A Simple Rewrite Notion for Call-time Choice Semantics [*]

Francisco J. López-Fraguas Juan Rodríguez-Hortalá Jaime Sánchez-Hernández

Dep. Sistemas Informáticos y Computación, Universidad Complutense de Madrid

fraguas@sip.ucm.es jrodrigu@fdi.ucm.es jaime@sip.ucm.es

Abstract

Non-confluent and non-terminating rewrite systems are interesting from the point of view of programming. In particular, existing functional logic languages use such kind of rewrite systems to define possibly non-strict non-deterministic functions. The semantics adopted for non-determinism is call-time choice, whose combination with non-strictness is not a trivial issue that has been addressed from a semantic point of view in the Constructor-based Rewriting Logic (*CRWL*) framework. We investigate here how to express call-time choice and non-strict semantics from a point of view closer to classical rewriting. The proposed notion of rewriting uses an explicit representation for sharing with *let*-constructions and is proved to be equivalent to the *CRWL* approach. Moreover, we relate this *let*-rewriting relation (and hence *CRWL*) with ordinary rewriting, providing in particular soundness and completeness of *let*-rewriting with respect to rewriting for a class of programs which are confluent in a certain semantic sense.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics

General Terms Theory, languages.

Keywords Functional-logic programming, term rewriting systems, constructor-based rewriting logic, non-determinism, call-time choice semantics, sharing, local bindings.

1. Introduction

Modern functional logic programs as considered in systems like *Curry* [12] or *Toy* [16] are constructor-based term rewrite systems, possibly non-terminating and non-confluent, thus defining possibly non-strict non-deterministic functions, as happens with the program in Figure 1.

The semantics adopted for non-determinism in those systems is *call-time choice* semantics [10, 13], also called sometimes *singular* semantics [23]. Loosely speaking, call-time choice conceptually means to pick a value for each argument of a function application before applying it. Call-time choice is easier to understand and implement in combination with strict semantics and eager evaluation in terminating systems as in [13], but can be made also compatible –via partial values and sharing– with non-strictness and laziness in the presence of non-termination.

In the example of Figure 1 the expression $heads(repeat(coin))$ can take, under call-time choice, the values $(0, 0)$ and $(1, 1)$, but not $(0, 1)$ or $(1, 0)$. The example illustrates also a key point here, that ordinary term rewriting is an unsound procedure for call-time choice semantics with non-determinism, since a possible rewrite is

$$heads(repeat(coin)) \to heads(coin : repeat(coin)) \to$$
$$heads(0 : repeat(coin)) \to heads(0{:}coin{:}repeat(coin)) \to$$
$$heads(0 : 1 : repeat(coin)) \to (0, 1)$$

In operational terms, call-time choice would have required to share the value for all the occurrences of $coin$ in the reduction above.

It is commonly accepted (see e.g. [11]) that call-time choice semantics combined with non-strict semantics is adequately formally expressed by the *CRWL* framework [9, 10]. An additional indication of the usefulness of *CRWL* is the large set of its extensions that have been devised to cope with relevant aspects of declarative programming: higher order functions, types, constraints, constructive failure, . . . (see [21] for a survey of the first works on the *CRWL* approach). However, a drawback of the *CRWL*-logic is its lack of a proper one-step reduction mechanism close both to the logic and to the computations, that could play a role similar to rewriting with respect to equational logic. Certainly *CRWL* includes operational procedures in the form of lazy narrowing based goal-solving calculi [10, 24], but they are too complex to be seen as the basic or 'fundamental' way to explain or understand how reduction can proceed in the presence of non-strict non-deterministic functions with call-time choice semantics.

Therefore, other works have been more influential on the operational side of the field, specially those based on the notion of needed narrowing [4], whose underlying theory is classical rewriting. Needed narrowing has become the 'official' operational procedure of functional logic languages, and has also been subject of various variations and improvements (see [11]).

These two coexisting branches of research (one based on *CRWL*, and the other based on classical rewriting via needed narrowing) have remained disconnected from the technical point of view, despite the fact that they both refer to what intuitively is the same programming language paradigm, as believed by most –if not all– people in the field.

This is not a satisfactory situation, because it precludes the possibility of applying –on a sound technical basis– results, notions and techniques from the semantic side to the operational side and viceversa. Our aim in this work is to establish that missing bridge.

A major problem is that needed narrowing adopts classical rewriting as underlying theory and therefore is not valid for call-time choice with non-determinism. This is overcome in practice by adding a sharing mechanism to the encoding of narrowing, but this is an implementation patch that is not enough for our technical purposes. Is there an existing notion of rewriting that can be used

[*] This work has been partially supported by the Spanish projects TIN2005-09207-C03-03 (MERIT-FORMS-UCM) and S-0505/TIC/0407 (PROMESAS-CAM).

$$\begin{array}{|ll|}\hline coin \rightarrow 0 & repeat(X) \;\rightarrow\; X\!:\!repeat(X) \\ coin \rightarrow 1 & heads(X\!:\!Y\!:\!Ys) \;\rightarrow\; (X,Y) \\ \hline \end{array}$$

Figure 1. A non-terminating and non-confluent program

instead? Of course, the issue of combining sharing with rewriting or other reductions notions is not new. But a review of the literature (in Section 7 we make a short summary) suggested to us that there was still room for proposing a new formulation of rewriting tailored to call-time choice as realized by functional logic languages, and trying to fulfil the following requirements:

• it should be based on a notion of rewrite step, as to be useful to follow how a computation proceeds step by step.

• it should be simple enough to be easily understandable for non-expert potential users. (e.g., students) of functional logic languages adopting call-time choice.

• it should be provably equivalent to *CRWL*.

• it should serve as a basis of subsequent notions of narrowing and narrowing strategies.

We propose then a simple variant of rewriting that uses local bindings in the form of *let*-expressions to express sharing. Not surprisingly, our *let*-rewriting is very close to existing formalisms to express sharing in different contexts, like in [18] for λ-calculus, or term graph rewriting [20]. We are also inspired by [17] where indexed unions of set expressions – a construction generalizing the idea of *let*-expressions – were used to express sharing in an extension of *CRWL* to deal with constructive failure.

We also investigate the connection between our *let*-rewriting relation and classical rewriting. As we will prove, in general *let*-rewriting is sound with respect to rewriting, and is also complete for confluent systems (more precisely, for deterministic programs, a semantic property close to confluence).

The rest of the paper is organized as follows. Section 2 presents some preliminaries about term rewriting and the *CRWL* framework. Section 3 contains a first discussion about how to express non-strict call-time choice by rewriting. Section 4 introduces local bindings in syntax to express sharing and defines *let*-rewriting as an adequate notion of rewriting for them. In Section 5 we prove the equivalence of *CRWL* and *let*-rewriting. In Section 6 we address the relationship between of *let*-rewriting and classical rewriting, proving in particular their equivalence for deterministic programs. Finally, Section 7 reviews related work and summarizes some conclusions. Some of the proofs have been moved to an appendix and some other are simply sketched or even completely omitted. Full proofs can be found at `http://gpd.sip.ucm.es/juanrh/pubs/ppdp2007/long.pdf`.

2. Preliminaries

2.1 Constructor based term rewriting systems

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

To avoid confusion with the usual terminology of *CRWL* (introduced below) we follow its approximation introducing two kinds of syntactic objects: expressions and terms. The set Exp of *expressions* is defined as $Exp \ni e ::= x \mid h(e_1, \ldots, e_n)$, where $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. The set $CTerm$ of *constructed terms* (or *c-terms*) has the same definition of Exp, but with h restricted to CS^n (so $CTerm \subseteq Exp$). The intended meaning is that Exp stands for evaluable expressions, i.e., expressions that can contain (user-defined) function symbols, while $CTerm$ stands for data terms representing values. We will write $e, e', \ldots$ for expressions and $t, s, t', s' \ldots$ for c-terms. The set of variables occurring in an expression e will be denoted as $var(e)$.

Contexts (with one hole) are defined by $Cntxt \ni \mathcal{C} ::= [\,] \mid h(e_1, \ldots, \mathcal{C}, \ldots, e_n)$, where $h \in CS^n \cup FS^n$. The application of a context $\mathcal{C}$ to an expression e, written as $\mathcal{C}[e]$, is defined inductively by $[\,][e] = e$; $h(e_1, \ldots, \mathcal{C}, \ldots, e_n)[e] = h(e_1, \ldots, \mathcal{C}[e], \ldots, e_n)$.

Substitutions are mappings $\theta : \mathcal{V} \longrightarrow Exp$ which extend naturally to $\theta : Exp \longrightarrow Exp$. We write $e\theta$ for the application of θ to e. The domain and range of θ are defined as $dom(\theta) = \{x \in \mathcal{V} \mid x\theta \neq x\}$ and $ran(\theta) = \bigcup_{x \in dom(\theta)} var(x\theta)$. Given a set of variables D the notation $\theta|_D$ represents the substitution θ restricted to D and $\theta|_{\backslash D}$ is a shortcut for $\theta|_{(\mathcal{V} \backslash D)}$. A *c-substitution* is a substitution θ such that $x\theta \in CTerm$ for all $x \in dom(\theta)$. We write $Subst$ and $CSubst$ for the sets of substitutions and c-substitutions. Throughout the paper, the notation $\overline{o}$ stands for tuples of any of the previous syntactic construction o.

A *constructor based rewrite rule* (or c-rewrite rule) has the form $f(\overline{t}) \rightarrow e$ where $f \in FS^n$, $e \in Exp$ and $\overline{t}$ is a linear tuple of c-terms, where linear means that no variable occurs twice in the tuple. Notice that we allow e to have extra variables (i.e., variables not occurring in the left-hand side). A constructor-based rewrite system (or *c-rewrite system*) is a set of c-rewrite rules. Given a c-rewrite system $\mathcal{P}$, its rewrite relation $\rightarrow_{\mathcal{P}}$ is defined by $\mathcal{C}[l\theta] \rightarrow_{\mathcal{P}} \mathcal{C}[r\theta]$, for any context $\mathcal{C}$, rule $l \rightarrow r \in \mathcal{P}$ and substitution θ. We write $\rightarrow^*_{\mathcal{P}}$ for the reflexive and transitive closure of the relation $\rightarrow_{\mathcal{P}}$. Since in this paper we only consider constructor based rules, we will often speak simply of rewrite rules or rewrite systems. Furthermore, we will usually omit the reference to $\mathcal{P}$ in $\rightarrow_{\mathcal{P}}$.

Confluence for constructor-based term rewrite systems is defined in the usual way: a program $\mathcal{P}$ is confluent if for any $e, e_1, e_2 \in Exp$ such that $e \rightarrow^*_{\mathcal{P}} e_1$, $e \rightarrow^*_{\mathcal{P}} e_2$ there exist $e_3 \in Exp$ such that both $e_1 \rightarrow^*_{\mathcal{P}} e_3$ and $e_2 \rightarrow^*_{\mathcal{P}} e_3$.

2.2 The CRWL framework

In the *CRWL* framework [9, 10], programs are c-rewrite systems, also called *CRWL-programs* (or simply 'programs') from now on. The original *CRWL* logic considered also the possible presence of *joinability* constraints as conditions in rules in order to give a better treatment of strict equality as built-in, which is a subject orthogonal to the aims of this paper. Furthermore, due to the semantic given to equality in functional logic and thanks to the allowance of extra variables in rules, it is possible to replace conditions by the use of an *if_then* function, as has been technically proved in [22] for *CRWL* and in [2] for term rewriting. Therefore, we consider only unconditional rules.

To deal with non-strictness at the semantic level, we enlarge Σ with a new constant constructor symbol $\perp$. The sets $Exp_\perp, CTerm_\perp, Subst_\perp, CSubst_\perp$ of partial expressions, etc., are defined naturally. Notice that $\perp$ does not appear in programs. Partial expressions are ordered by the *approximation* ordering $\sqsubseteq$ defined as the least partial ordering satisfying $\perp \sqsubseteq e$ and $e \sqsubseteq e' \Rightarrow \mathcal{C}[e] \sqsubseteq \mathcal{C}[e']$ for all $e, e' \in Exp_\perp, \mathcal{C} \in Cntxt$. This partial ordering can be extended to substitutions: given $\theta, \sigma \in Subst_\perp$ we say $\theta \sqsubseteq \sigma$ if $X\theta \sqsubseteq X\sigma$ for all $X \in \mathcal{V}$.

The semantics of a program $\mathcal{P}$ is determined in *CRWL* by means of a proof calculus able to derive reduction statements of the form

$e \rightarrowtriangle t$, with $e \in Exp_\perp$ and $t \in CTerm_\perp$, meaning informally that t is (or approximates to) a possible value of e, obtained by iterated reduction of e using $\mathcal{P}$ under call-time choice.

The *CRWL*-proof calculus is presented in Figure 2. Rule **(B)** allows any expression to be undefined or not evaluated (non-strict semantics). Rule **(OR)** expresses that to evaluate a function call we must choose a compatible program rule, perform parameter passing (by means of a c-substitution θ) and then reduce the right-hand side. The use of c-substitutions in **(OR)** is essential to express call-time choice; notice also that by the effect of θ in **(OR)**, extra variables in the right-hand side of a rule can be replaced by any c-term, but not by any expression as in the notion of ordinary rewriting $\rightarrow_\mathcal{P}$.

We write $\mathcal{P} \vdash_{CRWL} e \rightarrowtriangle t$ to express that $e \rightarrowtriangle t$ is derivable in the *CRWL*-calculus using the program $\mathcal{P}$. Given a program $\mathcal{P}$, the *CRWL-denotation* of an expression $e \in Exp_\perp$ is defined as $[\![e]\!]^\mathcal{P}_{CRWL} = \{t \in CTerm_\perp \mid \mathcal{P} \vdash_{CRWL} e \rightarrowtriangle t\}$.

(B) $\dfrac{}{e \rightarrowtriangle \perp}$ **(RR)** $\dfrac{}{x \rightarrowtriangle x}$ $x \in \mathcal{V}$

(DC) $\dfrac{e_1 \rightarrowtriangle t_1 \ \ldots \ e_n \rightarrowtriangle t_n}{c(e_1,\ldots,e_n) \rightarrowtriangle c(t_1,\ldots,t_n)}$ $c \in CS^n, t_i \in CTerm_\perp$

(OR) $\dfrac{e_1 \rightarrowtriangle t_1\theta \ \ldots \ e_n \rightarrowtriangle t_n\theta \quad e\theta \rightarrowtriangle t}{f(e_1,\ldots,e_n) \rightarrowtriangle t}$ $f(\bar{t}) \rightarrow e \in \mathcal{P}$, $\theta \in CSubst_\perp$

Figure 2. Rules of *CRWL*

As an example, Figure 3 shows a *CRWL*-derivation for the statement $heads(repeat(coin)) \rightarrowtriangle (0,0)$, using the program of Figure 1. Observe that in the derivation there is only one reduction statement for *coin* (namely *coin* $\rightarrowtriangle$ 0), and the obtained value 0 is then *shared* in the whole derivation, as corresponds to call-time choice. In alternative derivations, *coin* could be reduced to 1 (or to $\perp$). It is easy to see that $[\![heads(repeat(coin))]\!]^\mathcal{P}_{CRWL} = \{(0,0),(1,1),(\perp,0),(0,\perp),(\perp,1),(1,\perp),(\perp,\perp),\perp\}$.

Note that $(1,0),(0,1) \notin [\![heads(repeat(coin))]\!]^\mathcal{P}_{CRWL}$.

The following monotonicity lemma is a classical result in the *CRWL* framework [9, 10]:

LEMMA 1. *Given a program $\mathcal{P}$, $e \in Exp_\perp$, $t \in CTerm_\perp$ and $\theta, \theta' \in CSubst_\perp$ with $\theta \sqsubseteq \theta'$ then we have:*

$$\text{if } \mathcal{P} \vdash_{CRWL} e\theta \rightarrowtriangle t \text{ then } \mathcal{P} \vdash_{CRWL} e\theta' \rightarrowtriangle t$$

We stress the fact that the *CRWL*-calculus *is not* an operational mechanism for executing programs, but a way of describing the logic of programs. At the operational level the *CRWL* framework comes with various lazy narrowing-based goal-solving calculi [10, 24] not considered in this paper.

3. CRWL and rewriting: a first discussion

Our general concern is how to express non-strict call-time choice semantics by means of a simple rewriting-like one-step reduction relation. We started Section 1 by observing that ordinary term rewriting is not valid for that purpose. Now, we discard also the possibility of transforming the original system into another one such that using (ordinary) term rewriting it behaves as the original one under call-time choice. More precisely, we pose the following question:

For any given c-rewrite system $\mathcal{P}$, can we find another rewrite system (constructor based or not) $\mathcal{P}'$ such that for each expression e and constructed term t, (which can be ground or not) $\mathcal{P} \vdash_{CRWL} e \rightarrowtriangle t$ iff $e \rightarrow^*_{\mathcal{P}'} t$?

The answer to it is *'no'*, as the following simple example shows, exploiting the fact that rewriting is closed under substitutions while *CRWL*-provability is only closed under c-substitutions.

EXAMPLE 1. *Consider the rewrite system $\mathcal{P}$:*

$$f(X) \rightarrow c(X,X) \qquad coin \rightarrow 0 \qquad coin \rightarrow 1$$

and assume a system $\mathcal{P}'$ such that: $\mathcal{P} \vdash_{CRWL} e \rightarrowtriangle t \Leftrightarrow e \rightarrow^*_{\mathcal{P}'} t$, for all e, t. We will arrive to a contradiction.

Since $\mathcal{P} \vdash_{CRWL} f(X) \rightarrowtriangle c(X,X)$, we must have $f(X) \rightarrow^*_{\mathcal{P}'} c(X,X)$. Now, since $\rightarrow^*_{\mathcal{P}'}$ is closed under substitutions, we have $f(coin) \rightarrow^*_{\mathcal{P}'} c(coin,coin)$, and then we have the reductions $f(coin) \rightarrow^*_{\mathcal{P}'} c(coin,coin) \rightarrow^*_{\mathcal{P}'} c(0,1)$. But it is easy to see that $\mathcal{P} \vdash_{CRWL} f(coin) \rightarrowtriangle c(0,1)$ does not hold.

Another possibility is to impose the restriction that the substitution θ in a rewriting step must be a c-substitution, as it is done in the rule **(OR)** of *CRWL*. More precisely, we can define rewriting by the rule **(OR')** in Figure 4 below. With it the step $heads(repeat(coin)) \rightarrow heads(coin : repeat(coin))$ in the example of Figure 1 would not be legal anymore. This simple solution would be enough to deal with call-time choice and a *strict* semantics, but it is not sufficient for non-strictness, as shown by the following simple example:

EXAMPLE 2. *Consider the rewrite system given by the two rules $f(X) \rightarrow 0$ and $loop \rightarrow loop$. With a non-strict semantics $f(loop)$ should be reducible to 0. But with* **(OR')** *$f(loop) \rightarrow 0$ is not permitted; the only rewriting sequence starting with $f(loop)$ is $f(loop) \rightarrow f(loop) \rightarrow \ldots$, thus leaving $f(loop)$ semantically undefined, as would correspond to a strict semantics.*

What is missing is a rule allowing to reduce a not-needed (sub)-expression to a special constructor term with no information in it. Since not-neededness is undecidable, this special reduction must be allowed for any expression. This is given precisely by the rule **(B)** of *CRWL*, which is indeed a one-step rule. The result of this discussion is the one-step reduction relation $\rightarrowtail$ given in Figure 4.

It is not difficult to prove the following equivalence result:

THEOREM 1. *Let $\mathcal{P}$ be a CRWL-program, $e \in Exp_\perp, t \in CTerm_\perp$. Then $\mathcal{P} \vdash_{CRWL} e \rightarrowtriangle t$ iff $e \rightarrowtail^*_\mathcal{P} t$.*

PROOF: It is easy to see that $\rightarrowtail^*$ (the reflexive and transitive closure of $\rightarrowtail$) coincides with the derivability relation defined by the proof calculus called *BRC* in [10]. This means that $\mathcal{P} \vdash_{BRC} e \rightarrowtriangle e'$ *iff* $e \rightarrowtail^* e'$. But in that paper it is proved that, for $e \in Exp_\perp$ and $t \in CTerm_\perp$, *BRC*-derivability and *CRWL*-derivability (called there *GORC*-derivability) are equivalent, what implies:
$\mathcal{P} \vdash_{CRWL} e \rightarrowtriangle t \;\Leftrightarrow\; \mathcal{P} \vdash_{BRC} e \rightarrowtriangle t \;\Leftrightarrow\; e \rightarrowtail^* t.$ $\square$

We remark that **(OR')** essentially corresponds to innermost evaluation. So the result has the following interesting reading: non-strict call-time choice can be achieved via innermost evaluation if at any step one has the possibility of reducing a subexpression to $\perp$. For instance, a $\rightarrowtail$-rewrite sequence with the example of Figure 1 would be:

$$heads(repeat(coin)) \rightarrowtail heads(repeat(0)) \rightarrowtail$$
$$heads(0 : repeat(0)) \rightarrowtail heads(0 : 0 : repeat(0)) \rightarrowtail$$
$$heads(0 : 0 : \perp) \rightarrowtail (0,0)$$

The rules for $\rightarrowtail$ can actually serve for a very easy implementation of non-strict call-time choice, but with a major drawback: reduction follows an unnatural order and requires, at any step, an unavoidable guessing between the two rules **(B')** and **(OR')**, leading to high inefficiency. Therefore, $\rightarrowtail$ achieves only partially our goals and

$$
\dfrac{\dfrac{}{0 \rightarrowtriangle 0}\,DC}{\mathbf{coin} \rightarrowtriangle \mathbf{0}}\,OR
\quad
\dfrac{
\dfrac{\dfrac{}{0 \rightarrowtriangle 0}\,DC \quad \dfrac{\dfrac{\dfrac{}{0 \rightarrowtriangle 0}\,DC \quad \dfrac{\dfrac{\dfrac{}{0 \rightarrowtriangle 0}\,DC \quad \dfrac{}{repeat(0) \rightarrowtriangle \bot}\,B}{0 : repeat(0) \rightarrowtriangle 0 : \bot}\,DC}{repeat(0) \rightarrowtriangle 0 : \bot}\,OR}{0 : repeat(0) \rightarrowtriangle 0 : 0 : \bot}\,DC}{repeat(coin) \rightarrowtriangle 0 : 0 : \bot}\,OR
\quad
\dfrac{\dfrac{}{0 \rightarrowtriangle 0}\,DC \quad \dfrac{}{0 \rightarrowtriangle 0}\,DC}{(0,0) \rightarrowtriangle (0,0)}\,DC
}{heads(repeat(coin)) \rightarrowtriangle (0,0)}\,OR
$$

Figure 3. A *CRWL*-derivation

(B')	$\mathcal{C}[e]$	$\rightarrowtail$	$\mathcal{C}[\bot]$	for any $\mathcal{C} \in Cntxt, e \in Exp_\bot$
(OR')	$\mathcal{C}[f(t_1\theta, \ldots, t_n\theta)]$	$\rightarrowtail$	$\mathcal{C}[e\theta]$	for any $\mathcal{C} \in Cntxt,\ \ f(t_1, \ldots, t_n) \to e \in \mathcal{P}$, $\theta \in CSubst_\bot$

Figure 4. A one-step reduction relation for non-strict call-time choice

we cannot consider it as the natural reduction notion we are looking for.

4. Rewriting with local bindings

In this section we introduce local bindings in the form of *let*-expressions as a convenient way of expressing sharing. Formally the syntax for *let-expressions* is:

$$LExp \ni e ::= X \mid h(\overline{e}) \mid let\ X = e_1\ in\ e_2$$

where $X \in \mathcal{V}$, $h \in CS \cup FS$, $\overline{e}$ is a tuple of *let*-expressions, and e_1, e_2 are single *let*-expressions. We will use the notation $let\ \overline{X = a}\ in\ e$ as a shortcut for $let\ X_1 = a_1\ in \ldots in\ let\ X_n = a_n\ in\ e$. The notion of *one-hole context* is also extended to the new syntax:

$$\mathcal{C} ::= [\,] \mid let\ X = \mathcal{C}\ in\ e \mid let\ X = e\ in\ \mathcal{C} \mid h(\ldots, \mathcal{C}, \ldots)$$

The sets $FV(e)$ of *free* and $BV(e)$ *bound* variables of $e \in LExp$ are defined as:

$$FV(X) = \{X\};\ FV(h(\overline{e})) = \bigcup_{e_i \in \overline{e}} FV(e_i);$$
$$FV(let\ X = e_1\ in\ e_2) = FV(e_1) \cup (FV(e_2) \backslash \{X\});$$
$$BV(X) = \emptyset;\ BV(h(\overline{e})) = \bigcup_{e_i \in \overline{e}} BV(e_i);$$
$$BV(let\ X = e_1\ in\ e_2) = BV(e_1) \cup BV(e_2) \cup \{X\}$$

Notice that with the given definition of $FV(let\ X = e_1\ in\ e_2)$ there are not recursive *let*-bindings in the language since the possible occurrences of X in e_1 are not considered bound and therefore refer to a 'different' X. This is similar to what is done in [18], but not in [1, 14]. Recursive *let*s have their own interest but since they are not present in *CRWL*-programs (there are no *let*s at all in *CRWL*) and will neither appear in a *let*-rewriting reduction (to be defined below) unless they are already present in the c-rewrite system, we have decided not to consider them. Furthermore, there is not a general consensus about the reading of recursive *let*s in presence of non-determinism.

Notice that the notion of c-term has not changed with the introduction of *let*s: in particular c-terms do not contain *let*s, but can contain bound variables, as happens for example with (X, X) in the *let*-expression $let\ X = coin\ in\ (X, X)$.

As usual with syntactical binding constructs, we assume a variable convention according to what bound variables can be consistently renamed as to ensure that the same variable symbol does not occur free and bound within an expression. Moreover, to keep simple the management of substitutions, we assume that whenever θ is applied to an expression $e \in LExp$, the necessary renamings are done in e to ensure that $BV(e) \cap (dom(\theta) \cup ran(\theta)) = \emptyset$. With all these conditions the rules defining application of substitutions are simple while avoiding variable capture:

$$X\theta = \theta(X)$$
$$h(e_1, \ldots, e_n)\theta = h(e_1\theta, \ldots, e_n\theta)$$
$$(let\ X = e_1\ in\ e_2)\theta = let\ X = e_1\theta\ in\ e_2\theta$$

The *let*-rewriting relation $\to_l$ is shown in Figure 5. The rule **(Fapp)** performs a rewriting step in a proper sense, using a rule of the program. Note that only c-substitutions are allowed, to avoid copying of unevaluated expressions which would destroy sharing and call-time choice. **(Contx)** allows to select any subexpression as a redex for the derivation. The rest of the rules are syntactic manipulations of *let*-expressions. In particular **(LetIn)** transforms standard expressions by introducing a *let*-binding to express sharing. On the other hand, **(Bind)** removes a *let*-construction for a variable when its binding expression has been evaluated. **(Elim)** allows to remove a binding when the variable does not appear in the body of the construction, which means that the corresponding value is not needed for evaluation. This rule is needed because the expected normal forms are c-terms not containing *let*s. **(Flat)** is needed for flattening nested *let*s, otherwise some reductions could become wrongly blocked or forced to diverge. For example, with the rewrite rules $loop \to loop$ and $g(s(X)) \to 1$ and applying twice **(LetIn)** to the expression $g(s(loop))$, we obtain $let\ X = (let\ Y = loop\ in\ s(Y))\ in\ g(X)$. Without **(Flat)** we can only perform reductions on $loop$; with **(Flat)** we obtain $let\ Y = loop\ in\ let\ X = s(Y)\ in\ g(X)$ and then applying **(Bind)** and **(Elim)** we achieve the expected value 1. Notice that with the variable convention, the condition $Y \notin FV(e_3)$ in **(Flat)** would not be needed. We have written it in order to keep the rules independent of the convention. Quite different is the case of **(Elim)**, where the condition $X \notin FV(e_2)$ might hold or not.

As a complete derivation example, consider the program of Figure 1 and the derivation of Figure 6. Notice that there is not a unique $\to_l$-reduction leading to $(0, 0)$. The definition of $\to_l$ does not prescribe any particular strategy, a subject that has been left out of the scope of this paper.

5. Equivalence of let-rewriting and CRWL

In this section we will prove the soundness and completeness results of *let*-rewriting with respect to *CRWL*. To this purpose we will need to consider $\bot$ at some points. Therefore we define the set $LExp_\bot$ in the natural way. We also define the *shell* $|e|$ *of an expression* e that represents the outer constructor part of e, and is

(Contx)	$\mathcal{C}[e] \to_l \mathcal{C}[e']$, if $e \to_l e', \mathcal{C} \in Cntxt$
(LetIn)	$h(\ldots, e, \ldots) \to_l let\ X = e\ in\ h(\ldots, X, \ldots)$ if $h \in CS \cup FS$, e takes one of the forms $e \equiv f(\overline{e'})$ with $f \in FS$ or $e \equiv let\ Y = e'\ in\ e''$, and X is a fresh variable
(Flat)	$let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \to_l let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$ assuming that Y does not appear free in e_3
(Bind)	$let\ X = t\ in\ e \to_l e[X/t]$, if $t \in CTerm$
(Elim)	$let\ X = e_1\ in\ e_2 \to_l e_2$, if X does not appear free in e_2
(Fapp)	$f(t_1\theta, \ldots, t_n\theta) \to_l e\theta$, if $f(t_1, \ldots, t_n) \to e \in \mathcal{P}, \theta \in CSubst$

Figure 5. Rules of *let*-rewriting

$heads(repeat(coin)) \to_l$	$(\mathbf{LetIn})$
$let\ X = repeat(coin)\ in\ heads(X) \to_l$	$(\mathbf{LetIn})$
$let\ X = (let\ Y = coin\ in\ repeat(Y))\ in\ heads(X) \to_l$	$(\mathbf{Flat})$
$let\ Y = coin\ in\ let\ X = repeat(Y)\ in\ heads(X) \to_l$	$(\mathbf{Fapp})$
$let\ Y = 0\ in\ let\ X = repeat(Y)\ in\ heads(X) \to_l$	$(\mathbf{Bind})$
$let\ X = repeat(0)\ in\ heads(X) \to_l$	$(\mathbf{Fapp})$
$let\ X = 0 : repeat(0)\ in\ heads(X) \to_l$	$(\mathbf{LetIn})$
$let\ X = (let\ Z = repeat(0)\ in\ 0 : Z)\ in\ heads(X) \to_l$	$(\mathbf{Flat})$
$let\ Z = repeat(0)\ in\ let\ X = 0 : Z\ in\ heads(X) \to_l$	$(\mathbf{Fapp})$
$let\ Z = 0 : repeat(0)\ in\ let\ X = 0 : Z\ in\ heads(X) \to_l$	$(\mathbf{LetIn}, \mathbf{Flat})$
$let\ U = repeat(0)\ in\ let\ Z = 0 : U\ in\ let\ X = 0 : Z\ in\ heads(X) \to_l$	$(\mathbf{Bind}), \mathbf{2}$
$let\ U = repeat(0)\ in\ heads(0 : 0 : U) \to_l$	$(\mathbf{Fapp})$
$let\ U = repeat(0)\ in\ (0, 0) \to_l$	$(\mathbf{Elim})$
$(0, 0)$	

Figure 6. A let-rewriting derivation

defined as follows:

$$\begin{array}{lcl} |X| & = & X \\ |c(e_1, \ldots, e_n)| & = & c(|e_1|, \ldots, |e_n|) \\ |f(e_1, \ldots, e_n)| & = & \bot \\ |let\ X = e_1\ in\ e_2| & = & |e_2|[X/|e_1|] \end{array}$$

Notice that the information contained in *let*-bindings is taken into account for building up the shell of an expression.

5.1 Soundness

Concerning soundness we would like to prove something like this:

If $e \to_l e'$ *then* $\llbracket e' \rrbracket_{CRWL} \subseteq \llbracket e \rrbracket_{CRWL}$, for any $e, e' \in Exp$.

That is, $\to_l$-steps do not create new *CRWL*-semantic values. But *let*-expressions are not defined in *CRWL* and even if we start with an expression without *let*s, *let*-rewriting may introduce them by **(LetIn)**. To cope with this situation we enlarge the *CRWL*-calculus in Figure 2 to a new calculus *CRWL$_{let}$*, by adding a new rule for dealing with *let*-expressions:

$$\textbf{(Let)}\ \frac{e_1 \rightarrowtriangle t_1 \quad e[X/t_1] \rightarrowtriangle t}{let\ X = e_1\ in\ e \rightarrowtriangle t}$$

We write $\mathcal{P} \vdash_{CRWL_{let}} e \rightarrowtriangle t$ if $e \rightarrowtriangle t$ is derivable in the $CRWL_{let}$ calculus using the program $\mathcal{P}$. The $CRWL_{let}$-denotation of an expression $e \in LExp_\bot$ with respect to the program $\mathcal{P}$ is defined as

$$\llbracket e \rrbracket^{\mathcal{P}}_{CRWL_{let}} = \{t \in CTerm_\bot | \mathcal{P} \vdash_{CRWL_{let}} e \rightarrowtriangle t\}$$

We will omit the sub(super)-scripts when they are clear by the context.

CRWL$_{let}$ shares with *CRWL* the property of closedness under c-substitutions. The following result states this fact, together with some other useful properties related to shells that are not difficult to check by the appropriate induction in each case:

LEMMA 2. *Let $\mathcal{P}$ be a CRWL-program and $e \in LExp_\bot$. Then:*
(i) $\mathcal{P} \vdash_{CRWL_{let}} e \rightarrowtriangle t$ implies $\mathcal{P} \vdash_{CRWL_{let}} e\sigma \rightarrowtriangle t\sigma$, for any $t \in CTerm_\bot, \sigma \in CSubst_\bot$.
(ii) $|e| \in \llbracket e \rrbracket_{CRWL_{let}}$.
(iii) $\llbracket e \rrbracket_{CRWL_{let}} \subseteq |e| \uparrow$, where the upward closure $t \uparrow$ of $t \in CTerm_\bot$ is $t \uparrow = \{s \in CTerm_\bot | t \sqsubseteq s\}$.
(iv) $e \to_l e'$ implies $|e| \sqsubseteq |e'|$.

Parts *(ii)* to *(iv)* express that the shell of an expression represents 'stable' information contained in the expression (*(ii)* says that shells are in the denotation; *(iii)*, that everything in the denotation comes from refining it, and *(iv)* says that shells grow monotonically with reduction).

It is easy to establish the equivalence between *CRWL* and *CRWL$_{let}$* for expressions not involving *let*s.

LEMMA 3. *For any CRWL-program $\mathcal{P}$, $e \in Exp_\bot$ and $t \in CTerm_\bot$, we have: $\mathcal{P} \vdash_{CRWL} e \rightarrowtriangle t$ iff $\mathcal{P} \vdash_{CRWL_{let}} e \rightarrowtriangle t$. Therefore $\llbracket e \rrbracket^{\mathcal{P}}_{CRWL} = \llbracket e \rrbracket^{\mathcal{P}}_{CRWL_{let}}$.*

With the aid of *CRWL$_{let}$*, the theorem we are looking for can be stated as follows:

THEOREM 2 (One-Step Soundness of *let*-rewriting).
For any $e, e' \in LExp$,

$$e \to_l e' \textit{ implies } \llbracket e' \rrbracket_{CRWL_{let}} \subseteq \llbracket e \rrbracket_{CRWL_{let}}.$$

Notice that because of non-determinism $\subseteq$ cannot be replaced by $=$ in this theorem. The proof of Theorem 2 (which is given below) would proceed straightforwardly by a case distinction on the rules for $\to_l$, if the following *monotonocity under contexts* was true for any context $\mathcal{C}$:

$$[\![e]\!]_{CRWL_{let}} \subseteq [\![e']\!]_{CRWL_{let}} \textit{ implies } [\![\mathcal{C}[e]]\!]_{CRWL_{let}} \subseteq [\![\mathcal{C}[e']]\!]_{CRWL_{let}}$$

Unfortunately this property is false because of the possible capture of variables when switching from e to $\mathcal{C}[e]$, as the following example shows:

EXAMPLE 3. *If f is defined by $f(0) \to 1$ we have*

$$\{\bot\} \equiv [\![f(X)]\!] \subseteq [\![0]\!] \equiv \{\bot, 0\}$$

but when these expressions are placed within the context $let\ X = 0\ in\ [\,]$ we obtain

$$\{\bot, 1\} \equiv [\![let\ X = 0\ in\ f(X)]\!] \not\subseteq [\![let\ X = 0\ in\ 0]\!] \equiv \{\bot, 0\}.$$

To overcome this problem and prove Theorem 2 we need a stronger result showing that $\to_l$-steps preserve (in the sense of $\subseteq$) the *CRWL*$_{let}$-semantics even under substitutions. To formalize the idea some new notions are useful:

DEFINITION 1 (Hypersemantics).

(i) The hypersemantics *of an expression $e \in LExp_\bot$, written as $[\![\![e]\!]\!]_{CRWL_{let}}$, is a mapping from $CSubst_\bot$ into $\mathcal{P}(CTerm_\bot)$ defined as*

$$[\![\![e]\!]\!]_{CRWL_{let}}\ \theta = [\![e\theta]\!]_{CRWL_{let}}.$$

(ii) Hypersemantics of expressions are ordered as follows:

$$[\![\![e_1]\!]\!]_{CRWL_{let}} \Subset [\![\![e_2]\!]\!]_{CRWL_{let}} \textit{ iff } [\![e_1\theta]\!]_{CRWL_{let}} \subseteq [\![e_2\theta]\!]_{CRWL_{let}},\ \forall\theta \in CSubst_\bot$$

In other terms, iff $\forall\theta \in CSubst_\bot$, $\mathcal{P} \vdash_{CRWL_{let}} e_1\theta \to t$ implies $\mathcal{P} \vdash_{CRWL_{let}} e_2\theta \to t$.

Hypersemantics fulfils the desired monotonicity property:

LEMMA 4. *For any $e, e' \in LExp_\bot$, and every context $\mathcal{C}$ we have:*

$$[\![\![e]\!]\!]_{CRWL_{let}} \Subset [\![\![e']\!]\!]_{CRWL_{let}} \textit{ implies } [\![\![\mathcal{C}[e]]\!]\!]_{CRWL_{let}} \Subset [\![\![\mathcal{C}[e']]\!]\!]_{CRWL_{let}}$$

Now the idea is to prove for hypersemantics a result analogous to Theorem 2, which will become then an easy corollary. Two more lemmas are needed: the first is a standard substitution lemma and the second is a classical result for *CRWL* [10], that is also valid for $CRWL_{let}$.

LEMMA 5. *Given $e, e' \in LExp_\bot$, $\theta \in Subst_\bot$ and $X \in \mathcal{V}$ such that $X \notin dom(\theta)$ and $X \notin ran(\theta)$, then we have $(e[X/e'])\theta \equiv e\theta[X/e'\theta]$.*

LEMMA 6. *Let $e, e' \in LExp_\bot$, $t, t' \in CTerm_\bot$ be such that $e \sqsubseteq e'$ and $t \sqsupseteq t'$. If $e \to t$ then $e' \to t'$ with a proof of the same size or smaller.*

All these results allow to prove the expected generalization of Theorem 2 to hypersemantics.

THEOREM 3 (One-Step Hyper-Soundness of *let*-rewriting).
For any $e, e' \in LExp$

$$e \to_l e' \textit{ implies } [\![\![e']\!]\!]_{CRWL_{let}} \Subset [\![\![e]\!]\!]_{CRWL_{let}}$$

And now Theorem 2 follows naturally:
PROOF:[For Theorem 2] Assume $e \to_l e'$. By Theorem 3 we have $[\![\![e']\!]\!]_{CRWL_{let}} \Subset [\![\![e]\!]\!]_{CRWL_{let}}$, and therefore $[\![e'\theta]\!]_{CRWL_{let}} \subseteq [\![e\theta]\!]_{CRWL_{let}}$ for each $\theta \in CSubst_\bot$. Choosing $\theta = \epsilon$ (the empty substitution) we obtain $[\![e']\!]_{CRWL_{let}} \subseteq [\![e]\!]_{CRWL_{let}}$ as desired. □

One-step soundness as given by Theorem 2 is straightforwardly extended to several steps, that is, to the transitive and reflexive closure $\to_l^*$ of the *let*-rewriting relation $\to_l$:

COROLLARY 1. *For any $e, e' \in LExp$*

$$e \to_l^* e' \textit{ implies } [\![e']\!]_{CRWL_{let}} \subseteq [\![e]\!]_{CRWL_{let}}$$

PROOF: An immediate induction on the length of the derivation $e \to_l^* e'$. □

Finally we can easily get our main result concerning the soundness of *let*-rewriting with respect not only to the *CRWL*$_{let}$ calculus, but also to the original *CRWL* formulation:

THEOREM 4 (Soundness of *let*-rewriting).
Let $\mathcal{P}$ be a CRWL-program and $e \in Exp$. Then:
(i) $e \to_l^* e'$ implies $\mathcal{P} \vdash_{CRWL} e \to |e'|$, for any $e' \in LExp$.
(ii) $e \to_l^* t$ implies $\mathcal{P} \vdash_{CRWL} e \to t$, for any $t \in CTerm$.

PROOF: *(i)*: Assume $e \to_l^* e'$. Then, by Corollary 1 we have $[\![e']\!]_{CRWL_{let}} \subseteq [\![e]\!]_{CRWL_{let}}$. Since $|e'| \in [\![e']\!]_{CRWL_{let}}$ by Lemma 2, we get $|e'| \in [\![e]\!]_{CRWL_{let}}$, which means $\mathcal{P} \vdash_{CRWL_{let}} e \to |e'|$. By Lemma 3, we conclude $\mathcal{P} \vdash_{CRWL} e \to |e'|$.
(ii): trivial by *(i)*, since $|t| = t$ for $t \in CTerm$. □

5.2 Completeness

Now we look for the reverse implication of Theorem 4. Some additional results are needed for it. The first one concerns only $\to_l$-reductions:

LEMMA 7 (Peeling lemma). *For any $e \in LExp$ such that $e \notin \mathcal{V}$ we have that*

$$e \to_l^* let\ \overline{X = a}\ in\ g(\overline{t})$$

for some $g \in CS \cup FS, \overline{t} \subseteq CTerm$ and $\overline{a} \subseteq LExp$ such that $|a_i| = \bot$ for all $a_i \in \overline{a}$. Moreover, if $e \equiv h(e_1, \ldots, e_n)$ with $h \in CS \cup FS$, then

$$e \to_l^* let\ \overline{X = a}\ in\ h(t_1, \ldots, t_n)$$

under the conditions above, and verifying also that $t_i \equiv e_i$ whenever $e_i \in CTerm$.

Besides, we can state that in these derivations the rule **(Fapp)** *was not applied.*

We can think about a *let*-expression as a regular *CRWL*-term in which some additional sharing information has been encoded using *let* expressions. As we do not use the rule **(Fapp)** in the derivations for this lemma, we do not make progress in the evaluation of the implicit *CRWL*-term corresponding to e (thus not changing the corresponding *CRWL*-denotation), but we change the sharing-enriched representation of this *CRWL*-term in the *let*-rewriting syntax. What we do in these derivations is exposing the computed part of e concentrating it in $g(\overline{t})$, that is, the part whose shell is different from $\bot$. That is why we call it '*Peeling lemma*'.

The next result is already a technical completeness result preparing for our completeness theorems below:

LEMMA 8. *Let $\mathcal{P}$ be a CRWL-program, $e \in Exp$, and $t \in CTerm_\bot$ such that $t \neq \bot$. Then:*

$$\mathcal{P} \vdash_{CRWL} e \to t \textit{ implies } e \to_l^* let\ \overline{X = a}\ in\ t'$$

for some $t' \in CTerm$ and $\overline{a} \subseteq LExp$ in such a way that $t \sqsubseteq |let\ \overline{X = a}\ in\ t'|$ and $|a_i| = \bot$ for all $a_i \in \overline{a}$. As a consequence, $t \sqsubseteq t'[\overline{X/\ \bot}]$.

Our main results concerning the completeness of *let*-rewriting are now easy consequences of Lemma 8. The first shows that any c-term obtained by *CRWL* for an expression can be refined by a *let*-rewriting derivation.

THEOREM 5 (Completeness of *let*-rewriting).
Let $\mathcal{P}$ be a CRWL-program, $e \in Exp$, and $t \in CTerm_\perp$. Then:

$$\mathcal{P} \vdash_{CRWL} e \rightarrowtriangle t \text{ implies } e \rightarrow_l^* e'$$

for some $e' \in LExp$ such that $t \sqsubseteq |e'|$.

PROOF: If $t = \perp$ then we are done with $e \rightarrow_l^0 e$ as $\forall e, \perp \sqsubseteq |e|$. If $t \neq \perp$ then by Lemma 8 we have $e \rightarrow_l^* let\ \overline{X} = \overline{a}\ in\ t'$ such that $t \sqsubseteq |let\ \overline{X} = \overline{a}\ in\ t'|$. □

The next result considers the case of total c-terms:

THEOREM 6 (Completeness of *let*-rewriting for total solutions).
Let $\mathcal{P}$ be a CRWL-program, $e \in Exp$, and $t \in CTerm$. Then:

$$\mathcal{P} \vdash_{CRWL} e \rightarrowtriangle t \text{ implies } e \rightarrow_l^* t.$$

PROOF: Assume $\mathcal{P} \vdash_{CRWL} e \rightarrowtriangle t$, then by Lemma 8 we get $e \rightarrow_l^* let\ \overline{X} = \overline{a}\ in\ t'$ such that $t \sqsubseteq |let\ \overline{X} = \overline{a}\ in\ t| \equiv t'[\overline{X}/\perp]$, for some $t' \in CTerm, \overline{a} \subseteq LExp$. As $t \in CTerm$ then t is maximal w.r.t. $\sqsubseteq$, so $t \sqsubseteq t'[\overline{X}/\perp]$ implies $t'[\overline{X}/\perp] \equiv t$, but then $t'[\overline{X}/\perp] \in CTerm$ so it must happen that $FV(t') \cap \overline{X} = \emptyset$ and therefore $t' \equiv t'[\overline{X}/\perp] \equiv t$. But then $let\ \overline{X} = \overline{a}\ in\ t' \rightarrow_l^* t' \equiv t$ by zero or more steps of **(Elim)**, so $e \rightarrow_l^* let\ \overline{X} = \overline{a}\ in\ t' \rightarrow_l^* t$, that is $e \rightarrow_l^* t$. □

As a final corollary of this result and the part *(ii)* of the soundness Theorem 4 we obtain a strong equivalence result for both formalisms:

THEOREM 7 (Equivalence of *CRWL* and *let*-rewriting).
Let $\mathcal{P}$ be a CRWL-program, $e \in Exp$, and $t \in CTerm$. Then:

$$\mathcal{P} \vdash_{CRWL} e \rightarrowtriangle t \text{ iff } e \rightarrow_l^* t.$$

This constitutes the main result in the paper.

6. Let-rewriting versus classical rewriting

In this section we examine the relationship between *let*-rewriting and ordinary rewriting for TRS. We will first prove in 6.1 that *let*-rewriting is sound with respect to rewriting. As we know since the discussion starting the paper, completeness does not hold in general because, in presence on non-determinism, rewriting (that corresponds to run-time choice) can obtain more results than *let*-rewriting (call-time choice). However, we will be able to prove completeness for programs that are *deterministic*, a property close to confluence that will be defined in 6.2.

Thanks to the equivalence of *CRWL* and *let*-rewriting we can choose the most appropriate point of view for each of the two goals (soundness and completeness): we will use *let*-rewriting for proving soundness, and the proof calculus of *CRWL* for defining the property of determinism and proving that, under determinism, completeness holds.

6.1 Soundness of let-rewriting w.r.t. classical rewriting

Firstly, we need a syntactic transformation from $LExp$ into Exp, removing the *let* constructions (thus losing the sharing information they provide). Given $e \in LExp$ we define its transformation into a standard expression $\widehat{e}$ as:

$$\begin{aligned} &\widehat{X} \equiv X \\ &\widehat{h(e_1, \ldots, e_n)} \equiv h(\widehat{e_1}, \ldots, \widehat{e_n}) \\ &\widehat{let\ X_p = e_1\ in\ e_2} \equiv \widehat{e_2}[X_p/\widehat{e_1}] \end{aligned}$$

This transformation satisfies the following properties:

LEMMA 9. *For all $e \in LExp$ we have $\widehat{e} \in Exp$, $var(\widehat{e}) \subseteq FV(e)$, $|\widehat{e}| \equiv |e|$. Moreover, for all $e \in Exp$ we have $\widehat{e} \equiv e$.*

The following lemmas can be easily proved by induction on the structure of expressions:

LEMMA 10. *For all $e, s, s' \in Exp$, $X \in \mathcal{V}$, $s \rightarrow^* s'$ implies $e[X/s] \rightarrow^* e[X/s']$.*

LEMMA 11. *For all $e, s \in LExp$, $X \in \mathcal{V}$: $\widehat{e[X/s]} \equiv \widehat{e}[X/\widehat{s}]$.*

Using these lemmas we get a first soundness result, stating that what can be done in one step of *let*-rewriting, can also be done in zero or more steps of ordinary rewriting, after erasing the sharing information by the transformation $\widehat{\ }$:

LEMMA 12. *For all $e, e' \in LExp$ we have: $e \rightarrow_l e'$ implies $\widehat{e} \rightarrow^* \widehat{e'}$.*

Some other soundness results follow easily from the lemma above. The first one expresses that any expression (not involving *let*'s) reachable by *let*-rewriting can be also reached by ordinary rewriting. In other terms, *let*-rewriting ($\rightarrow_l^*$) is a sub-relation of rewriting ($\rightarrow^*$), when ($\rightarrow_l^*$) is restricted to ordinary expressions (not involving *let*'s).

THEOREM 8.
For any $e, e' \in LExp$, $e \rightarrow_l^* e'$ implies $\widehat{e} \rightarrow^* \widehat{e'}$. As a consequence, if $e, e' \in Exp$, then $e \rightarrow_l^* e'$ implies $e \rightarrow^* e'$.

PROOF: An immediate induction on the length of the *let*-derivation, using Lemma 12 for the inductive step. For the remaining statement, if $e, e' \in Exp$ then $e \equiv \widehat{e}, e' \equiv \widehat{e'}$ by Lemma 9, and therefore $e \equiv \widehat{e} \rightarrow^* \widehat{t} \equiv t$. □

The next result, based on the correspondence of *CRWL* and *let*-rewriting established in Section 5, is a soundness theorem for *CRWL* with respect to ordinary rewriting.

THEOREM 9. *For all $e \in Exp$ and $t \in CTerm_\perp$, $\mathcal{P} \vdash_{CRWL} e \rightarrowtriangle t$ implies $\exists e' \in Exp$ such that $e \rightarrow^* e'$ and $t \sqsubseteq |e'|$.*

PROOF: Assume $\mathcal{P} \vdash_{CRWL} e \rightarrowtriangle t$, then by Theorem 5 $\exists e'' \in LExp$ such that $e \rightarrow_l^* e''$ and $t \sqsubseteq |e''|$. Then by Theorem 8 combined with Lemma 9 we get $e \equiv \widehat{e} \rightarrow^* \widehat{e''}$. But then we can choose $e' \equiv \widehat{e''}$ because $\widehat{e''} \in Exp$ by Lemma 9, and $|e'| \equiv |\widehat{e''}| = |e''| \sqsupseteq t$, by Lemma 9 again. □

6.2 Completeness of *CRWL* w.r.t. classical rewriting

As commented before, we cannot expect to get a completeness result of the *CRWL* framework w.r.t. classical rewriting for any program, but only for the class of deterministic programs, which are defined as follows:

DEFINITION 2 (Deterministic *CRWL*-program).
A CRWL-program $\mathcal{P}$ is deterministic *iff the denotation $[\![e]\!]^{\mathcal{P}}$ of any expression $e \in Exp_\perp$ is a directed set. In other words, iff $\forall e \in Exp_\perp$ and $t_1, t_2 \in [\![e]\!]^{\mathcal{P}}$ there exists $t_3 \in [\![e]\!]^{\mathcal{P}}$ with $t_1 \sqsubseteq t_3$ and $t_2 \sqsubseteq t_3$.*

Determinism as defined here is intuitively close to confluence, but the two notions do not coincide. Determinism does not imply confluence, as the following example shows:

EXAMPLE 4. *Consider the program $\mathcal{P}$ given by the three rules*

$$f \rightarrow a \qquad f \rightarrow loop \qquad loop \rightarrow loop$$

where a is a constructor. It is clear that $\mathcal{P}$ is not confluent (f can be reduced to a and $loop$, which cannot be joined to a common reduct), *but is deterministic, since* $[\![f]\!]^{\mathcal{P}} = \{\bot, a\}$, $[\![loop]\!]^{\mathcal{P}} = \{\bot\}$ *and* $[\![a]\!]^{\mathcal{P}} = \{\bot, a\}$, *each of them being a directed set.*

We conjecture that the reverse implication (confluence ⇒ determinism) is true, but a precise proof of this fact seems surprisingly complicated and we have not yet completed it.

Determinism has been defined as a semantic property. However, thanks to the equivalence of *CRWL* and *let*-rewriting, it can be also characterized in terms of reduction, as the following result shows:

LEMMA 13. *A CRWL-program* $\mathcal{P}$ *is deterministic iff for any expressions* $e, e', e'' \in Exp$ *with* $e \to_l^* e'$ *and* $e \to_l^* e''$, *there exists* $e''' \in Exp$ *such that* $e \to_l^* e'''$ *and* $|e'''| \sqsupseteq |e'|, |e'''| \sqsupseteq |e''|$.

We do not know if in this result *let*-rewriting can be replaced by ordinary rewriting.

We need also the following auxiliary notions:

DEFINITION 3 (Denotation for a substitution).
Given a CRWL-program P, for all $\sigma \in Subst_\bot$ *its denotation is defined as* $[\![\sigma]\!] = \{\theta \in CSubst_\bot | dom(\theta) = dom(\sigma) \wedge \forall X \in dom(\theta), \mathcal{P} \vdash_{CRWL} \sigma(X) \to\!\!\!\!\triangleright\, \theta(X)\}$.

DEFINITION 4 (Deterministic substitution).
The set of deterministic substitutions *for a given CRWL-program* $\mathcal{P}$, $DSubst_\bot$ *is defined as*

$$\begin{aligned} DSubst_\bot = \{\theta \in Subst_\bot | \quad & \forall X \in dom(\theta). \\ & [\![\theta(X)]\!]^{\mathcal{P}} \textit{ is a directed set}\} \end{aligned}$$

Using these notions we can develop an extension of the proof calculus for *CRWL* which does call-by-name parameter passing only when it is safe for call-time choice. The extended calculus $CRWL^d$ contains the same rules of *CRWL* and the following additional rule:

$$(\mathbf{OR^d})\ \frac{r\theta \to\!\!\!\!\triangleright\, t}{f(\overline{p})\theta \to\!\!\!\!\triangleright\, t} \quad \text{if } (f(\overline{p}) \to r) \in \mathcal{P} \text{ and } \theta \in DSubst_\bot$$

Besides, for every $e \in Exp_\bot$ we define its denotation in this calculus as $[\![e]\!]^d = \{t \in CTerm_\bot | \mathcal{P} \vdash_{CRWL^d} e \to\!\!\!\!\triangleright\, t\}$. Notice that this relation is undecidable (as happens with confuence) because the problem of checking whether a *CRWL*-denotation is a directed set or not is undecidable.

We will see that $CRWL^d$ proves exactly the same approximation statements that *CRWL* proves; to do that we must prove first the following auxiliary results:

LEMMA 14. *For any CRWL-program* $\mathcal{P}$ *and for all* $\sigma \in DSusbt_\bot$, $[\![\sigma]\!]$ *is a directed set.*

LEMMA 15. *For any CRWL-program* $\mathcal{P}$ *and for all* $\sigma \in DSusbt_\bot$, $e \in Exp_\bot, t \in CTerm_\bot$, $\mathcal{P} \vdash_{CRWL} e\sigma \to\!\!\!\!\triangleright\, t$ *implies* $\exists\theta \in [\![\sigma]\!]$ *such that* $\mathcal{P} \vdash_{CRWL} e\theta \to\!\!\!\!\triangleright\, t$.

Now we have at our disposal the tools needed to state and prove the adequacy of $CRWL^d$:

THEOREM 10. *For any CRWL-program* $\mathcal{P}$ *and* $\forall e \in Exp_\bot$, $[\![e]\!]^d = [\![e]\!]$.

Now we are ready to prove our first completeness result:

LEMMA 16. *For any CRWL-program* $\mathcal{P}$, *if it is deterministic then for all* $e, e' \in Exp$, $e \to^* e'$ *implies* $[\![e']\!] \subseteq [\![e]\!]$.

The previous lemma, together with the equivalence of *CRWL* and *let*-rewriting given by Theorem 7, allows to obtain strong relationships between rewriting, let-rewriting and *CRWL*, for the class of determinsitic programs.

THEOREM 11.
Let $\mathcal{P}$ *be a deterministic CRWL-program,* $e, e' \in Exp, t \in CTerm$. *Then:*
a) $e \to^* e'$ *implies* $e \to_l^* e''$ *for some* $e'' \in LExp$ *with* $|e''| \sqsupseteq |e'|$.
b) $e \to^* t$ *iff* $e \to_l^* t$ *iff* $\mathcal{P} \vdash_{CRWL} e \to\!\!\!\!\triangleright\, t$.

PROOF: *a)* Assume $e \to^* e'$. Then $[\![e']\!] \subseteq [\![e]\!]$ by Lemma 16. Now, it is a known property of *CRWL* that $|e'| \in [\![e']\!]$, and then $|e'| \in [\![e]\!]$, which means that $\mathcal{P} \vdash_{CRWL} e \to\!\!\!\!\triangleright\, |e'|$. Therefore, by Theorem 7 there exists $e'' \in LExp$ such that $e \to_l^* e''$ with $|e''| \sqsupseteq |e'|$.

b) That $e \to_l^* t$ iff $\mathcal{P} \vdash_{CRWL} e \to\!\!\!\!\triangleright\, t$, and that $e \to_l^* t$ implies $e \to^* t$ have been already proved for arbitrary programs in Theorems 7 and 8 respectively. What remains to be proved is that $e \to^* t$ implies $e \to_l^* t$ (i.e., $\mathcal{P} \vdash_{CRWL} e \to\!\!\!\!\triangleright\, t$). Assume $e \to^* t$. Then $[\![t]\!] \subseteq [\![e]\!]$ by Lemma 16. Now, it is an easy property of *CRWL* that $t \in [\![t]\!]$, and therefore $t \in [\![e]\!]$, which exactly means that $\mathcal{P} \vdash_{CRWL} e \to\!\!\!\!\triangleright\, t$. □

Notice that in part a) we cannot ensure $e \to^* e'$ implies $e \to_l^* e'$, because rewriting can reach some intermediate expressions not reachable by *let*-rewriting. For instance, given the deterministic program with the rules $g \to a$ and $f(x) \to c(x, x)$, we have $f(g) \to^* c(g, a)$, but not $f(g) \to_l^* c(g, a)$. Still, part a) is a strong completeness result for let-rewriting wrt rewriting for deterministic programs, since it says that the outer constructed part obtained in a rewriting derivation can be also obtained or even refined in a *let*-derivation. Combined with Theorem 8, part a) expresses a kind of equivalence between *let*-rewtiting and rewriting, valid for general derivations, even non-terminating ones. For terminated derivations reaching a constructor term (not further reducible), part b) gives an even stronger equivalence result.

7. Related work and conclusions

This work tries to fill a gap existing in the functional logic programming field, which is the technical disconnection between the two most accepted approaches to the paradigm: one, given by the *CRWL* framework, more biased to the semantics, and the other, focused in operational aspects, based on the theory or term rewriting. We feel that the missing piece was a precise, simple, high level and clear one-step reduction mechanism that is close to rewriting but at the same time respects call-time choice semantics for possibly non-confluent and non-terminating constructor-based rewrite systems.

There exist previous proposals that combine sharing with rewriting or narrowing, even for the specific case of functional logic programs. We briefly discuss now why we decided not to adopt them for our aim of comparison with *CRWL*.

A usual approach to expressing different levels of sharing in rewriting is term graph rewriting [20], a variant of which for constructor based systems was studied in [6, 7]. However, the class of programs is smaller in that work, since rewrite rules in term graph rewrite systems must be orthogonal and extra variables are not considered. These restrictions were dropped in [3], but it does not contain any formal treatment for the properties of the proposed notions. Furthermore, and admitting that this is arguable, we consider that graph rewriting is a complex mechanism to reason about. For instance, we see graph homomorphisms as a more involved notion than matching. Therefore, we find it more comfortable, whenever possible, to use textual or equational counterparts of graph rewriting, as in essence is our *let*-rewriting or the λ-calculus with sharing of [18].

In [1] there is a proposal of two operational (natural and small-step) semantics for functional logic programs supporting sharing (call-time choice semantics), using a flat representation of programs coming from an implicit program transformation encoding the demand analysis used by needed narrowing, and some kind of heaps to express bindings for variables. As in our case, *let*-expressions are used to express sharing. The approach is useful as a technical basis for implementation and program manipulation purposes; but we think that, as happens with *CRWL* but for rather different reasons –too low-level and close to a particular operational strategy– it cannot be seen as the 'essential' basic reduction mechanism to understand non-strict call-time choice. Furthermore, to relate technically *CRWL* with [1] turns out to be a really hard task, that has been done in [15] but only for a restricted class of programs and expressions.

Local bindings *let X=e in ...* resemble oriented conditions $e \rightarrow X$ of the deterministic conditional rewrite systems of [19]. But they consider 3-CTRS systems and, most importantly, a different semantics for equality, according to which call-time choice is not respected.

Finally, for proving the completeness of a transformation that eliminates extra variables, [5] uses a variant or rewriting explicit substitution. However, their variant performs sharing only for the extra variables to be eliminated and not for the whole process of rewriting, and then they do not really achieve call-time choice.

Our concrete contributions can be summarized as follows:

- We have further clarified the well known fact that ordinary rewriting is not adequate for call-time choice, by showing that no program transformation can serve to fully simulate call-time choice by ordinary rewriting (Sect. 3). Therefore, the classical theory of TRS cannot serve as technical foundation for functional logic programs with call-time choice. Then we have proposed two one-step variants of rewriting.
- The first variant (Sect. 3) is very simple but of limited interest since it alters the natural sequence of rewriting in real computations.
- The second one (called *let*-rewriting in the paper) defines rewriting with local bindings. The rules for *let*-rewriting are very similar, but adapted to term rewriting with call-time choice, to those for λ-calculus with sharing [18], and can be seen as a particular textual (equational) presentation of graph term rewriting [20].
- As a major technical task we have proved the equivalence of *let*-rewriting and *CRWL*, which is the core of our contribution. Equivalence is a strong result that allows to apply known and future results about *CRWL* to *let*-rewriting and viceversa. Just to mention an example, the program transformations proved to be correct for *CRWL* in [15] are also valid for *let*-rewriting. As a technical tool for proving equivalence we have extended the *CRWL* logic itself to deal with local bindings, which might be a useful side-product.
- We have proved that for deterministic programs (a semantic condition very close to confluence) let-rewriting (hence *CRWL*-derivability) and ordinary rewriting coincide in some precise technical sense, while in general let-rewriting is a sub-relation of rewriting. We stress the fact that this is a new, technically non-trivial result connecting the *CRWL* and rewrite worlds; to the best of our knowledge, this kind of results were completely missing in the literature. Furthermore, we strongly conjecture (and we are hopefully very close to a complete proof) that confluence of a *CRWL*-program (in the ordinary sense of TRS) implies semantic determinism, which will imply that under confluence rewriting and let-rewriting are equivalent in some technical sense. This very intuitive (but hard to prove!) result will give further evidence (if it finally becomes proved) of the benefits of having connected *CRWL* and rewriting, since a result related purely to rewriting would become proved using semantical reasoning tools.

We must warn that *let*-rewriting as presented in this paper does not pretend to be in its own the working operational procedure for c-rewrite systems with call-time choice (functional-logic programs), for several reasons: first, we have not considered any rewriting strategy – something needed in practice – otherwise the rewriting space is too large. Second, there are two situations in computations where rewriting is not enough and must be lifted to narrowing: when the program uses extra variables (narrowing must be used then to obtain their values; rewriting 'magically' guesses them in the parameter passing substitution) and when the initial expression to reduce has variables. The extension of our work to cope with narrowing and strategies is left to future work. But we think that to present first a notion of rewriting with respect to which one can prove correctness and completeness of subsequent notions of narrowing and strategies is an advantage rather than a lack of our approach.

As additional future work, we plan to extend our work to the HO case as to obtain rewriting counterparts of HO-*CRWL* [8], and to relate technically *let*-rewriting with more formalisms like term graph rewriting or explicit substitutions, obtaining thus a wider picture of reduction under non-strict call-time choice.

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. Evaluation strategies for functional logic programming. *Electronic Notes in Theoretical Computer Science*, 57, 2001.

[3] S. Antoy, D. Brown, and S. Chiang. Lazy context cloning for non-deterministic graph rewriting. In *Proc. of the 3rd International Workshop on Term Graph Rewriting, Termgraph'06*, pages 61–70, Vienna, Austria, April 2006. To appear in ENTCS.

[4] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *J. ACM*, 47(4):776–822, 2000.

[5] S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Twenty Second International Conference on Logic Programming*, pages 87–101, Seattle, WA, Aug. 2006. Springer LNCS 4079.

[6] R. Echahed and J.-C. Janodet. On constructor-based graph rewriting systems. Research Report 985-I, IMAG, 1997.

[7] R. Echahed and J.-C. Janodet. Admissible graph rewriting and narrowing. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 325 – 340, Manchester, June 1998. MIT Press.

[8] J. González-Moreno, M. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. In *Proc. International Conference on Logic Programming (ICLP'97)*, pages 153–167. MIT Press, 1997.

[9] 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.

[10] 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.

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

[12] 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.

[13] H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.

[14] 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.

[15] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Relating two semantic descriptions of functional logic programs. In *Proc. Jornadas sobre Programación y Lenguajes (PROLE'06)*, pages 31–40. CINME, 2006.

[16] 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.

[17] F. López-Fraguas and J. Sánchez-Hernández. Functional logic programming with failure: A set-oriented view. In *Proc. International Conference on Logic for Programming and Automated Reasoning (LPAR'01)*, pages 455–469. Springer LNAI 2250, 2001.

[18] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *J. Funct. Program.*, 8(3):275–317, 1998.

[19] M. Marin and A. Middeldorp. New completeness results for lazy conditional narrowing. In *PPDP*, pages 120–131, 2004.

[20] D. Plump. Essentials of term graph rewriting. *Electr. Notes Theor. Comput. Sci.*, 51, 2001.

[21] 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.

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

[23] H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, 1992.

[24] R. d. Vado-Vírseda. A demand-driven narrowing calculus with overlapping definitional trees. In *Proc. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming (PPDP'03)*, pages 213–227. ACM Press, 2003.

A. Some proofs

In this appendix we use the sets $[\mathcal{P}]_\perp$ and $[\mathcal{P}]$ of partial and total c-instances of the rules of a program $\mathcal{P}$, respectively, defined as:

$$\begin{array}{lcl} [\mathcal{P}]_\perp & = & \{(f(t) \to e)\theta | \theta \in CSubst_\perp\} \\ [\mathcal{P}] & = & \{(f(t) \to e)\theta | \theta \in CSubst\} \end{array}$$

Now, the rule **(OR)** of $CRWL_{(let)}$ can be reformulated as:

$$\textbf{(OR)}\ \frac{e_1 \twoheadrightarrow t_1 \ \ldots\ e_n \twoheadrightarrow t_n \quad e \twoheadrightarrow t}{f(e_1, \ldots, e_n) \twoheadrightarrow t} \quad (f(\bar{t}) \to e) \in [\mathcal{P}]_\perp$$

and the rule **(Fapp)** as:

$$\textbf{(Fapp)} \quad f(t_1, \ldots, t_n) \to_l e \quad \text{if } f(t_1, \ldots, t_n) \to e \in \mathcal{P}$$

PROOF:[For Theorem 3] We assume $\theta \in CSubst_\perp$ such that $e'\theta \twoheadrightarrow t$. We must prove that $e\theta \twoheadrightarrow t$. The case where $e'\theta \twoheadrightarrow\perp$ holds trivially using the rule (B), so we will prove the other by a case distinction on the rule of the let calculus applied:

(Contx) If $\mathcal{C}[e] \to_l \mathcal{C}[e']$ because $e \to_l e'$, we can always suppose that $e \to_l e'$ without applying **(Contx)**, because if it was applied then we would have $e \equiv \mathcal{C}'[e_1] \to_l \mathcal{C}'[e_2] \equiv e'$ with $e_1 \to_l e_2$, and so we can define $\mathcal{C}''[] \equiv \mathcal{C}[\mathcal{C}'[]]$ and $\mathcal{C}''[e_1] \equiv \mathcal{C}[\mathcal{C}'[e_1]] \equiv \mathcal{C}[e] \to_l \mathcal{C}[e'] \equiv \mathcal{C}[\mathcal{C}'[e_2]] \equiv \mathcal{C}''[e_2]$. We can repeat that process ensuring that the rule **(Contx)** was not applied in $e \to_l e'$. So, by the proof of the other cases, $[\![e']\!] \Subset [\![e]\!]$, and by Lemma 4, $[\![\mathcal{C}[e']]\!] \Subset [\![\mathcal{C}[e]]\!]$, and we are done.

(Elim) Assume $let\ X = e_1\ in\ e_2 \to_l e_2$ and $\theta \in CSubts_\perp$ such that $\mathcal{P} \vdash_{CRWL_{let}} e_2\theta \twoheadrightarrow t$:

$$\dfrac{\dfrac{}{e_1\theta \twoheadrightarrow\perp}\ B \quad \dfrac{}{e_2\theta[X/\perp] \equiv e_2\theta \twoheadrightarrow t}\ Hypothesis}{let\ X = e_1\theta\ in\ e_2\theta \twoheadrightarrow t}\ Let$$

We know that $X \notin ran(\theta)$ because of the way we have defined substitutions [1]. Then as $X \notin FV(e_2)$ for the condition of **(Elim)**, $X \notin FV(e_2\theta)$ and so $e_2\theta[X/\perp] \equiv e_2\theta$.

(Bind) Assume $let\ X = t_1\ in\ e \to_l e[X/t_1]$ and $\theta \in CSubst_\perp$ such that $\mathcal{P} \vdash_{CRWL_{let}} (e[X/t_1])\theta \twoheadrightarrow t$:

$$\dfrac{\dfrac{}{t_1\theta \twoheadrightarrow t_1\theta}\ DC{*} \quad \dfrac{}{e\theta[X/t_1\theta] \equiv (e[X/t_1])\theta \twoheadrightarrow t}\ Hyp}{let\ X = t_1\theta\ in\ e\theta \twoheadrightarrow t}\ Let$$

By our definition of substitutions we assume $X \notin dom(\theta)$ and $X \notin ran(\theta)$, so by Lemma 5 we have $e\theta[X/t_1\theta] \equiv (e[X/t_1])\theta$. Besides, "rule" $[DC*]$ refers to the fact that $\forall t \in CTerm_\perp.\mathcal{P} \vdash_{CRWL_{let}} t \twoheadrightarrow t$ (very easy to prove).

(Flat) Assume $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \to_l let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$ and $\theta \in CSubts_\perp$ such that $\mathcal{P} \vdash_{CRWL_{let}} (let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3))\theta \twoheadrightarrow t$. This proof must be must be of the shape of:

$$\dfrac{e_1\theta \twoheadrightarrow t_1 \quad \dfrac{e_2\theta[Y/t_1] \twoheadrightarrow t_2\ (e_3\theta[Y/t_1] \equiv e_3\theta)[X/t_2] \twoheadrightarrow t}{(let\ X = e_2\theta\ in\ e_3\theta)[Y/t_1] \twoheadrightarrow t}\ Let}{let\ Y = e_1\theta\ in\ (let\ X = e_2\theta\ in\ e_3\theta) \twoheadrightarrow t}\ Let$$

for some $t_1, t_2 \in CTerm_\perp$. Besides, because of the way we have defined substitutions, $Y \notin ran(\theta)$, so as by the condition of **(Flat)**, $Y \notin FV(e_3)$, then $Y \notin FV(e_3\theta)$ and we can say $e_3\theta[Y/t_1] \equiv e_3\theta$. So:

$$\dfrac{\dfrac{\dfrac{}{e_1\theta \twoheadrightarrow t_1}\ Hyp \quad \dfrac{}{e_2\theta[Y/t_1] \twoheadrightarrow t_2}\ Hyp}{let\ Y = e_1\theta\ in\ e_2\theta \twoheadrightarrow t_2}\ Let \quad \dfrac{}{e_3\theta[X/t_2] \twoheadrightarrow t}\ Hyp}{let\ X = (let\ Y = e_1\theta\ in\ e_2\theta)\ in\ e_3\theta \twoheadrightarrow t}\ Let$$

(LetIn) Assume $h(d_1, \ldots, e, \ldots, d_n) \to_l let\ X = e\ in\ h(d_1, \ldots, X, \ldots, d_n)$ and $\theta \in CSubts_\perp$ such that $\mathcal{P} \vdash_{CRWL_{let}} (let\ X = e\ in\ h(d_1, \ldots, X, \ldots, d_n))\theta \twoheadrightarrow t$. This proof will reduce to the proofs $e\theta \twoheadrightarrow t_1$ and $h(d_1, \ldots, X, \ldots, d_n)\theta[X/t_1] \twoheadrightarrow t$ for some $t_1 \in CTerm_\perp$. By the variable convection $X \notin dom(\theta)$ and $X \notin ran(\theta)$, so as X is fresh then $\forall i.X \notin FV(d_i\theta)$, hence $h(d_1, \ldots, X, \ldots, d_n)\theta[X/t_1] \equiv h(d_1\theta, \ldots, t_1, \ldots, d_n\theta)$. Now there are two possible cases:

a) $h = c \in DC$, then $h(d_1\theta, \ldots, t_1, \ldots, d_n\theta) \twoheadrightarrow t$ must proved by (DC) as:

$$\frac{d_1\theta \twoheadrightarrow s_1\ \ldots\ t_1 \twoheadrightarrow t'_1\ \ldots\ d_n\theta \twoheadrightarrow s_n}{c(d_1\theta, \ldots, t_1, \ldots, d_n\theta) \twoheadrightarrow c(s_1, \ldots, t'_1, \ldots, s_n) \equiv t}$$

for some $s_1, \ldots, s_n, t'_1 \in CTerm_\perp$. As $\forall t \in CTerm_\perp$, $t \twoheadrightarrow t'$ implies $t' \sqsubseteq t$ (easy to prove, as only B, RR and DC could be applied), then $t'_1 \sqsubseteq t_1$, and so as $e\theta \twoheadrightarrow t_1$, by Lemma 6 we have $e\theta \twoheadrightarrow t'_1$. Then we have proofs for $d_1\theta \twoheadrightarrow s_1 \ldots e\theta \twoheadrightarrow t'_1 \ldots d_n\theta \twoheadrightarrow s_n$, and with (DC) we can build a proof for $c(d_1\theta, \ldots, e\theta, \ldots, d_n\theta) \twoheadrightarrow c(s_1, \ldots, t'_1, \ldots, s_n) \equiv t$.

[1] Actually, to prove this theorem properly, we cannot restrict the substitution to fulfil these restrictions, so in fact we rename the bound variables in an α-conversion fashion and use the equivalence $e[X/t] \equiv e[X/Y][Y/t]$ (with Y the new bound variable), to use the hypothesis. We will assume this convention from now on.

b) $h = f \in FS$, then $h(d_1\theta, \ldots, t_1, \ldots, d_n\theta) \rightarrowtriangle t$ must be:

$$\frac{d_1\theta \rightarrowtriangle s_1 \ \ldots \ t_1 \rightarrowtriangle t'_1 \ \ldots \ d_n\theta \rightarrowtriangle s_n \quad r \rightarrowtriangle t}{f(d_1\theta, \ldots, t_1, \ldots, d_n\theta) \rightarrowtriangle t} \ OR$$

for some $(f(s_1, \ldots, t'_1, \ldots, s_n) \to r) \in [\mathcal{P}]_\perp$. Again as $\forall t \in CTerm_\perp, t \rightarrowtriangle t'$ implies $t' \sqsubseteq t$, then $t'_1 \sqsubseteq t_1$, and so as $e\theta \rightarrowtriangle t_1$, by Lemma 6 we have $e\theta \rightarrowtriangle t'_1$. So we have proofs for $d_1\theta \rightarrowtriangle s_1 \ldots e\theta \rightarrowtriangle t'_1 \ldots d_n\theta \rightarrowtriangle s_n$ and $r \rightarrowtriangle t$ and then with (OR) we can build the proof for $f(d_1\theta, \ldots, e\theta, \ldots, d_n\theta) \rightarrowtriangle t$.

(Fapp) Assume $f(t_1, \ldots, t_n) \to_l r$ with $(f(p_1, \ldots, p_n) = e)\sigma \in [\mathcal{P}]$ such that $\forall i.p_i\sigma = t_i$ and $e\sigma = r$, and $\theta \in CSubts_\perp$ such that $\mathcal{P} \vdash_{CRWL_{let}} r\theta \rightarrowtriangle t$. Then as $\theta \circ \sigma \in CSubts_\perp, \forall i.p_i\sigma\theta = t_i\theta$ and $e\sigma\theta = r\theta$ we conclude $(f(p_1, \ldots, p_n) = e)\sigma\theta \in [\mathcal{P}]_\perp$ and so:

$$\frac{\dfrac{}{t_1\theta \rightarrowtriangle t_1\theta}\ DC^* \quad \ldots \quad \dfrac{}{t_n\theta \rightarrowtriangle t_n\theta}\ DC^* \quad \dfrac{}{r\theta \rightarrowtriangle t}\ Hyp}{f(t_1\theta, \ldots, t_n\theta) \rightarrowtriangle t} \ OR$$

□

PROOF:[For Theorem 10] As $CRWL^d$ inherits all the rules of *CRWL* then it is trivially complete. All that is left is proving that the rule **OR**^{d} is sound. Let us suppose an application of **OR**^{d} in which its premise is a *CRWL*-proof, not only a $CRWL^d$-proof, we will see that we can replace that application of **OR**^{d} with an application of **OR**, obtaining exactly the same result. If the starting proof was the following:

$$\frac{r\sigma \rightarrowtriangle t}{f(p_1, \ldots, p_n)\sigma \rightarrowtriangle t} \ \mathrm{OR}^d$$

with $(f(p_1, \ldots, p_n) \to r) \in \mathcal{P}$ and $\sigma \in DSubst_\perp$. Then, as σ is deterministic, applying Lemma 15 under $\mathcal{P} \vdash_{CRWL} r\sigma \rightarrowtriangle t$ we get that there must exist $\theta \in [\![\sigma]\!]$ such that $\mathcal{P} \vdash_{CRWL} r\theta \rightarrowtriangle t$. Besides, we can prove that $\forall i \in \{1, \ldots, n\}$, $\mathcal{P} \vdash_{CRWL} p_i\sigma \rightarrowtriangle p_i\theta$, by induction on the structure of each p_i:

Base cases

• $p_i \equiv X \in \mathcal{V}$: Then there are two possible cases, if $X \notin dom(\sigma)$ then by definition of $[\![\sigma]\!]$ we have that $X \notin dom(\theta)$, so $\mathcal{P} \vdash_{CRWL}$ $\sigma(X) \equiv X \rightarrowtriangle X \equiv \theta(X)$, by **RR**. On the other hand, if $X \in dom(\sigma)$ then by definition of $[\![\sigma]\!]$ we have that $X \in dom(\theta)$ and $\mathcal{P} \vdash_{CRWL} \sigma(X) \rightarrowtriangle \theta(X)$.

• $p_i \equiv c \in CS^0$: Similar to de previous case for $X \notin dom(\sigma)$.

Inductive step Then $p_i \equiv c(t_1, \ldots, t_n)$ and we can do

$$\frac{\dfrac{IH}{t_1\sigma \rightarrowtriangle t_1\theta} \quad \ldots \dfrac{IH}{t_n\sigma \rightarrowtriangle t_n\theta}}{c(t_1\sigma, \ldots, t_n\sigma) \rightarrowtriangle c(t_1\theta, \ldots, t_n\theta)} \ DC$$

As $\theta \in [\![\sigma]\!]$ then $\theta \in CSubst_\perp$ and so it can be used to apply **OR** as follows:

$$\frac{p_1\sigma \rightarrowtriangle p_1\theta \quad \ldots \quad p_n\sigma \rightarrowtriangle p_n\theta \quad r\theta \rightarrowtriangle t}{f(p_1, \ldots, p_n)\sigma \rightarrowtriangle t} \ OR$$

On the other hand, if the starting proof was:

$$\frac{r\sigma \rightarrowtriangle t}{f\sigma \equiv f \rightarrowtriangle t} \ \mathrm{OR}^d \text{ with } (f \to r) \in \mathcal{P} \text{ y } \sigma \in DSubst_\perp$$

then we would have $\theta \in [\![\sigma]\!] \subseteq CSusbt_\perp$ such that $\mathcal{P} \vdash_{CRWL}$ $r\theta \rightarrowtriangle t$, as in the previous case, and we could use it to apply **OR**:

$$\frac{r\theta \rightarrowtriangle t}{f\sigma \equiv f \rightarrowtriangle t} \ OR$$

We have just covered the case where the premise used to apply **OR**^{d} is also a *CRWL*-proof, but for any $CRWL^d$-proof we can apply this transformation from its leaves (the application of rules without premise, like **B** or **RR**) climbing to its parents (the proofs for which they are premises), obtaining an equivalent *CRWL*-proof. □

The next result is a well known result in the scope of *CRWL* and will be used to prove Lemma 7.

LEMMA 17. *Let linear* $p \in CTerm$, *and* $t_1 \in CTerm_\perp$, $t_2 \in CTerm$, $\theta \in CSubst_\perp$. *Then* $p\theta = t_1$ *and* $t_1 \sqsubseteq t_2$ *implies* $\exists \theta' \in CSubst$ *such that* $p\theta' = t_2$ *and* $\theta \sqsubseteq \theta'$.

PROOF:[For Lemma 7] By induction on the structure of e:

Base Case : $e \equiv h$: Then $h \to_l^0 h$, ok with $\overline{X} \equiv \emptyset$.

Inductive Step :

- $e \equiv h(e_1, \ldots, e_n)$: Let us do it for just one argument, for $h(e_1)$. If $e_1 \in CTerm$ then we are done with $\overline{X} \equiv \emptyset$ and $h(e_1) \to_l^0 h(e_1)$, so let us suppose that $e_1 \notin CTerm$. Then $e_1 \notin \mathcal{V}$ so by IH, $e_1 \to_l^* let\ \overline{X_1 = a_1}\ in\ h_1(\overline{t_1})$ with $\overline{X_1} \not\equiv \emptyset$, so:

  $$\begin{array}{l} h(e_1) \to_l^* h(let\ \overline{X_1 = a_1}\ in\ h_1(\overline{t_1})) \\ \quad \to_l^* let\ Y_1 = (let\ \overline{X_1 = a_1}\ in\ h_1(\overline{t_1}))\ in\ h(Y_1) \\ \quad \to_l^* let\ \overline{X_1 = a_1}\ in\ let\ Y_1 = h_1(\overline{t_1})\ in\ h(Y_1) \end{array}$$

  by i.h., **(LetIn)** and several applications of **(Flat)**. Then there are two possible cases:

  a) $h_1 = f_1 \in FS$: Then we are done as $\forall a_i \in \overline{a}.|a_i| = \perp$ by the IH, and $|f_1(\overline{t_1})| = \perp$.

  b) $h_1 = c_1 \in DC$: Then $let\ \overline{X_1 = a_1}\ in\ let\ Y_1 = c_1(\overline{t_1})\ in\ h(Y_1) \to_l let\ \overline{X_1 = a_1}\ in\ h(c_1(\overline{t_1}))$ by **(Bind)**, and we are done as $\forall a_i \in \overline{a}.|a_i| = \perp$ by the IH.

  Using this techniques we can extend the proof to the case when h has more than one argument.

- $e = let\ X = e_1\ in\ e_2$: Let us assume $e_1 \notin \mathcal{V}$ and $e_2 \notin \mathcal{V}$, then we apply the IH to e_1 and e_2:

  $$\begin{array}{l} let\ X = e_1\ in\ e_2 \to_l^* \\ let\ X = (let\ \overline{X_1 = a_1}\ in\ h_1(\overline{t_1}))\ in \\ \quad (let\ \overline{X_2 = a_2}\ in\ h_2(\overline{t_2})) \to_l^* \text{ (by IH)} \\ let\ \overline{X_1 = a_1}\ in\ let\ X = h_1(\overline{t_1})\ in \\ \quad let\ \overline{X_2 = a_2}\ in\ h_2(\overline{t_2}) \text{ (by } \textbf{(Flat)} \text{ several times).} \end{array}$$

  Then there are two possible cases:

  a) $h_1 = f_1 \in FS$: Then we are done as $\forall a_i \in \overline{a_1} \cup \overline{a_2}.|a_i| = \perp$ by the IH, and $|f_1(\overline{t_1})| = \perp$.

  b) $h_1 = c_1 \in DC$: Then:

  $$\begin{array}{l} let\ \overline{X_1 = a_1}\ in\ let\ X = c_1(\overline{t_1})\ in \\ \quad let\ \overline{X_2 = a_2}\ in\ h_2(\overline{t_2}) \to_l^* \\ let\ \overline{X_1 = a_1}\ in\ let\ \overline{X_2 = a_2[X/c_1(\overline{t_1})]}\ in \\ \quad h_2(\overline{t_2})[X/c_1(\overline{t_1})] \end{array}$$

  by **(Bind)**, and we are done as for every $\theta \in CSubst$, $|e| = \perp$ implies $|e\theta| = \perp$ (easy to prove).

  Using this techniques we can extend the proof to the case when e_1 or e_2 are variables.

□

PROOF:[For Lemma 8] By induction on the size s of the *CRWL*-proof, that we measure as the number of *CRWL* rules applied:

Base Case: $s = 1$. Let us see which rule was applied:

B This contradicts the hypothesis because then $t \equiv \perp$, so we are done. In the rest of the proof we will assume that $t \not\equiv \perp$ because otherwise we would be in this case.

RR Then we have $\mathcal{P} \vdash_{CRWL} X \rightarrowtriangle X$. But then $X \rightarrow_l^0 X$ and $X \sqsubseteq X \equiv |X|$, so we are done with $\overline{X} = \emptyset$.

DC Then we have $\mathcal{P} \vdash_{CRWL} c \rightarrowtriangle c$. But then $c \rightarrow_l^0 c$ and $c \sqsubseteq c \equiv |c|$, so we are done with $\overline{X} = \emptyset$.

Inductive Step: $s > 1$. Let us see which rule was applied:

DC Then we have $e \equiv c(e_1, \dots, e_n)$ and the *CRWL*-proof has the form:

$$\frac{e_1 \rightarrowtriangle t_1, \dots, e_n \rightarrowtriangle t_n}{c(e_1, \dots, e_n) \rightarrowtriangle c(t_1, \dots, t_n)}\ DC$$

In the general case we can have $t_i = \perp$ for some i's and $t_j \neq \perp$ for the remaining ones. For simplicity we consider the case the case $n = 2$ with $t_1 = \perp$ and $t_2 \neq \perp$ (it is easy to extend the result for the general case), we have $\mathcal{P} \vdash_{CRWL} c(e_1, e_2) \rightarrowtriangle c(\perp, t_2)$. Then by IH over the second argument we have $e_2 \rightarrow_l^* \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ t'_2$, with $t'_2 \in CTerm$, $|a_{2_i}| = \perp$ for every a_{2_i} and $|\mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ t'_2| = t'_2[\overline{X_2/\perp}] \sqsupseteq t_2$. So:

$$\begin{array}{ll}
c(e_1, e_2) \rightarrow_l^* c(e_1, \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ t'_2) & \text{by IH} \\
\rightarrow_l^* \mathit{let}\ Y = (\mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ t'_2)\ \mathit{in}\ c(e_1, Y) & \text{by (LetIn)} \\
\rightarrow_l^* \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ \mathit{let}\ Y = t'_2\ \mathit{in}\ c(e_1, Y)) & \text{by (Flat)}^* \\
\rightarrow_l^* \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ c(e_1, t'_2) & \text{by (Bind)}
\end{array}$$

Then there are several possible cases:

a) $e_1 = f_1(\overline{e_1})$: Then $\mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ c(f_1(\overline{e_1}), t'_2) \rightarrow_l \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ \mathit{let}\ Z = f_1(\overline{e_1})\ \mathit{in}\ c(Z, t'_2)$, by **(LetIn)**. So we are done as $|a_{2_i}| = \perp$ for every a_{2_i} by the IH, $|f_1(\overline{e_1})| = \perp$ and

$$\begin{array}{c}
|\mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ \mathit{let}\ Z = f_1(\overline{e_1})\ \mathit{in}\ c(Z, t'_2)| = \\
c(Z, t'_2)[\overline{X_2/\perp}, Z/\perp] \sqsupseteq c(\perp, t_2)
\end{array}$$

because $t'_2[\overline{X_2/\perp}] \sqsupseteq t_2$ by the IH, and Z is fresh and so does not appear in t'_2.

b) $e_1 = t'_1 \in CTerm$; Then we are done as $|a_{2_i}| = \perp$ for every a_{2_i} by the IH, and

$$|\mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ c(t'_1, t'_2)| = c(t'_1, t'_2)[\overline{X_2/\perp}] \sqsupseteq c(\perp, t_2)$$

because $t'_2[\overline{X_2/\perp}] \sqsupseteq t_2$ by the IH.

c) $e_1 = c_1(\overline{e_1}) \notin CTerm$: Then by Lemma 7, $c_1(\overline{e_1}) \rightarrow_l^* \mathit{let}\ \overline{X_1 = a_1}\ \mathit{in}\ c_1(\overline{t_1})$ such that $|a_{1_i}| = \perp$ for every a_{1_i}. But then:

$$\begin{array}{l}
\mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ c(c_1(\overline{e_1}), t'_2) \\
\rightarrow_l^* \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ c(\mathit{let}\ \overline{X_1 = a_1}\ \mathit{in} \\
\qquad c_1(\overline{t_1}), t'_2) \text{ (by Lemma 7)} \\
\rightarrow_l^* \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ \mathit{let}\ Y = (\mathit{let}\ \overline{X_1 = a_1}\ \mathit{in} \\
\qquad c_1(\overline{t_1}))\ \mathit{in}\ c(Y, t'_2) \text{ (by LetIn)} \\
\rightarrow_l^* \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ \mathit{let}\ \overline{X_1 = a_1}\ \mathit{in} \\
\qquad \mathit{let}\ Y = c_1(\overline{t_1})\ \mathit{in}\ c(Y, t'_2) \text{ (by Flat}^*\text{)} \\
\rightarrow_l^* \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ \mathit{let}\ \overline{X_1 = a_1}\ \mathit{in} \\
\qquad c(c_1(\overline{t_1}), t'_2) \text{ by (Bind), as } Y \text{ is fresh.}
\end{array}$$

Then we are done as $|a_{1_i}| = \perp$ for every a_{1_i} by Lemma 7, $|a_{2_i}| = \perp$ for every a_{2_i} by the IH, and

$$\begin{array}{l}
|\mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ \mathit{let}\ \overline{X_1 = a_1}\ \mathit{in}\ c(c_1(\overline{t_1}), t'_2)| \\
= c(c_1(\overline{t_1}), t'_2)[\overline{X_1/\perp}][\overline{X_2/\perp}] \sqsupseteq c(\perp, t_2)
\end{array}$$

because $t'_2[\overline{X_2/\perp}] \sqsupseteq t_2$ by the IH, and $\overline{X_1}$ are fresh and so do not appear in t'_2.

d) $e_1 = \mathit{let}\ X = e_{11}\ \mathit{in}\ e_{12}$: this case is impossible as in Lemma 8 we assume $e \in Term$, without lets!

OR If f has no arguments ($n = 0$) then we have:

$$\frac{r\theta \rightarrowtriangle t}{f \rightarrowtriangle t}\ OR$$

with $(f \rightarrowtriangle r\theta) \in [\mathcal{P}]_\perp$. Let us define $\theta' \in CSubst$ as the substitution which is equal to θ except that every $\perp$ introduced by θ is replaced with some constructor symbol or variable. Then $\theta \sqsubseteq \theta'$, so by Lemma 6 we have $\mathcal{P} \vdash_{CRWL} r\theta' \rightarrowtriangle t$ with a proof of the same size. But then appliying the IH to this proof we get $r\theta' \rightarrow_l^* \mathit{let}\ \overline{X = a}\ \mathit{in}\ t'$ under the conditions of the lemma. But then $f \rightarrow_l e\theta' \rightarrow_l^* \mathit{let}\ \overline{X = a}\ \mathit{in}\ t'$ applying (Fapp) in the first step, so we are done.

If $n > 0$, we will proceed as in the case for (DC), doing a preliminary version for $\mathcal{P} \vdash_{CRWL} f(e_1, e_2) \rightarrowtriangle t$ which can be easily extended for the general case. Then we have:

$$\frac{e_1 \rightarrowtriangle \perp\ \ e_2 \rightarrowtriangle t_2\ r \rightarrowtriangle t}{f(e_1, e_2) \rightarrowtriangle t}\ OR$$

such that $t_2 \not\equiv \perp$, and with $(f(p_1, p_2) = e)\theta \in [\mathcal{P}]_\perp$ such that $p_1\theta = \perp$, $p_2\theta = t_2$ and $e\theta = r$. Then applying the IH to $\mathcal{P} \vdash_{CRWL} e_2 \rightarrowtriangle t_2$ we get that $e_2 \rightarrow_l^* \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ t'_2$ such that $|a_{2_i}| = \perp$ for every a_{2_i} and $|\mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ t'_2| = t'_2[\overline{X_2/\perp}] \sqsupseteq t_2$. So:

$$\begin{array}{ll}
f(e_1, e_2) \rightarrow_l^* f(e_1, \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ t'_2) & \text{by the IH} \\
\rightarrow_l^* \mathit{let}\ Y = (\mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ t'_2)\ \mathit{in}\ f(e_1, Y) & \text{by (LetIn)} \\
\rightarrow_l^* \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ \mathit{let}\ Y = t'_2\ \mathit{in}\ f(e_1, Y) & \text{by (Flat)}^* \\
\rightarrow_l^* \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ f(e_1, t'_2) & \text{by (Bind)}
\end{array}$$

Then applying Lemma 7 we get

$$f(e_1, t'_2) \rightarrow_l^* \mathit{let}\ \overline{X_1 = a_1}\ \mathit{in}\ f(t'_1, t'_2)$$

such that $|a_{1_i}| = \perp$ for every a_{1_i}. Now as $t'_2[\overline{X_2/\perp}] \sqsupseteq t_2$ then $(t'_1, t'_2) \sqsupseteq (\perp, t_2)$, so by Lemma 17 there must exists $\theta' \in CSubst$ such that $\theta \sqsubseteq \theta'$ and $(p_1, p_2)\theta' = (t'_1, t'_2)$. Then by Lemma 6, as $\mathcal{P} \vdash_{CRWL} r \equiv e\theta \rightarrowtriangle t$ then $\mathcal{P} \vdash_{CRWL} e\theta' \rightarrowtriangle t$ with a proof of the same size. As $\theta' \in CSubst$ and $e \in Term$ (because it is part of the program) then $e\theta' \in Term$ and we can apply the IH to that Crwl-proof getting that $e\theta' \rightarrow_l^* \mathit{let}\ \overline{X = a}\ \mathit{in}\ t'$ such that $|a_i| = \perp$ for every a_i and $|\mathit{let}\ \overline{X = a}\ \mathit{in}\ t'| = t'[\overline{X/\perp}] \sqsupseteq t$. So:

$$\begin{array}{l}
\mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ f(e_1, t'_2) \\
\rightarrow_l^* \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ \mathit{let}\ \overline{X_1 = a_1}\ \mathit{in}\ f(t'_1, t'_2) \text{ (by Lemma 7)} \\
\rightarrow_l^* \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ \mathit{let}\ \overline{X_1 = a_1}\ \mathit{in}\ e\theta' \text{ (by Fapp)} \\
\rightarrow_l^* \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ \mathit{let}\ \overline{X_1 = a_1}\ \mathit{in} \\
\qquad \mathit{let}\ \overline{X = a}\ \mathit{in}\ t' \text{ by } 2^{nd} \text{ IH.}
\end{array}$$

Then $|a_{2_i}| = \perp$ for every a_{2_i} by IH, $|a_{1_i}| = \perp$ for every a_{1_i} by Lemma 7 and $|a_i| = \perp$ for every a_i by IH. As the variables in $\overline{X_1} \cup \overline{X_2}$ are fresh variables introducted by the *let*-calculus, none of those can appear in t. So $t'[\overline{X/\perp}] \sqsupseteq t$ implies that $\forall p \in O(t')$ such that $t'|_p = Y$ such that $Y \in \overline{X_1} \cup \overline{X_2}$ then $t|_p = \perp$. So $|\mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ \mathit{let}\ \overline{X_1 = a_1}\ \mathit{in}\ \mathit{let}\ \overline{X = a}\ \mathit{in}\ t'| = t'[\overline{X/\perp}][\overline{X_1/\perp}][\overline{X_2/\perp}] \sqsupseteq t$. □