Relating two semantic descriptions of functional logic programs

(Work in progress)  

F.J. López-Fraguas  
J. Rodríguez-Hortalá  
J. Sánchez-Hernández

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
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 CRWLFLC, a new intermediate formal
description. Section 6 is the main part of the work and studies the relation
between CRWLFLC 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 Σ = CS ∪ 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 . . . for constructors, f, g . .
for functions and x, y . . . for variables taken from a numerable set V.

The set of expressions Exp is defined as usual: e ::= x | h(e₁, . . . , e_n),
where h ∈ CS^n ∪ FS^n and e₁, . . . , e_n ∈ 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 expres-
sions while CTerm are data terms. We will also use the extended signature
Σ⊥ = Σ ∪ {⊥}, where ⊥ is a new constant (0-arity constructor) that stands
for undefined value. Over this signature we build the sets Exp⊥ and CTerm⊥
in the natural way. The set CSubst (CSubst⊥ resp.) stands for substitutions
or mappings from V to CTerm (CTerm⊥ resp.). Both kind of substitutions
will be written as θ, σ . . . . The notation σθ denotes the composition of sub-
stitutions in the usual way. The notation 5 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
the current work, as \textit{FLC} does not consider built-in equality, we restrict the class of programs. Then a \textit{CRWL}-program \(P\) 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 \(P_f\) for the set of rules defining \(f\).

Rules of \textit{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 \(\theta\)) and then reduces the body. This logic proves approximation statements of the form \(e \rightarrow t\), where \(e \in Exp_\bot\) and \(t \in CTerm_\bot\). Given a program \(P\), the denotation of an expression \(e\) with respect to \textit{CRWL} is defined as \([e]_{CRWL}^P = \{t \mid e \rightarrow t\}\).

### 3 The \textit{FLC} Language and its Natural Semantics

The language \textit{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 \textit{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 \((f)\text{case}\) expressions. \textit{Let bindings} are a convenient way to achieve sharing.

An additional \textit{normalization step} over programs is assumed in [1]. In normalized expressions each constructor or 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 \textit{FLC}: a natural (\textit{big-step}) semantics in the style of Launchbury’s semantics [13] for lazy evaluation (with

---

**Fig. 1. Rules of CRWL**

\[
\begin{array}{c}
\text{(B)} & e \rightarrow \bot \\
\text{(RR)} & x \rightarrow x & x \in V \\
\text{(DC)} & e_1 \rightarrow t_1 \ldots e_n \rightarrow t_n \quad c(e_1, \ldots, e_n) \rightarrow c(t_1, \ldots, t_n) & c \in CS^n, \ t_i \in CTerm_\bot \\
\text{(Red)} & e_1 \rightarrow t_1 \theta \ldots e_n \rightarrow t_n \theta \ e\theta \rightarrow t & (f(t_1, \ldots, t_n) = e) \in P \\
& f(e_1, \ldots, e_n) \rightarrow t & \theta \in CSubst_\bot \\
\end{array}
\]
Programs: $P ::= D_1 \ldots D_m$
Function definitions: $D ::= f(x_1, \ldots, x_n) = e$

Expressions

| $e ::= x$ | (variable) |
| $| c(e_1, \ldots, e_n)$ | (constructor call) |
| $| f(e_1, \ldots, e_n)$ | (function call) |
| $| \text{case } e \text{ of } \{ p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n \}$ | (rigid case) |
| $| \text{fcase } e \text{ of } \{ p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n \}$ | (flexible case) |
| $| e_1 \text{ or } e_2$ | (disjunction) |
| $| \text{let } x_1 = e_1, \ldots, x_n = e_n \text{ in } e$ | (let binding) |

Patterns: $p ::= c(x_1, \ldots, x_n)$

---

**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 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) $v$. $\Gamma, \Delta$ are *heaps* consisting of bindings $x \mapsto e$ for variables. An initial configuration has the form $[] : e$.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(VarCons)</strong></td>
<td>$\Gamma[x \mapsto t] : x \Downarrow \Delta : t$ t constructor-rooted</td>
</tr>
<tr>
<td><strong>(VarExp)</strong></td>
<td>$\Gamma[x \mapsto e] : e \Downarrow \Delta : v$ $\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto v] : v$ e not constructor-rooted, $e \neq x$</td>
</tr>
<tr>
<td><strong>(Val)</strong></td>
<td>$\Gamma : v \Downarrow \Delta : v$ $v$ constructor-rooted or variable with $\Gamma[v] = v$</td>
</tr>
<tr>
<td><strong>(Fun)</strong></td>
<td>$\Gamma : e\rho \Downarrow \Delta : v$ $\Gamma : f(\overline{y}_n) \Downarrow \Delta : v$ $f(\overline{y}_n) = e \in P$ and $\rho = { y_n \mapsto x_n }$</td>
</tr>
<tr>
<td><strong>(Let)</strong></td>
<td>$\Gamma[y_k \mapsto e_k\rho] : e \Downarrow \Delta : v$ $\Gamma : \text{let } { x_k = e_k } \text{ in } e \Downarrow \Delta : v$ $\rho = { x_k \mapsto y_k }$ and $\overline{y}_k$ are fresh variables</td>
</tr>
<tr>
<td><strong>(Let)</strong></td>
<td>$\Gamma : e_1 \Downarrow \Delta : v$ $\Gamma : e_1 \text{ or } e_2 \Downarrow \Delta : v$ $i \in {1,2}$</td>
</tr>
<tr>
<td><strong>(Select)</strong></td>
<td>$\Gamma : e \Downarrow \Delta : c(\overline{y}_n)$ $\Delta : e_i\rho \Downarrow \Theta : v$ $p_i = c(x_n)$ $\Gamma : (f)\text{case } e \text{ of } { p_k \mapsto e_k } \Downarrow \Theta : v$ and $\rho = { x_n \mapsto y_n }$</td>
</tr>
</tbody>
</table>

---

**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.

![Diagram](image)

**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 \( e \rightarrow t \) where \( t \in CTerm_{\bot} \).

\[
\begin{align*}
\text{(B)} & \quad e \rightarrow \bot \\
\text{(RR)} & \quad x \rightarrow x & x \in \mathcal{V} \\
\text{(DC)} & \quad e_1 \rightarrow t_1 \ldots e_n \rightarrow t_n \quad c(e_1, \ldots, e_n) \rightarrow c(t_1, \ldots, t_n) & c \in CS^n, \ t_i \in CTerm_{\bot} \\
\text{(Red)} & \quad e^\theta \rightarrow t \qquad f(\overline{t}) \rightarrow t & (f(\overline{y}) = e) \in \mathcal{P}, \ \theta = \frac{y/t}\neq \mathcal{T} \\
\text{(Case)} & \quad e \rightarrow c(\overline{t}) \quad e_i^\theta \rightarrow t & p_i = c(\overline{x}) \text{ for some } i \\
\text{(Or)} & \quad e_i \rightarrow t \quad \text{for some } i \in \{1, 2\} \\
\text{(Let)} & \quad e' \rightarrow t' \quad e[x/t'] \rightarrow t & \text{let } \{x = e'\} \text{ in } e \rightarrow t
\end{align*}
\]

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 $P$ the denotation of an expression $e$ with respect to $CRWL_{FLC}$ is defined as $[e]_{CRWL_{FLC}} = \{ t \mid e \rightarrow t \}$.

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 $[e]_{CRWL} = [e]_{CRWL}$ 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 $P', e'$ their $FLC$-transformations. Then $[e]_{CRWL} = [e']_{CRWL_{FLC}}$.

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

We need some more technical preliminary notions:

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

Fig. 6. Transformation from CRWL to FLC syntax

- **Valid heap**: A heap $\Gamma$ is valid if $[] : e \Downarrow \Gamma : v$ for some $e, v$.

- **$\text{ligs}(\Gamma, e)$**: The bindings of a valid heap $\Gamma$ 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 $\text{ligs}(\Gamma, e) = \text{def} \{ x_1 = e_1 \} \ldots \{ x_n = e_n \}$ in $e$.

- **$[\Gamma, e]$**: Expresses the set of terms that CRWL can reach, applying the heap $\Gamma$ to the expression $e$. Formally, $[\Gamma, e] = \text{def} \{ t | \text{ligs}(\Gamma, e) \rightarrow t \}$.

- **$\text{norm}(e)$**: If $e^* = \text{let} \{ x_1 = e_1 \} \ldots \text{in} \{ x_n = e_n \}$ in $e'$, then $\text{norm}(e) = ([x_1 \mapsto e_1, \ldots, x_n \mapsto e_n], e')$. It is a kind of reverse of $\text{ligs}$.

Our main result concerning the completeness of CRWL 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 $x \in \text{var}(e)$, then $[\Delta, e] \subseteq [\Gamma, e]$.

**Theorem 6.3** If $\Gamma : e \Downarrow \Delta : v$, then:

- **(H)** $[\Delta, x] \subseteq [\Gamma, x]$, for all $x \in \text{dom}(\Gamma)$
- **(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 $\text{var}(e) \subseteq \text{dom}(\Gamma)$.

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

Completeness of FLC with respect to CRWL

Theorem 6.5 If \( e \rightarrow c(t_1, \ldots, t_n) \) and \( (\Gamma, e') = \text{norm}(e) \), then \( \Delta : c(x_1, \ldots, x_n) \downarrow \) \( \Delta : c(x_1, \ldots, x_n) \), for some \( x_1, \ldots, x_n \) verifying \( \text{tigs}(\Delta, x_i) \rightarrow t_i \) 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\textsubscript{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\textsubscript{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\textsubscript{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 \textit{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 \textit{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


