VarExp)} & \dfrac{\Gamma[x \mapsto e] : e \Downarrow \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto v] : v} & \begin{array}{l} e \text{ not constructor-rooted,} \\ e \neq x \end{array} \\[3ex]
\textbf{(Val)} & \Gamma : v \Downarrow \Gamma : v & v \text{ constructor-rooted or variable with } \Gamma[v] = v \\[2ex]
\textbf{(Fun)} & \dfrac{\Gamma : e\rho \Downarrow \Delta : v}{\Gamma : f(\overline{x_n}) \Downarrow \Delta : v} & f(\overline{y_n}) = e\ \in P \text{ and } \rho = \{\overline{y_n \mapsto x_n}\} \\[3ex]
\textbf{(Let)} & \dfrac{\Gamma[\overline{y_k \mapsto e_k\rho}] : e \Downarrow \Delta : v}{\Gamma : \mathit{let}\ \{\overline{x_k = e_k}\}\ \mathit{in}\ e \Downarrow \Delta : v} & \begin{array}{l} \rho = \{\overline{x_k \mapsto y_k}\} \\ \text{and } \overline{y_k} \text{ are fresh variables} \end{array} \\[3ex]
\textbf{(Or)} & \dfrac{\Gamma : e_i \Downarrow \Delta : v}{\Gamma : e_1 \mathit{or}\ e_2 \Downarrow \Delta : v} & i \in \{1, 2\} \\[3ex]
\textbf{(Select)} & \dfrac{\Gamma : e \Downarrow \Delta : c(\overline{y_n}) \qquad \Delta : e_i\rho \Downarrow \Theta : v}{\Gamma : (f)\mathit{case}\ e\ \mathit{of}\ \{\overline{p_k \mapsto e_k}\} \Downarrow \Theta : v} & \begin{array}{l} p_i = c(\overline{x_n}) \\ \text{and } \rho = \{\overline{x_n \mapsto y_n}\} \end{array}
\end{array}
$$

Fig. 3. Natural Semantics for *FLC*

[3] The rule *Guess* of [1] is skipped due to some restrictions to be imposed in the next section.

4 *CRWL* vs. *FLC*: Working Plan

In order to establish the relation between *CRWL* and *FLC* (in Section 6) firstly we adapt *CRWL* to the syntax of *FLC*. For this purpose we introduce the rewriting logic $CRWL_{FLC}$ as a variant of *CRWL* with specific rules for managing *let*, *or* and *case* expressions.

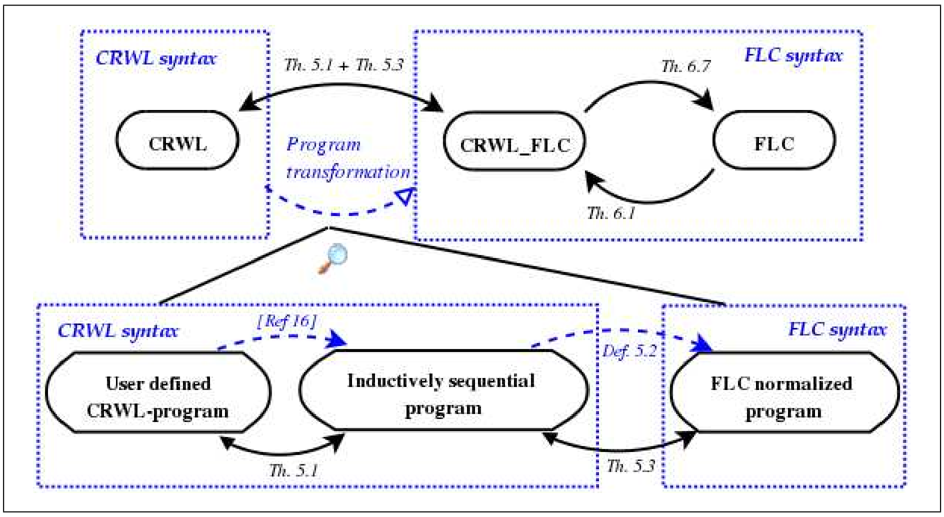

Fig. 4. Proof's plan

The relation between *CRWL* and *FLC* is established through this intermediate logic. The working plan is sketched in Figure 4. Given a pair program/expression in *CRWL* we transform them into *FLC*-syntax and study the semantic equivalence of both versions of *CRWL* (Theorems 5.1 and 5.2). Then we focus on the equivalence of *FLC* with respect to $CRWL_{FLC}$ in a common syntax context (Theorems 6.5 and 6.1). *FLC* and *CRWL* are very different frameworks from the syntactical and the semantical points of view. The advantage of splitting the problem is that on one hand both versions of *CRWL* are very close from the point of view of semantics; on the other hand $CRWL_{FLC}$ and *FLC* share the same syntax. The syntactic transformation and its correctness will be explained in Sect. 5.1.

There are important differences between *FLC* and $CRWL_{FLC}$ that complicates the task of relating them. The heaps used in *FLC* for storing variable bindings have not any (explicit) correspondence in *CRWL*. Another important difference is that the first one obtains *head normal forms* for expressions, while the second is able to obtain any value of the denotation of an expression (in particular a normal form if it exists).

Differences do not end here. There are still two important points that enforces us to take some decisions: (1) *FLC* performs narrowing while *CRWL* is a pure rewriting relation. In this paper we address this inconvenience by considering only the rewriting fragment of *FLC*. Narrowing acts in *FLC* either due to the presence of logical variables in expressions to evaluate or because of the use of extra variables in program rules (those not appearing in left-hand sides). So we can isolate the rewriting fragment by excluding this kind of variables throughout this work. (2) The other difference is due to the fact that *FLC* allows recursive *let* constructions. Since there is not a well established consensus about the semantics of such constructions in a non-deterministic context, and furthermore they cannot be introduced in the transformation of *CRWL*-programs, we exclude recursive *let*'s from the language in this work. Once this decision is taken it is not difficult to see that a *let* with multiple variable bindings may be expressed as a sequence of nested *let*'s, each with a unique binding. For simplicity and without loss of generality we will consider only this kind of *let*'s. We assume from now on that programs and expressions fulfil the conditions imposed in (1) and (2).

5 The proof calculus $CRWL_{FLC}$

The rewriting logic $CRWL_{FLC}$ preserves the main features of *CRWL* from a semantical point of view, but it uses the *FLC*-syntax for expressions and programs. In particular it allows *let*, *case* and *or* constructs, but like *CRWL* it proves statements of the form $e \to t$ where $t \in CTerm_\perp$.

$$\textbf{(B)}\quad \frac{}{e \to \perp} \qquad\qquad \textbf{(RR)}\quad \frac{}{x \to x} \qquad x \in \mathcal{V}$$

$$\textbf{(DC)}\quad \frac{e_1 \to t_1 \ \ldots \ e_n \to t_n}{c(e_1,\ldots,e_n) \to c(t_1,\ldots,t_n)} \qquad c \in CS^n,\ t_i \in CTerm_\perp$$

$$\textbf{(Red)}\quad \frac{e\theta \to t}{f(\overline{t}) \to t} \qquad (f(\overline{y}) = e) \in \mathcal{P},\ \theta = [\overline{y/t}]$$

$$\textbf{(Case)}\quad \frac{e \to c(\overline{t}) \qquad e_i\theta \to t}{case\ e\ of\ \{\overline{p_i \to e_i}\} \to t} \qquad \begin{array}{l} p_i = c(\overline{x}) \text{ for some } i \\ \theta = [\overline{x/t}] \end{array}$$

$$\textbf{(Or)}\quad \frac{e_i \to t}{e_1\ or\ e_2 \to t} \qquad \text{for some } i \in \{1,2\}$$

$$\textbf{(Let)}\quad \frac{e' \to t' \quad e[x/t'] \to t}{let\ \{x = e'\}\ in\ e \to t}$$

Fig. 5. Rules of $CRWL_{FLC}$

Rules of $CRWL_{FLC}$ are presented in Figure 5. The first three ones **(B)**, **(RR)** and **(DC)** are directly incorporated from *CRWL*. Rules **(Case)**, **(Or)** and **(Let)** have also a clear reading. Finally, rule **(Red)** is a simplified version of the corresponding in *CRWL*, as now we can guarantee that any function call in a derivation can only use c-terms as arguments. This is easy to check: the initial expression to reduce is in normalized form (arguments are all variables) and the substitutions applied by the calculus (in rules **(Red)**, **(Case)** and **(Let)**) can only introduce c-terms. Given a program $\mathcal{P}$ the denotation of an expression e with respect to $CRWL_{FLC}$ is defined as $[\![e]\!]^{\mathcal{P}}_{CRWL_{FLC}} = \{t \mid e \rightarrow t\}$.

5.1 Relation between $CRWL_{FLC}$ and CRWL

We obtain here an equivalence result for $CRWL_{FLC}$ and *CRWL*. A skeleton of the proof is given in the zoomed part of Fig 4. It is based on a program transformation from *CRWL*-syntax (user syntax) to *FLC*-syntax. A similar translation is assumed but not made explicit in [1]. For technical convenience we split the transformation into two parts: first, and still within *CRWL*-syntax, we transform P into another program P' which is *inductively sequential* ([2,9]), except for a function *or* defined by the two rules $X\ or\ Y = X$ and $X\ or\ Y = Y$. The function *or* concentrates all the non-sequentiality (hence, all the indeterminism) of functions in right-hand sides. We speak of 'inductively sequential with *or*' (IS_{or}) programs. Alternatively, programs can be transformed into overlapping inductively sequential format (see [9]), where a function might have several rules with the same left-hand side (as happens with the rules of *or*). Both formats are easily interchangeable. Such kind of transformations are well-known in functional logic programming. In the *CRWL* setting, a particular transformation has been proposed in [16], where it is proved the following result:

Theorem 5.1 *Let P be a CRWL-program and e an expression.*
Then $[\![e]\!]^{P}_{CRWL} = [\![e]\!]^{P'}_{CRWL}$ where P' is the IS_{or} transformed program of P.

Now, to transform IS_{or} programs into (normalized) *FLC*-syntax can be done by simply mimicking the inductive structure of function definitions by means of (possibly nested) *case* expressions. We omit the concrete algorithm due to the lack of space. Instead, we give in Fig. 6 an example of the two program transformation steps (first to IS_{or}, then to *FLC*). Notice that the final *FLC*-program does not contain rules for *or*, since it is included in the syntax of *FLC*, and there is a specific rule governing its semantics in the $CRWL_{FLC}$-calculus.

The following equivalence result states the correctness of the transformation.

Theorem 5.2 *Let P be an IS CRWL-program and, e an CRWL-expression, and $\hat{P}, \hat{e}$ their FLC-transformations. Then $[\![e]\!]^{P}_{CRWL} = [\![\hat{e}]\!]^{\hat{P}}_{CRWL_{FLC}}$.*

6 Relation between $CRWL_{FLC}$ and *FLC*

We need some more technical preliminary notions:

- $dom(\Gamma)$: The set of variables bound in the heap Γ.

Constructor symbols: 0$\in CS^0$, s$\in CS^1$

Source CRWL-program

f(0,Y) = s(Y)
f(X,0) = X
f(s(X),s(Y)) = s(f(X,Y))

Transformed IS_{or} CRWL-program

f(X,Y) = f$_1$(X,Y) or f$_2$(X,Y)
f$_1$(0,Y) = s(Y)
f$_1$(s(X),s(Y)) = s(f(X,Y))
f$_2$(X,0) = X
X or Y = X   X or Y = Y

Transformed normalized FLC-program

f(X,Y) = f$_1$(X,Y) *or* f$_2$(X,Y)
f$_1$(X,Y) = *case* X *of* { 0 $\rightarrow$ s(Y);
          s(X$_1$) $\rightarrow$ *case* Y *of* { s(Y$_1$) $\rightarrow$ *let* U=f(X$_1$,Y$_1$)
                                   in s(U)} }
f$_2$(X,Y) = *case* Y *of* {0 $\rightarrow$ X}

Fig. 6. Transformation from *CRWL* to *FLC* syntax

- *Valid heap*: A heap Γ is valid if $[] : e \Downarrow \Gamma : v$ for some e, v.
- $ligs(\Gamma, e)$: The bindings of a valid heap Γ can be ordered in a way such that $\Gamma = [x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]$ where each e_i does not depend on x_j iff $j >= i$. That is because recursive bindings are forbidden. Then we define $ligs([x_1 \mapsto e_1, \ldots, x_n \mapsto e_n], e) =_{def} let\ \{x_1 = e_1\}\ in \ldots let\ \{x_n = e_n\}\ in\ e$.
- $[\![\Gamma, e]\!]$: Expresses the set of terms that $CRWL_{FLC}$ can reach, applying the heap Γ to the expression e. Formally, $[\![\Gamma, e]\!] =_{def} [\![ligs(\Gamma, e)]\!]_{CRWL_{FLC}} = \{t \mid ligs(\Gamma, e) \rightarrow t\}$.
- $norm(e)$: If $e^* = let\ \{x_1 = e_1\}\ in \ldots\ in\ let\ \{x_n = e_n\}\ in\ e'$, then $norm(e) = ([x_1 \mapsto e_1, \ldots, x_n \mapsto e_n], e')$. It is a kind of reverse of *ligs*.

Our main result concerning the *completeness of CRWL$_{FLC}$ with respect to FLC* is:

Theorem 6.1 *If* $\Gamma : e \Downarrow \Delta : v$, *then* $[\![\Delta, v]\!] \subseteq [\![\Gamma, e]\!]$.

Its proof becomes easy with the aid of some auxiliary results.

Lemma 6.2 *If* $[\![\Delta, x]\!] \subseteq [\![\Gamma, x]\!]$, *for all* $x \in var(e)$, *then* $[\![\Delta, e]\!] \subseteq [\![\Gamma, e]\!]$.

Theorem 6.3 *If* $\Gamma : e \Downarrow \Delta : v$, *then:*
(H) $[\![\Delta, x]\!] \subseteq [\![\Gamma, x]\!]$, *for all* $x \in dom(\Gamma)$ ***(R)*** $[\![\Delta, v]\!] \subseteq [\![\Delta, e]\!]$

The property *(H)* tells us what happens with heaps, while *(R)* relates the results of the computation. The following Corollary is an immediate consequence of Lemma 6.2 and*(H)*.

Corollary 6.4 (H') *If* $\Gamma : e \Downarrow \Delta : v$, *then* $[\![\Delta, e]\!] \subseteq [\![\Gamma, e]\!]$, *for all* e *with* $var(e) \subseteq dom(\Gamma)$.

Proof. *(Theorem 6.1)* Assume $\Gamma : e \Downarrow \Delta : v$. Then, by property *(R)* of Theorem 6.3 we have $[\![\Delta, v]\!] \subseteq [\![\Delta, e]\!]$, and by Corollary 6.4 (H') we have

$[\![\Delta, e]\!] \subseteq [\![\Gamma, e]\!]$, because it must happen that $var(e) \subseteq dom(\Gamma)$, since the FLC-derivation has succeeded. But then $[\![\Delta, v]\!] \subseteq [\![\Gamma, e]\!]$. □

Completeness of FLC with respect to CRWL$_{FLC}$ is given by the following result, whose proof is still under development:

Theorem 6.5 *If* $e \rightarrow c(t_1, \ldots, t_n)$ *and* $(\Gamma, e') = norm(e)$*, then* $\Gamma : e' \Downarrow \Delta : c(x_1, \ldots, x_n)$*, for some* $x_1, \ldots, x_n$ *verifying* $ligs(\Delta, x_i) \rightarrow t_i$ *for each* $i \in \{1, \ldots, n\}$*.*

7 Conclusions and Future Work

In this paper we study the relationship between *CRWL* [6,7] and *FLC* [1], two formal semantical descriptions of first order functional logic programming with call-time choice semantics for non-deterministic functions. The long distance between these two settings, even at syntactical level, discourages any direct proof of equivalence. Instead, we have chosen *FLC* as common language, to which *CRWL* can be adapted by means of a program transformation and a new *CRWL$_{FLC}$* proof calculus for the resulting *FLC*-programs. The program transformation itself is not very novel, although its formulation here is original, but the *CRWL$_{FLC}$* calculus and its relation to the original are indeed novel and could be useful for future works.

The most important and involved part of the paper establishes the relation between the *CRWL$_{FLC}$* logic and the natural semantics given to *FLC* in [1]. We give an equivalence result for ground expressions and for the class of *FLC*-programs not having recursive *let* bindings nor extra variables. This is not so restrictive as it could seem: it has been proved [5,4] that extra variables can be eliminated from programs, and recursive *let*'s do not appear in the translation to *FLC*-syntax of *CRWL*-programs. Still, dropping such restrictions is desirable, and we hope to do it in the next future.

We did not expect proofs to be easy. Despite of that, we are a bit surprised by the great difficulties we have encountered, even with the imposed restrictions over expressions and programs. This suggests to look for new insights, not only at the level of the proofs but also in the sense of finding new alternative semantical descriptions of functional logic programs.

References

[1] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.

[2] S. Antoy. Definitional trees. In *Proc. 13th Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632, 1992.

[3] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. ACM Symposium on Principles of Programming Languages (POPL'94)*, pages 268–279. ACM Press, 1994.

[4] S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proc. Int. Conf. on Logic Programming (ICLP'06)*. Springer LNCS, 2005.

[5] J. de Dios Castro. Eliminación de variables extra en programación lógico-funcional. Master's thesis, DSIP-UCM, May 2005.

[6] J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. European Symposium on Programming (ESOP'96)*, pages 156–172. Springer LNCS 1058, 1996.

[7] J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.

[8] M. Hanus. The integration of functions into logic programming: A survey. *Journal of Logic Programming*, 19-20:583–628, 1994. Special issue 'Ten Years of Logic Programming'.

[9] M. Hanus. Functional logic programming: From theory to Curry. Technical report, Christian-Albrechts-Universität Kiel, 2005.

[10] M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[11] M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.

[12] H. Hussmann. Non-deterministic algebraic specifications and non-confluent term rewriting. *Journal of Logic Programming*, 12:237–255, 1992.

[13] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.

[14] F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.

[15] M. Rodríguez-Artalejo. Functional and constraint logic programming. In *Revised Lectures of the International Summer School CCL'99*, pages 202–270. Springer LNCS 2002, 2001.

[16] J. Sánchez-Hernández. *Una aproximación al fallo constructivo en programación declarativa multiparadigma.* PhD thesis, DSIP-UCM, June 2004.