# Narrowing for Non-Determinism with Call-Time Choice Semantics[*]

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

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
`fraguas@sip.ucm.es, jrodrigu@fdi.ucm.es, jaime@sip.ucm.es`

**Abstract.** In a recent work we have proposed *let*-rewriting, a simple one-step relation close to ordinary term rewriting but able, via local bindings, to express sharing of computed values. In this way, *let*-rewriting reflects the call-time choice semantics for non-determinism adopted by modern functional logic languages, where programs are rewrite systems possibly non-confluent and non-terminating. Equivalence with *CRWL*, a well known semantic framework for functional logic programming, was also proved. In this paper we extend that work providing a notion of *let*-narrowing which is adequate for call-time choice as proved by a lifting lemma for *let*-rewriting similar to Hullot's lifting lemma for ordinary rewriting and narrowing.

## 1 Introduction

Programs in functional-logic languages (see [7] for a recent survey) are constructor based term rewriting systems possibly non-confluent and non-terminating, as happens in the following example:

$$coin \rightarrow 0 \qquad repeat(X) \ \rightarrow \ X : repeat(X)$$
$$coin \rightarrow 1 \qquad heads(X : Y : Ys) \ \rightarrow \ (X, Y)$$

Here *coin* is a 0-ary non-deterministic function that can be evaluated to 0 or 1, and *repeat* introduces non-termination on the system. The presence of non-determinism enforces to make a decision about *call-time* (also called *singular)* or *run-time choice* (or *plural*) semantics [10, 15]. Consider for instance the expression $heads(repeat(coin))$:

- run-time choice gives $(0,0), (0,1), (1,0)$ and $(1,1)$ as possible results. Rewriting can be used for it as in the following derivation:

$$heads(repeat(coin)) \rightarrow heads(coin : coin : \ldots) \rightarrow$$
$$(coin, coin) \rightarrow (0, coin) \rightarrow (0, 1)$$

- under call-time choice we obtain only the values $(0,0)$ and $(1,1)$ (*coin* is evaluated only once and this value must be *shared*).

Modern functional-logic languages like Curry [8] or Toy [12] adopt call-time choice, as it seems more appropriate in practice. Although the classical theory of TRS (term rewriting systems) is the basis of some influential papers in the field, specially those related to *needed narrowing* [2], it cannot really serve as technical foundation if call-time choice is taken into account, because ordinary rewriting corresponds to run-time choice. This was a main motivation for the $CRWL$[1] framework [5, 6], that is considered an adequate semantic foundation (see [7]) for the paradigm.

From an intuitive point of view there must be a strong connection between $CRWL$ and classical rewriting, but this has not received much attention in the past. Recently, in [11] we have started to investigate such a connection, by means of *let-rewriting*, that enhances ordinary rewriting with explicit *let*-bindings to express sharing, in a similar way to what was done in [13] for the $\lambda$-calculus. A discussion of the reasons for having proposed *let*-rewriting instead of using other existing formalisms like term graph-rewriting [14, 3] or specific operational semantics for FLP [1] can be found in [11].

In this paper we extend *let*-rewriting to a notion of *let-narrowing*. Our main result will be a *lifting lemma* for *let*-narrowing in the style of Hullot's one for classical narrowing [9].

We do not pretend that *let*-narrowing as will be presented here can replace advantageously existing versions of narrowing like needed narrowing [2] or natural narrowing [4], which are well established as appropriate operational procedures for functional logic programs. *Let*-narrowing is more a complementary proposal: needed or natural narrowing express refined strategies with desirable optimality properties to be preserved in practice, but they need to be patched in implementations in order to achieve sharing (otherwise they are unsound for call-time choice). *Let*-rewriting and *let*-narrowing intend to be the theoretical basis of that patch, so that sharing needs not be left anymore out of the scope of technical works dealing with rewriting-based operational aspects of functional logic languages. Things are then well prepared for recasting in the future the needed or natural strategies to the framework of *let*-rewriting and narrowing.

The rest of the paper is organized as follows. Section 2 contains a short presentation of *let*-rewriting. Section 3 tackles our main goal, extending *let*-rewriting to *let*-narrowing and proving its soundness and completeness. Finally, Section 4 contains some conclusions and future lines of research. Omitted proofs can be found at `gpd.sip.ucm.es/fraguas/papers/longWLP07.pdf`.

## 2 Preliminaries

### 2.1 Constructor-based term rewrite systems

We consider a first order signature $\Sigma = CS \cup FS$, where $CS$ and $FS$ are two disjoint set of *constructor* and defined *function* symbols respectively, all them with associated arity. We write $CS^n$ ($FS^n$ resp.) for the set of constructor (function)

---

[1] $CRWL$ stands for $C$onstructor-based $Re\,W$riting $L$ogic.

symbols of arity $n$. We write $c, d, \ldots$ for constructors, $f, g, \ldots$ for functions and $X, Y, \ldots$ for variables of a numerable set $\mathcal{V}$. The notation $\overline{o}$ stands for tuples of any kind of syntactic objects.

The set $Exp$ of *expressions* is defined as $Exp \ni e ::= X \mid h(e_1, \ldots, e_n)$, where $X \in \mathcal{V}$, $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. The set $CTerm$ of *constructed terms* (or *c-terms*) is defined like $Exp$, 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 function symbols, while $CTerm$ stands for data terms representing values. We will write $e, e', \ldots$ for expressions and $t, s, \ldots$ for c-terms. The set of variables occurring in an expression $e$ will be denoted as $var(e)$.

We will frequently use *one-hole contexts*, defined as $Cntxt \ni \mathcal{C} ::= [\ ] \mid h(e_1, \ldots, \mathcal{C}, \ldots, e_n)$, with $h \in CS^n \cup FS^n$. The application of a context $\mathcal{C}$ to an expression $e$, written by $\mathcal{C}[e]$, is defined inductively as $[\ ][e] = e$ and $h(e_1, \ldots, \mathcal{C}, \ldots, e_n)[e] = h(e_1, \ldots, \mathcal{C}[e], \ldots, e_n)$.

The set *Subst* of *substitutions* consists of mappings $\theta : \mathcal{V} \longrightarrow Exp$, which extend naturally to $\theta : Exp \longrightarrow Exp$. We write $e\theta$ for the application of $\theta$ to $e$, and $\theta\theta'$ for the composition, defined by $X(\theta\theta') = (X\theta)\theta'$. The domain and range of $\theta$ 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 $W \subseteq \mathcal{V}$ we write by $\theta|_W$ the restriction of $\theta$ to $W$, and $\theta|_{\backslash D}$ is a shortcut for $\theta|_{(\mathcal{V} \backslash D)}$. We will sometimes write $\theta = \sigma[W]$ instead of $\theta|_W = \sigma|_W$. In most cases we will use c-substitutions $\theta \in CSubst$, verifying that $X\theta \in CTerm$ for all $X \in dom(\theta)$. We say that $e$ *subsumes* $e'$, and write $e \preceq e'$, if $e\theta = e'$ for some $\theta$; we write $\theta \preceq \theta'$ if $X\theta \preceq X\theta'$ for all variables $X$ and $\theta \preceq \theta'[W]$ if $X\theta \preceq X\theta'$ for all $X \in W$.

A *constructor-based term rewriting system* $\mathcal{P}$ (*CTRS*, also called *program* along this paper) is a set of c-rewrite rules of the form $f(\overline{t}) \to e$ where $f \in FS^n$, $e \in Exp$ and $\overline{t}$ is a linear $n$-tuple of c-terms, where linearity means that variables occur only once in $\overline{t}$. Notice that we allow $e$ to contain *extra variables*, i.e., variables not occurring in $\overline{t}$. Given a program $\mathcal{P}$, its associated rewrite relation $\to_{\mathcal{P}}$ is defined as: $\mathcal{C}[l\theta] \to_{\mathcal{P}} \mathcal{C}[r\theta]$ for any context $\mathcal{C}$, rule $l \to r \in \mathcal{P}$ and $\theta \in Subst$. Notice that in the definition of $\to_{\mathcal{P}}$ it is allowed for $\theta$ to instantiate extra variables to any expression. We write $\overset{*}{\to}_{\mathcal{P}}$ for the reflexive and transitive closure of the relation $\to_{\mathcal{P}}$. In the following, we will usually omit the reference to $\mathcal{P}$.

## 2.2 Rewriting with local bindings

In [11] we have proposed *let-rewriting* as an alternative rewriting relation for CTRS that uses *let*-bindings to get an explicit formulation of sharing, i.e., call-time choice semantics. Although the primary goal for this relation was to establish a closer relationship between classical rewriting and the *CRWL*-framework of [6], *let*-rewriting is interesting in its own as a simple one-step reduction mechanism for call-time choice. This relation manipulates *let-expressions*, defined as: $LExp \ni e ::= X \mid h(e_1, \ldots, e_n) \mid let\ X = e_1\ in\ e_2$, where $X \in \mathcal{V}$, $h \in CS \cup FS$, and $\overline{e}_1, \ldots, \overline{e}_n \in LExp$. The notation $let\ \overline{X = a}\ in\ e$ abbreviates

*let* $X_1 = a_1$ *in* ... *in let* $X_n = a_n$ *in* $e$. The notion of context is also extended to the new syntax: $\mathcal{C} ::= [\,] \mid let\ X = \mathcal{C}\ in\ e \mid let\ X = e\ in\ \mathcal{C} \mid h(\ldots, \mathcal{C}, \ldots)$.

*Free* and *bound* (or *produced*) variables of $e \in LExp$ are defined as:

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

We assume a variable convention according to which 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 $\theta$ 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:

$$X\theta = \theta(X);\quad h(\overline{e})\theta = h(\overline{e\theta});\quad (let\ X = e_1\ in\ e_2)\theta\ =\ (let\ X = e_1\theta\ in\ e_2\theta)$$

The *let*-rewriting relation $\rightarrow_l$ is shown in Figure 1. 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 (see [11]). Figure 2 contains a *let*-rewriting derivation for the expression *heads(repeat(coin))* using the program example of Sect. 1.

## 3   *Let*-narrowing

It is well known that in functional logic computations there are situations where rewriting is not enough, and must be lifted to some kind of *narrowing*, because the expression being reduced contains variables for which different bindings might produce different evaluation results. This might happen either because variables are already present in the initial expression to reduce, or due to the presence of extra variables in the program rules. In the latter case *let*-rewriting certainly works, but not in an effective way, since the parameter passing substitution 'magically' guesses the right values for those extra variables.

The standard definition of *narrowing* as a lifting of rewriting in ordinary TRS says (adapted to the notation of contexts): $\mathcal{C}[f(\overline{t})] \rightsquigarrow_\theta \mathcal{C}\theta[r\theta]$, if $\theta$ is a mgu

$$\textbf{(Contx)}\ \mathcal{C}[e] \rightarrow_l \mathcal{C}[e'], \quad \text{if } e \rightarrow_l e',\ \mathcal{C} \in \textit{Cntxt}$$

$$\textbf{(LetIn)}\ h(\ldots, e, \ldots) \rightarrow_l \textit{let } X = e \textit{ in } h(\ldots, X, \ldots)$$
$$\text{if } h \in CS \cup FS,\ e \text{ takes one of the forms } e \equiv f(\overline{e'}) \text{ with } f \in FS^n \text{ or}$$
$$e \equiv \textit{let } Y = e' \textit{ in } e'', \text{ and } X \text{ is a fresh variable}$$

$$\textbf{(Flat)}\ \textit{let } X = (\textit{let } Y = e_1 \textit{ in } e_2) \textit{ in } e_3 \rightarrow_l \textit{let } Y = e_1 \textit{ in } (\textit{let } X = e_2 \textit{ in } e_3)$$
$$\text{assuming that } Y \text{ does not appear free in } e_3$$

$$\textbf{(Bind)}\ \textit{let } X = t \textit{ in } e \rightarrow_l e[X/t], \quad \text{if } t \in \textit{CTerm}$$

$$\textbf{(Elim)}\ \textit{let } X = e_1 \textit{ in } e_2 \rightarrow_l e_2, \quad \text{if } X \text{ does not appear free in } e_2$$

$$\textbf{(Fapp)}\ f(t_1\theta, \ldots, t_n\theta) \rightarrow_l e\theta, \quad \text{if } f(t_1, \ldots, t_n) \rightarrow e \in \mathcal{P},\ \theta \in \textit{CSubst}$$

**Fig. 1.** Rules of *let*-rewriting

| | |
|---|---|
| $heads(repeat(coin)) \rightarrow_l$ | $\textbf{(LetIn)}$ |
| $\textit{let } X = repeat(coin) \textit{ in } heads(X) \rightarrow_l$ | $\textbf{(LetIn)}$ |
| $\textit{let } X = (\textit{let } Y = coin \textit{ in } repeat(Y)) \textit{ in } heads(X) \rightarrow_l$ | $\textbf{(Flat)}$ |
| $\textit{let } Y = coin \textit{ in let } X = repeat(Y) \textit{ in } heads(X) \rightarrow_l$ | $\textbf{(Fapp)}$ |
| $\textit{let } Y = 0 \textit{ in let } X = repeat(Y) \textit{ in } heads(X) \rightarrow_l$ | $\textbf{(Bind)}$ |
| $\textit{let } X = repeat(0) \textit{ in } heads(X) \rightarrow_l$ | $\textbf{(Fapp)}$ |
| $\textit{let } X = 0 : repeat(0) \textit{ in } heads(X) \rightarrow_l$ | $\textbf{(LetIn)}$ |
| $\textit{let } X = (\textit{let } Z = repeat(0) \textit{ in } 0 : Z) \textit{ in } heads(X) \rightarrow_l$ | $\textbf{(Flat)}$ |
| $\textit{let } Z = repeat(0) \textit{ in let } X = 0 : Z \textit{ in } heads(X) \rightarrow_l$ | $\textbf{(Fapp)}$ |
| $\textit{let } Z = 0 : repeat(0) \textit{ in let } X = 0 : Z \textit{ in } heads(X) \rightarrow_l$ | $\textbf{(LetIn, Flat)}$ |
| $\textit{let } U = repeat(0) \textit{ in let } Z = 0 : U \textit{ in let } X = 0 : Z \textit{ in } heads(X) \rightarrow_l$ | $\textbf{(Bind)}, \textbf{2}$ |
| $\textit{let } U = repeat(0) \textit{ in } heads(0 : 0 : U) \rightarrow_l$ | $\textbf{(Fapp)}$ |
| $\textit{let } U = repeat(0) \textit{ in } (0, 0) \rightarrow_l$ | $\textbf{(Elim)}$ |
| $(0, 0)$ | |

**Fig. 2.** A let-rewriting derivation

of $f(\overline{t})$ and $f(\overline{s})$, where $f(\overline{s}) \rightarrow r$ is a fresh variant of a rule of the TRS. We note that frequently the narrowing step is not decorated with the whole unifier $\theta$, but with its projection over the variables in the narrowed expression. The condition that the binding substitution $\theta$ is a mgu can be relaxed to accomplish with certain narrowing strategies like needed narrowing [2], which use unifiers but not necessarily most general ones.

This definition of narrowing cannot be directly translated as it is to the case of *let*-rewriting, for two important reasons. The first is not new: because of call-time choice, binding substitutions must be c-substitutions, as already happened in *let*-rewriting. The second is that produced variables (those introduced by **(LetIn)** and bound in a *let* construction) should not be narrowed, because their role is to express intermediate values that are evaluated at most once and shared, according to call-time choice. Therefore the value of produced variables should be

better obtained by evaluation of their binding expressions, and not by bindings coming from narrowing steps. Furthermore, to narrow on produced variables destroys the structure of *let*-expressions.

The following example illustrates some of the points above.

*Example.* Consider the following program over natural numbers (represented with constructors 0 and $s$):

$$0 + Y \rightarrow Y \qquad\qquad even(X) \rightarrow if \ (Y+Y == X) \ then \ true$$
$$s(X) + Y \rightarrow s(X+Y) \qquad if \ true \ then \ Y \rightarrow Y$$
$$0 == 0 \rightarrow true \qquad\qquad s(X) == s(Y) \rightarrow X == Y$$
$$0 == s(Y) \rightarrow false \qquad s(X) == 0 \rightarrow false$$
$$coin \rightarrow 0 \qquad\qquad\qquad coin \rightarrow s(0)$$

Notice that the rule for *even* has an extra variable $Y$. With this program, the evaluation of *even(coin)* by *let*-rewriting could start as follows:

$even(coin) \rightarrow_l let \ X = coin \ in \ even(X)$
$\rightarrow_l let \ X = coin \ in \ if \ Y + Y == X \ then \ true$
$\rightarrow_l^* let \ X = coin \ in \ let \ U = Y + Y \ in \ let \ V = (U == X) \ in \ if \ V \ then \ true$
$\rightarrow_l^* let \ U = Y + Y \ in \ let \ V = (U == 0) \ in \ if \ V \ then \ true$

Now, all function applications involve variables and therefore narrowing is required to continue the evaluation. But notice that if we perform classical narrowing in (for instance) *if V then true*, then the binding $\{V/true\}$ will be created thus obtaining *let U=Y+Y in let true=(U==0) in if true then true* which is not a legal expression of *LExp* (because of the binding *let true =(U==0)*). Something similar would happen if narrowing is done in $U == 0$. What is harmless is to perform narrowing in $Y + Y$, giving the binding $\{Y/0\}$ and the (local to this expression) result 0. Put in its context, we obtain now:

$$let \ U = 0 \ in \ let \ V = (U == 0) \ in \ if \ V \ then \ true$$
$$\rightarrow_l let \ V = (0 == 0) \ in \ if \ V \ then \ true$$
$$\rightarrow_l let \ V = true \ in \ if \ V \ then \ true$$
$$\rightarrow_l if \ true \ then \ true \rightarrow_l true$$

This example shows that *let*-narrowing *must protect produced variables* against bindings. To express this we could add to the narrowing relation a parameter containing the set of protected variables. Instead of that, we have found more convenient to consider a distinguished set $PVar \subset \mathcal{V}$ of *produced variables* $X_p, Y_p, \ldots$, to be used according to the following criteria: variables bound in a *let* expression must be of $PVar$ (therefore *let* expressions have the form *let $X_p$=e in e'*); the parameter passing c-substitution $\theta$ in the rule **(Fapp)** of *let*-rewriting replaces extra variables in the rule by c-terms not having variables of $PVar$; and rewriting (or narrowing) sequences start with initial expressions $e$ not having free occurrences of produced variables (i.e., $FV(e) \cap PVar = \emptyset$). Furthermore we will need the following notion:

**Definition 1 (Admissible substitutions).** *A substitution $\theta$ is called* admissible *iff $\theta \in CSubst$ and $(dom(\theta) \cup ran(\theta)) \cap PVar = \emptyset$.*

The one-step *let*-narrowing relation $e \leadsto_{\theta}^{l} e'$ (assuming a given program $\mathcal{P}$) is defined in Fig. 3. The rules *Elim, Bind, Flat, LetIn* of *let*-rewriting are kept untouched except for the decoration with the empty substitution $\epsilon$. The important rules are **(Contx)** and **(Narr)**. In **(Narr)**, $\theta \in CSubst$ ensures that call-time choice is respected; notice also that produced variables are non bound in the narrowing step (by *(ii)*), and that bindings for extra variables and for variables in the expression being narrowed cannot contain produced variables (by *(iii)*). Notice, however, that if $\theta$ is chosen to be a mgu (which is always possible) then the condition *(iii)* is always fulfilled. Notice also that not the whole $\theta$ is recorded in the narrowing step, but only its projection over the relevant variables, which guarantees that the annotated substitution is an admissible one. In the case of **(Contx)** notice that $\theta$ is either $\epsilon$ or is obtained by **(Narr)** applied to the inner $e$. By the conditions imposed over unifiers in **(Narr)**, $\theta$ does not bind any produced variable, including those in *let* expressions surrounding $e$, which guarantees that any piece of the form *let* $X_p = r$ *in* $\ldots$ occurring in $\mathcal{C}$ becomes *let* $X_p = r\theta$ *in* $\ldots$ in $\mathcal{C}\theta$ after the narrowing step.

---

**(Contx)** $\mathcal{C}[e] \leadsto_{\theta}^{l} \mathcal{C}\theta[e']$     if $e \leadsto_{\theta}^{l} e'$, $\mathcal{C} \in Cntxt$
**(Narr)** $f(\bar{t}) \leadsto_{\theta|_{FV(f(\bar{t}))}}^{l} r\theta$, for any fresh variant $(f(\bar{p}) \to r) \in \mathcal{P}$ and $\theta \in CSubst$
   such that:
   i) $f(\bar{t})\theta \equiv f(\bar{p})\theta$.
   ii) $dom(\theta) \cap PVar = \emptyset$.
   iii) $ran(\theta|_{\setminus FV(f(\bar{p}))}) \cap PVar = \emptyset$.
**(X)** $e \leadsto_{\epsilon}^{l} e'$     if $e \to_{l} e'$ using $\boldsymbol{X} \in \{Elim, Bind, Flat, LetIn\}$.

---

**Fig. 3.** Rules of *let*-narrowing

The one-step relation $\leadsto_{\theta}^{l}$ is extended in the natural way to the multiple-steps narrowing relation $\leadsto^{l^*}$, which is defined as the least reflexive relation verifying:

$$e \leadsto_{\theta_1}^{l} e_1 \leadsto_{\theta_2}^{l} \ldots e_n \leadsto_{\theta_n}^{l} e' \Rightarrow e \leadsto_{\theta_1 \ldots \theta_n}^{l^*} e'$$

We write $e \leadsto_{\theta}^{l^n} e'$ for a n-steps narrowing sequence.

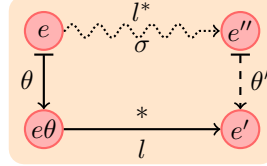### 3.1 Soundness and completeness of *let*-narrowing

In this section we show the adequacy of *let*-narrowing wrt *let*-rewriting. We assume a fixed program $\mathcal{P}$. We start by proving *soundness*, as stated in the following result:

**Theorem 1 (Soundness of *let*-narrowing).** *For any $e, e' \in LExp$, $e \leadsto_{\theta}^{l^*} e'$ implies $e\theta \to_{l}^{*} e'$.*

Completeness is, as usual, more complicated to prove. The key result is the following generalization to the framework of *let*-rewriting of Hullot's *lifting lemma* [9] for classical rewriting and narrowing, stating that any rewrite

sequence for a particular instance of an expression can be generalized by a narrowing derivation.

**Lemma 1 (Lifting lemma for *let*-rewriting).** *Let $e, e' \in LExp$ such that $e\theta \to_l^* e'$ for an admissible $\theta$, and let $\mathcal{W}$ be a set of variables with $dom(\theta) \cup FV(e) \subseteq \mathcal{W}$. Then there exist a* let-*narrowing derivation $e \rightsquigarrow_\sigma^{l^*} e''$ and an admissible $\theta'$ such that $e''\theta' = e'$ and $\sigma\theta' = \theta[\mathcal{W}]$. Besides, the* let-*narrowing derivation can be chosen to use mgu's at each* **(Narr)** *step. Graphically:*



As an immediate corollary we obtain the following completeness result of *let*-narrowing for finished *let*-rewriting derivations:

**Theorem 2 (Completeness of *let*-narrowing).**
*Let $e \in LExp, t \in CTerm$ and $\theta$ an admissible c-substitution. If $e\theta \to_l^* t$, then there exist a* let-*narrowing derivation $e \rightsquigarrow_\sigma^{l^*} t'$ and an admissible $\theta'$ such that $t'\theta' = t$ and $\sigma\theta' = \theta[FV(e)]$.*

## 3.2  *Let*-narrowing versus narrowing for deterministic systems

The relationship between *let*-rewriting ($\to_l$) and ordinary rewriting ($\to$) is examined in [11], where $\to_l$ is proved to be sound wrt $\to$, and complete for the class of *deterministic* programs, a notion close but not equivalent to confluence (see [11] for the technical definition).

The class of deterministic programs is conjectured in [11] to be wider that that of confluent programs, and it certainly contains all inductively sequential programs (without extra variables). The following result holds (see [11]):

**Theorem 3.** *Let $\mathcal{P}$ be any program, $e \in Exp, t \in CTerm$. Then:*

**(a)** $e \to_l^* t$ *implies $e \to^* t$.*
**(b)** *If in addition $\mathcal{P}$ is deterministic, then the reverse implication holds.*

Joining this with the results of the previous section we can easily establish some relationships between *let*-narrowing and ordinary rewriting/narrowing, as follows (we assume here that all involved substitutions are admissible):

**Theorem 4.** *Let $\mathcal{P}$ be any program, $e \in Exp, \theta \in CSubst, t \in CTerm$. Then:*

**(a)** $e \rightsquigarrow_\theta^{l^*} t$ *implies $e\theta \to^* t$.*
**(b)** *If in addition $\mathcal{P}$ is deterministic, then:*
  **($b_1$)** *If $e\theta \to^* t$, there exist $t' \in CTerm$, $\sigma, \theta' \in CSubst$ such that $e \rightsquigarrow_\sigma^{l^*} t'$, $t'\theta' = t$ and $\sigma\theta' = \theta[var(e)]$ (and therefore $t' \preceq t, \sigma \preceq \theta[var(e)]$).*
  **($b_2$)** *If $e \rightsquigarrow_\theta^* t$, the same conclusion of ($b_1$) holds.*

Part *(a)* expresses soundness of $\rightsquigarrow^l$ wrt rewriting, and part *(b)* is a completeness result for $\rightsquigarrow^l$ wrt rewriting/narrowing, for the class of deterministic programs.

# 4  Conclusions

Our aim in this work was to progress in the effort of filling an existing gap in the functional logic programming field, where up to recently there was a lack of a simple and abstract enough one-step reduction mechanism close enough to ordinary rewriting but at the same time respecting non-strict and call-time choice semantics for possibly non-confluent and non-terminating constructor-based rewrite systems (possibly with extra variables), and trying to avoid the complexity of graph rewriting [14]. These requirements were not met by two well established branches in the foundations of the field: one is the *CRWL* approach, very adequate from the point of view of semantics but operationally based on somehow complicated narrowing calculi [6, 16] too far from the usual notion of term rewriting. The other approach focuses on operational aspects in the form of efficient narrowing strategies like needed narrowing [2] or natural narrowing [4], based on the classical theory of rewriting, sharing with it the major drawback that rewriting is an *unsound* operation for call-time choice semantics of functional logic programs. There have been other attempts of coping operationally with call-time choice [1], but relying in too low-level syntax and operational rules.

In a recent work [11] we established a technical bridge between both approaches (*CRWL*/classical rewriting) by proposing a notion of rewriting with sharing by means of local *let* bindings, in a similar way to what has been done for sharing and lambda-calculus in [13]. Most importantly, we prove there strong equivalence results between *CRWL* and *let*-rewriting.

Here we continue that work by contributing a notion of *let*-narrowing (narrowing for sharing) which we prove sound and complete with respect to *let*-rewriting. We think that *let*-narrowing is the simplest proposed notion of narrowing that is close to the usual notions of TRS and at the same time is proved adequate for call-time choice semantics. The main technical insight for let-narrowing has been the need of protecting produced (locally bound) variables against narrowing over them. We have also proved soundness of *let*-narrowing wrt ordinary rewriting and completeness for the wide class of deterministic programs, thus giving a technical support to the intuitive fact that combining sharing with narrowing does not create new answers when compared to classical narrowing, and at the same time does not lose answers in case of deterministic systems. As far as we know these results are new in the narrowing literature.

The natural extension of our work will be to add strategies to *let*-rewriting and *let*-narrowing, an issue that has been left out of this paper but is needed as foundation of effective implementations. But we think that the clear script we have followed so far (first presenting a notion of rewriting with respect to which we have been able to prove correctness and completeness of a a subsequent notion of narrowing, to which add strategies in future work) is an advantage rather than a lack of our approach.

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

3. 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.

4. S. Escobar, J. Meseguer, and P. Thati. Natural narrowing for general term rewriting systems. In *RTA*, pages 279–293, 2005.

5. 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.

6. 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.

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

8. 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.

9. J. Hullot. Canonical forms and unification. In *Proc. 5th Conference on Automated Deduction*, pages 318–334. Springer LNCS 87, 1980.

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

11. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proc. Principles and Practice of Declarative Programming, ACM Press*, pages 197–208, 2007.

12. 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.

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

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

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

16. 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.