Rewriting and Call-Time Choice: The HO Case*

Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá,
and Jaime 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. It is known that the behavior of non-deterministic functions with call-time choice semantics, present in current functional logic languages, is not well described by usual approaches to reduction like ordinary term rewriting systems or λ-calculus. The presence of HO features makes things more difficult, since reasoning principles that are essential in a standard (i.e., deterministic) functional setting, like extensionality, become wrong. In this paper we propose *HOlet*-rewriting, a notion of rewriting with local bindings that turns out to be adequate for programs with HO non-deterministic functions, as it is shown by strong equivalence results with respect to *HOCRWL*, a previously existing semantic framework for such programs. In addition, we give a sound and complete notion of *HOlet*-narrowing, we show by a case study the usefulness of the achieved combination of semantic and reduction notions, and finally we prove within our framework that a standard approach to the implementation of HO features, namely translation to FO, is still valid for HO nondeterministic functions.

1 Introduction

Functional logic programming (FLP, for short; see [12,14] for surveys) integrates features of logic programming and functional programming. Typically FLP adopts mostly a (lazy) functional style, thus making intensive use of higher order (HO) functions. However, most of the work about FLP focuses on first order (FO) aspects of programs, thus limiting the applicability of results.

This is not a satisfactory situation, especially taking into account that the presence of functions that are at the same time HO and non-deterministic leads to somehow surprising behaviors, as shown by the example we sent recently to the Curry mailing list [13]:

Example 1. Consider the following program computing with natural numbers represented by the constructors 0 and $s/1$, and where + is defined as usual.

```
g X -> 0          f -> g          f' X -> f X
h X -> s 0        f -> h

fadd F G X -> (F X) + (G X)       fdouble F -> fadd F F
```

* This work has been partially supported by the Spanish projects Merit-Forms-UCM (TIN2005-09207-C03-03) and Promesas-CAM (S-0505/TIC/0407).

J. Garrigue and M. Hermenegildo (Eds.): FLOPS 2008, LNCS 4989, pp. 147–162, 2008.

Notice that f and f' are non-deterministic functions that are (by definition of f') extensionally equivalent; from the point of view of standard functional programming they should be seen as 'the same function'. However, consider the expressions *(fdouble f 0)* and *(fdouble f' 0)*. In modern FLP languages like Curry [16] or Toy [21], the possible values for *(fdouble f 0)* are *0, s (s 0)*, while *(fdouble f' 0)* can be in addition reduced to *s 0*.

This behavior corresponds to *call-time choice* [17,11], the semantics for non-determinism adopted by those systems. Operationally call-time choice is very close to the *sharing* mechanism used in functional languages to implement lazy evaluation.

The example was sent[1] to point out that η-expansion and η-reduction are not valid for such systems, because extensionally equivalent functions (e.g., f and f') can be semantically distinguishable when put in the same context (e.g., *double* [ ] 0), a fact that does not happen neither in standard (i.e, deterministic) functional programs[2], nor in FO FLP. We remark also that with *run-time choice* [17,11], f and f' will be indistinguishable (*double f 0* and *double f' 0* would both produce *0, s 0, s (s 0)* as possible results). Therefore, it is the combination *HO + Non-determinism + call-time choice* which makes things different.

That combination was addressed in *HOCRWL* [7,8], an extension to HO of *CRWL*[3] [11], a semantic framework specifically devised for FLP with call-time choice semantics for non-determinism (see [28] for a survey of *CRWL* and its extensions). *HOCRWL* provides logic and model-theoretic semantics, based on an *intensional* view of functions, where different descriptions –in the form of *HO-patterns*– of the same extensional function are distinguished as different data. This allows expressive programs and is simpler than λ-calculus-based HO unification, which is an alternative approach followed in the logic programming setting [23]. Previous work on the intensional view of HO-FLP [10] did not consider non-determinism. Other works covering HO in FLP, [24,15], consider orthogonal or inductively sequential (henceforth deterministic) systems; if extended directly to the non-deterministic case, they would realize run-time choice, as happens also with [4], where a type-based translation to FO in the spirit of [29,9] is proposed. We remark also that [15] is close to the theory of HO rewriting [27], and therefore has η-expansion as a valid procedure, against the expected properties of the languages considered by ours. Finally, [1] copes with call-time choice but their approach to HO is again based on a FO-translation, in contrast to ours.

A weak point of the original *(HO)CRWL*-way to FLP is that it does not come with a clear, simple notion of one-step reduction similar to one-step rewriting. In [19] we proposed *let-rewriting*, a notion of rewriting with local bindings adequate to FO *CRWL* semantics, and at the same time simpler and more abstract than other reduction notions based on term graph rewriting [26,6] or natural operational semantics [1]. *Let*-rewriting was generalized to *let*-narrowing in [18].

[1] As far as we know, it was the first time that this behavior was noticed.

[2] Although the addition of primitive functions not definable in the language like *seq* in Haskell [25] can also destroy extensionality.

[3] CRWL stands for *Constructor Based Rewriting Logic*.

Our aim in this work is to extend the notion of *let*-rewriting/narrowing to the HO case. We address various foundational aspects –definition of *HOlet*-rewriting and equivalence wrt the declarative semantics given by *HOCRWL* (Sect. 3), *HOlet*-narrowing and its soundness and completeness wrt *HOlet*-rewriting (Sect. 4)– and also more applied aspects, as are the use of our framework to language development (Sect. 5) or the proof of correctness within our framework of a scheme of translation to FO, the basis of a standard approach [29,9,4] to the implementation of HO stuff in FO settings.

There are still some other important issues –evaluation strategies (including concurrency), types, constraints– that have been left out of the scope of the paper. Finally, we are not inventing HO FLP, but only contributing to some aspects of its foundation. Therefore it is not our aim in this paper convincing of the practical interest of HO FLP: other documents [16,28,7,4] contain enough evidences of that. Omitted proofs can be found in [20].

2 Preliminaries: *HOCRWL*

We present here some basic notions and new results about *HOCRWL* [7].

2.1 Expressions, Patterns and Programs

We consider *function* symbols $f, g, \ldots \in FS$, *constructor* symbols $c, d, \ldots \in CS$, and *variables* $X, Y, \ldots \in \mathcal{V}$; each $h \in FS \cup CS$ has an associated *arity*, $ar(h) \in \mathbb{N}$; FS^n (resp. CS^n) is the set of function (resp. constructor) symbols with arity n. The notation $\overline{o}$ stands for tuples of any kind of syntactic objects o. The set of *applicative expressions* is defined by $Exp \ni e ::= X \mid h \mid (e_1\ e_2)$. As usual, application is left associative and outer parentheses can be omitted, so that $e_1\ e_2\ \ldots e_n$ stands for $((\ldots(e_1\ e_2)\ldots)\ e_n)$. The set of variables occurring in e is written by $var(e)$. A distinguished set of expressions is that of *patterns* $t, s \in Pat$, defined by: $t ::= X \mid c\ t_1 \ldots t_n \mid f\ t_1 \ldots t_m$, where $0 \leq n \leq ar(c), 0 \leq m < ar(f)$. Patterns are irreducible expressions playing the role of *values*. *FO-patterns*, defined by $FOPat \ni t ::= X \mid c\ t_1 \ldots t_n\ (n = ar(c))$, correspond to FO constructor terms, representing ordinary non-functional data-values. Partial applications of symbols $h \in FS \cup CS$ to other patterns are HO-patterns and can be seen as truly data-values representing functions from an *intensional* point of view. Examples of patterns with the signature of Ex. 1 are: *0, s X, s, f', fadd f' f'*. The last three are HO-patterns. Notice that *f, fadd f f* are not patterns since f is not a pattern ($ar(f) = 0$).

Expressions $X\ e_1 \ldots e_m$ ($m \geq 0$) are called *flexible* (*variable application* when $m > 0$). *Rigid* expressions have the form $h\ e_1 \ldots e_m$; moreover, they are *junk* if $h \in CS^n$ and $m > n$, *active* if $h \in FS^n$ and $m \geq n$, and *passive* otherwise.

Contexts are expressions with a hole defined as $Cntxt \ni \mathcal{C} ::= [\,] \mid \mathcal{C}\ e \mid e\ \mathcal{C}$. Application of $\mathcal{C}$ to e (written $\mathcal{C}[e]$) is defined by $[\,][e] = e$; $(\mathcal{C}\ e')[e] = \mathcal{C}[e]\ e'$; $(e'\ \mathcal{C})[e] = e'\ \mathcal{C}[e]$. Substitutions $\theta \in Subst$ are finite mappings from variables to expressions; $[X_i/e_i, \ldots, X_n/e_n]$ is the substitution which assigns $e_i \in Exp$ to the corresponding $X_i \in \mathcal{V}$. We will mostly use *pattern-substitutions*

$PSubst = \{\theta \in Subst \mid \theta(X) \in Pat, \forall X \in \mathcal{V}\}$. We write ϵ for the identity substitution, $dom(\theta)$ for the domain of θ, and $vRan(\theta) = \bigcup_{X \in dom(\theta)} var(X\theta)$.

As usual while describing semantics of non-strict languages, we enlarge the signature with a new 0-ary constructor symbol $\bot$, which can be used to build the sets $Expr_\bot, Pat_\bot, PSubst_\bot$ of *partial* expressions, patterns and p-substitutions resp. Partial expressions are ordered by the *approximation* ordering $\sqsubseteq$ defined as the least partial ordering satisfying $\bot \sqsubseteq e$ and $e \sqsubseteq e' \Rightarrow \mathcal{C}[e] \sqsubseteq \mathcal{C}[e']$ for all $e, e' \in Exp_\bot, \mathcal{C} \in Cntxt$. This partial ordering can be extended to substitutions: given $\theta, \sigma \in Subst_\bot$ we say $\theta \sqsubseteq \sigma$ if $X\theta \sqsubseteq X\sigma$ for all $X \in \mathcal{V}$.

A *HOCRWL*-program (or simply a *program*) consists of one or more *program rules* for each $f \in FS^n$, having the form $f\ t_1 \ldots t_n \to r$ where $(t_1, \ldots, t_n)$ is a linear (i.e. variables occur only once) tuple of (maybe HO) patterns and r is any expression. Notice that confluence or termination is not required, and that r may have variables not occurring in $f\ t_1 \ldots t_n$ (we write $vExtra(R)$ for such variables in a rule R). The original *HOCRWL* logic considered also *joinability* conditions in rules to achieve a better treatment of strict equality as built-in, which is a subject orthogonal to the aims of this paper. Therefore, we consider only unconditional rules.

Some related languages, like Curry, do not allow HO-patterns in left-hand sides of function definitions. We remark that all the notions and results in the paper are applicable to programs with this restriction and we stress the fact that Example 1 is one of them.

Given a program $\mathcal{P}$, the set of its rule instances is $[\mathcal{P}] = \{(l \to r)\theta \mid (l \to r) \in \mathcal{P}, \theta \in PSubst\}$. The set $[\mathcal{P}]_\bot$ is defined similarly replacing $PSubst$ by $PSubst_\bot$. To require $\theta \in PSubst_{(\bot)}$ instead of $\theta \in Subst_{(\bot)}$ is essential to achieve call-time choice in the next sections.

2.2 The *HOCRWL* Proof Calculus [7]

The semantics of a program $\mathcal{P}$ is determined in *HOCRWL* by means of a proof calculus able to derive reduction statements of the form $e \twoheadrightarrow t$, with $e \in Exp_\bot$ and $t \in Pat_\bot$, meaning informally that t is (or approximates to) a possible value of e, obtained by evaluation of e using $\mathcal{P}$ under call-time choice. Besides this logical semantics, *HOCRWL* programs come in [7] with a model-theoretic semantics based on applicative algebras, with existence of a least Herbrand model. We will not use this aspect of the semantics here.

The *HOCRWL*-proof calculus is presented in Fig. 1. We write $\mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t$ to express that $e \twoheadrightarrow t$ is derivable in that calculus using the program $\mathcal{P}$. The *HOCRWL-denotation* of an expression $e \in Exp_\bot$ is defined as $[\![e]\!]^{\mathcal{P}}_{HOCRWL} = \{t \in Pat_\bot \mid \mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t\}$. $\mathcal{P}$ and *HOCRWL* are frequently omitted in those notations.

In Example 1 we have $[\![f\,double\ f\ 0]\!] = \{0, s\ (s\ 0), \bot, s\ \ \bot, s\ (s\ \ \bot)\}$ and $[\![f\,double\ f'\ 0]\!] = \{0, s\ 0, s\ (s\ 0), \bot, s\ \ \bot, s\ (s\ \ \bot)\}$.

We will use the following (new) result stating an important compositionality property of the semantics of *HOCRWL*-expressions: the semantics of a whole expression depends only on the semantics of its constituents, in a particular form

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

$$\textbf{(DC)}\ \frac{e_1 \twoheadrightarrow t_1\ \ldots\ e_n \twoheadrightarrow t_m}{h\ e_1 \ldots e_m \twoheadrightarrow h\ t_1 \ldots t_m} \quad h \in \Sigma,\ \text{if } h\ t_1 \ldots t_m \text{ is a partial pattern, } m \geq 0$$

$$\textbf{(OR)}\ \frac{e_1 \twoheadrightarrow t_1 \ldots\ e_n \twoheadrightarrow t_n\ \ r\ a_1 \ldots a_m \twoheadrightarrow t}{f\ e_1 \ldots e_n\ a_1 \ldots a_m \twoheadrightarrow t} \quad \text{if } m \geq 0, (f\ t_1 \ldots t_n \to r) \in [\mathcal{P}]_\bot$$

Fig. 1. (*HOCRWL*-calculus)

reflecting the idea of call-time choice. The second part of the theorem is a technical result, needed in some proofs, concerning the size of the involved derivations.

Theorem 1 (Compositionality of *HOCRWL* semantics)

(i) $[\![\mathcal{C}[e]]\!] = \bigcup_{t \in [\![e]\!]} [\![\mathcal{C}[t]]\!]$, *for any program* $\mathcal{P}$ *and expression* $e \in Exp_\bot$. *In other terms,* $\mathcal{C}[e] \twoheadrightarrow t \Leftrightarrow \exists s.(e \twoheadrightarrow s \wedge \mathcal{C}[s] \twoheadrightarrow t)$.

(ii) In the ($\Rightarrow$) *part of (i), if* $t \neq \bot, \mathcal{C} \neq [\ ]$ *and the derivation of* $\mathcal{C}[e] \twoheadrightarrow t$ *has size* K, *then the derivations of* $e \twoheadrightarrow s$ *and* $\mathcal{C}[s] \twoheadrightarrow t$ *can be chosen with sizes* $< K$ *and* $\leq K$ *respectively.*

3 Higher Order *let*-rewriting

To express sharing, as is required for call-time choice, we enhance the syntax of expressions (and contexts) with a *let* construct for local bindings, in the spirit of [5,22,19]: $LExp \ni e ::= X \mid h \mid e_1\ e_2 \mid let\ X = e_1\ in\ e_2$

$Cntxt \ni \mathcal{C} ::= [\ ] \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid let\ X = \mathcal{C}\ in\ e \mid let\ X = e\ in\ \mathcal{C}$

We consider expressions $let\ X = e_1\ in\ e_2$ as passive and rigid. The sets $FV(e)$ and $BV(e)$ of free and bound variables resp. of a *let*-expression e 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 $FV(let\ X = e_1\ in\ e_2)$ recursive *let*-bindings are not allowed since the possible occurrences of X in e_1 are not considered as bound and therefore refer to a 'different' X. We assume appropriate renamings of bound variables ensuring that bound and free variables are kept distinct, and that whenever θ is applied to $e \in LExp$, $BV(e) \cap (dom(\theta) \cup vRan(\theta)) = \emptyset$, so that $(let\ X = e_1\ in\ e_2)\theta = let\ X = e_1\theta\ in\ e_2\theta$ and $(\mathcal{C}[e])\theta = \mathcal{C}\theta[e\theta]$.

The *shell* of an expression, written as $|e|$, is a pattern containing the 'stable' outer information of e, not to be destroyed by reduction:

$$|X\ e_1 \ldots e_m| = \begin{cases} X & \text{if } m = 0 \\ \bot & \text{if } m > 0 \end{cases}$$

$$|h\ e_1 \ldots e_m| = \begin{cases} h\ |e_1| \ldots |e_m| & \text{if } (h \in CS^n, m \leq n) \text{ or } (h \in FS^n, m < n) \\ \bot & \text{otherwise (junk or active expression)} \end{cases}$$
$$|(let\ X = e_1\ in\ e_2)\ a_1 \ldots a_m| = |(e_2[X/e_1])\ a_1 \ldots a_m|$$

Notice that in FO [19] we defined $|(let\ X = e_1\ in\ e_2)| = |e_2|[X/|e_1|]$. This would lose information in the HO case: for instance, $|let\ X = s\ in\ X\ 0|$ would be $\bot$, instead of the more accurate *s 0* given by the definition above.

The $HOCRWL_{let}$ proof calculus for proving statements $e \twoheadrightarrow t$ ($e \in LExp_\bot, t \in Pat_\bot$) results from adding to Fig. 1 the rule:

$$\textbf{(Let)}\quad \frac{e_1 \twoheadrightarrow t_1 \quad (e_2[X/t_1])\ a_1 \ldots a_m \twoheadrightarrow t}{(let\ X = e_1\ in\ e_2)\ a_1 \ldots a_m \twoheadrightarrow t} \qquad (m \geq 0)$$

It is easy to see that for programs and expressions without *let*s both calculi coincide, giving $[\![e]\!]_{HOCRWL} = [\![e]\!]_{HOCRWL_{let}}$, and then we write simply $[\![e]\!]$.

Theorem 1 does not hold as it is for *let*-expressions (assume, for instance, the program rule *f 0 = 1* and take $e \equiv f\ X$, $\mathcal{C} \equiv let\ X{=}0\ in\ [\,]$). However, a more limited form of compositionality will suffice to our needs:

Theorem 2 (Weak compositionality of $HOCRWL_{let}$ semantics)
For any $\mathcal{P}$ *and* $e, e' \in LExp_\bot$: $[\![C\ [e]]\!] = \bigcup_{t \in [\![e]\!]} [\![C\ [t]]\!]$, *if* $BV(C) \cap FV(e) = \emptyset$.

As a consequence, *(i)* $[\![e\ e']\!] = \bigcup_{t \in [\![e]\!]} [\![t\ e']\!]$ *(ii)* $[\![e\ e']\!] = \bigcup_{t \in [\![e']\!]} [\![e\ t]\!]$
(iii) $[\![let\ X = e\ in\ e']\!] = \bigcup_{t \in [\![e]\!]} [\![e'[X/t]]\!]$

3.1 Rewriting with Local Bindings

Figure 2 defines the *HOlet*-rewriting relation $\rightarrow^l$. Rule *(Fapp)* uses a program rule to reduce a function application, but only when the arguments are already patterns, otherwise call-time choice would be violated. Non-pattern arguments of applications are moved to local bindings by *(LetIn)*. Local bindings of patterns to variables are applied in *(Bind)*, since in this case copying is harmless. *(Elim)* erases useless bindings. *(Flat)* and *(LetAp)* manage local bindings; they are needed to avoid some reductions to get stuck. Notice that with the variable convention, the condition $Y \notin FV(e_3)$ in *(Flat)* and *(LetAp)* would not be needed; we have written it in order to keep the rules independent of the convention. Finally, any of these rules can be applied to any subexpression by *(Contx)*. It includes an additional technical condition to avoid undesired variable captures when *(Fapp)* was applied inside a surrounding context and the used program rule has extra variables. If, for instance, a program rule is $f \rightarrow Y$, the rule *(Contxt)* avoids the step *let X=0 in f* $\rightarrow^l$ *let X=0 in X* and also the step *let X=f in X* $\rightarrow^l$ *let X=X in X*.

The following derivation corresponds to Example 1:

fdouble f 0 $\rightarrow^l_{\{LetIn,Cntx\}}$ *(let F=f in fdouble F) 0*
$\rightarrow^l_{LetAp}$ *let F=f in fdouble F 0* $\rightarrow^l_{\{Fapp,Cntx\}}$ *let F=f in fadd F F 0*
$\rightarrow^l_{\{Fapp,Cntx\}}$ *let F=f in F 0 + F 0*
$\rightarrow^l_{\{Fapp,Cntx\}}$ *let F=g in F 0 + F 0* $\rightarrow^l_{Bind}$ *g 0 + g 0* $\rightarrow^{l*}$ *0*

(Fapp) $f\ t_1 \ldots t_n \rightarrow^l r$, if $(f\ t_1 \ldots t_n \rightarrow r) \in [\mathcal{P}]$

(LetIn) $e_1\ e_2 \rightarrow^l let\ X = e_2\ in\ e_1\ X$ (X fresh), if e_2 is an active expression, variable application, junk or *let* rooted expression.

(Bind) $let\ X = t\ in\ e \rightarrow^l e[X/t]$, if $t \in Pat$

(Elim) $let\ X = e_1\ in\ e_2 \rightarrow^l e_2$, if $X \notin FV(e_2)$

(Flat) $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \rightarrow^l let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$ if $Y \notin FV(e_3)$

(LetAp) $(let\ X = e_1\ in\ e_2)\ e_3 \rightarrow^l let\ X = e_1\ in\ e_2\ e_3$, if $X \notin FV(e_3)$

(Contx) $\mathcal{C}[e] \rightarrow^l \mathcal{C}[e']$, if $\mathcal{C} \neq [\ ]$, $e \rightarrow^l e'$ using any of the previous rules, and in case $e \rightarrow^l e'$ is a (Fapp) step using $(f\ \overline{p} \rightarrow r)\theta \in [\mathcal{P}]$ then $vRan(\theta|_{\setminus var(\overline{p})}) \cap BV(\mathcal{C}) = \emptyset$.

Fig. 2. Higher order *let*-rewriting relation $\rightarrow^l$

Notice that the fist step is justified because f is active. In contrast, since f' is a pattern, a derivation for *fdouble f' 0* could proceed as follows:

$$fdouble\ f'\ 0 \rightarrow^l fadd\ f'\ f'\ 0 \rightarrow^l f'\ 0\ +\ f'\ 0 \rightarrow^{l^*} f\ 0\ +\ f\ 0 \rightarrow^{l^*} g\ 0\ +\ h\ 0 \rightarrow^{l^*} s\ 0$$

The rules of $\rightarrow^l$ have been carefully tuned up to ensure that program rules are the only possible source of non-termination, as ensured by the following result.

Proposition 1. *The relation* $\rightarrow^l{}_{\setminus Fapp}$ *defined by the rules of Fig. 2 except* (Fapp) *is terminating.*

This is a natural requirement. However, at some point we will find useful to consider the more liberal relation $\rightarrow^L$ obtained replacing *(LetIn)* by:

$$\textbf{(LetIn')} \quad e_1\ e_2 \rightarrow^L let\ X = e_2\ in\ e_1\ X \quad (X \text{ fresh})$$

which is less restrictive (then $\rightarrow^l\ \subseteq\ \rightarrow^L$). However $\rightarrow^L{}_{\setminus Fapp}$ becomes non-terminating, as shown by: $s\ 0 \rightarrow^l{}_{LetIn'} let\ X = 0\ in\ s\ X \rightarrow^l{}_{Bind} s\ 0 \rightarrow^l \ldots$

3.2 Adequacy of *HOlet*-rewriting to *HOCRWL*

We compare here $\rightarrow^l$ to *HOCRWL*-derivability $\twoheadrightarrow$, proving that essentially $\rightarrow^l$ gives no more (*soundness*) and no less (*completeness*) results than $\twoheadrightarrow$.

As in [19], the following notion is useful to establish soundness:

Definition 1 (Hypersemantics)

(i) The hypersemantics *of an expression* $e \in LExp_\bot$*, written as* $[\![e]\!]$*, is a mapping* $[\![e]\!] : PSubst_\bot \longrightarrow \mathcal{P}(Pat_\bot)$ *defined by* $[\![e]\!](\theta) = [\![e\theta]\!]$.

(ii) Hypersemantics of expressions are ordered as follows:

$$[\![e_1]\!] \Subset [\![e_2]\!] \text{ iff } [\![e_1\theta]\!] \subseteq [\![e_2\theta]\!],\ \forall \theta \in PSubst_\bot$$

The main reason for introducing hypersemantics is that it enjoys the following nice monotonicity-under-contexts property, while $[\![\_]\!]$ does not:

Lemma 1 (Monotonicity of hypersemantics)
$[\![e]\!] \Subset [\![e']\!]$ implies $[\![\mathcal{C}[e]]\!] \Subset [\![\mathcal{C}[e']]\!]$, for any $e, e' \in LExp_\bot$, $\mathcal{C} \in Cntxt$.

Monotonicity under contexts is the key for our next result, stating that hypersemantics does not grow under *HOlet*-rewriting steps:

Lemma 2 (One-Step Hyper-Soundness of *HOlet*-rewriting)
$e \rightarrow^l e'$ implies $[\![e']\!] \Subset [\![e]\!]$, for any $e, e' \in LExp$.

Notice that $\Subset$ cannot be replaced here by $=$, due to non-determinism.

Lemma 2, together with the easy observation that $[\![e_1]\!] \Subset [\![e_2]\!]$ implies $[\![e_1]\!] \subseteq [\![e_2]\!]$ (just take $\theta = \epsilon$) and an obvious induction over derivation lengths, leads to our main correctness result for $\rightarrow^l$:

Theorem 3 (Soundness of *HOlet*-rewriting). *Let $\mathcal{P}$ be a program, $e, e' \in LExp$. Then: (i) $e \rightarrow^{l^*} e'$ implies $[\![e']\!] \subseteq [\![e]\!]$, and therefore $e \twoheadrightarrow |e'|$*
(ii) $e \rightarrow^{l^*} t$ implies $e \twoheadrightarrow t$, for any $t \in Pat$.

The proof of this result can be easily extended to the larger relation $\rightarrow^L$ (the one which uses (LetIn') instead of (LetIn)).

Regarding completeness of *let*-rewriting, a key in the FO case was the *peeling lemma* ([19], Lemma 7), a technical result giving a kind of standard form in which the implicit or explicit sharing information contained in $e \in Exp$ can be expressed. It is not obvious how to proceed in the HO case, since straightforward generalizations of the FO peeling lemma turn out to be false. However, we have found that the following weak HO version is enough for our purposes:

Lemma 3 (Weak peeling lemma). *Let $h\ e_1 \ldots e_m \in Exp$ with $h \in \Sigma^n$ (n and m can be different). Then $h\ e_1 \ldots e_m \rightarrow^{l^*} let\ \overline{X = a}\ in\ h\ t_1 \ldots t_m$, for some $t_1, \ldots, t_m \in Pat, \overline{a} \subseteq Exp$ such that $|\overline{a}| = \overline{\bot}, t_i \equiv e_i$ for every $e_i \in Pat$. Besides, in this derivation the rule (Fapp) is not applied.*

With this result and some monotonicity properties of *HOCRWL*-derivability, we can prove a very technical but strong completeness result for $\rightarrow^l$ wrt $\twoheadrightarrow$:

Lemma 4 (Completeness lemma for *HOlet*-rewriting). *For any program $\mathcal{P}$, $e \in Exp$ and $t \in Pat_\bot$ with $t \neq \bot$, the following holds: $\mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t$ implies $e \rightarrow^{l^*} let\ \overline{X = a}\ in\ t'$, for some $t' \in Pat$ and $\overline{a} \subseteq Exp$ 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, $t \sqsubseteq t'[\overline{X/\ \bot}]$.*

The condition $t \neq \bot$ is needed, as can be seen just taking $\mathcal{P} = \{f \rightarrow f\}$, $e \equiv f$ and $t \equiv \bot$.

From Lemma 4 we can obtain our main completeness result for $\rightarrow^l$:

Theorem 4 (Completeness of *HOlet*-rewriting). *Let $\mathcal{P}$ be a program, $e \in Exp$, and $t \in Pat_\bot$. Then:*

(i) $\mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t$ implies $e \rightarrow^{l^*} e'$, for some $e' \in LExp$ such that $t \sqsubseteq |e'|$.
(ii) If in addition $t \in Pat$, then $e \rightarrow^{l^*} t$.

Joining together the last parts of Theorems 3 and 4, we obtain a strong equivalence result for $\rightarrow^l$ and $\twoheadrightarrow$:

Theorem 5 (Equivalence of *HOlet*-rewriting and *HOCRWL*)
$\mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t$ iff $e \rightarrow^{l^*} t$, for any $\mathcal{P}$, $e \in Exp$, and $t \in Pat$.

This justifies our claim that $\rightarrow^l$ is truly the reduction face of *HOCRWL*-semantics.

4 Higher Order *let*-narrowing

For some FLP computations rewriting is not enough, and must be lifted to some kind of *narrowing*; this happens when the expression being reduced contains variables for which different bindings might produce different evaluation results. Narrowing is an old subject in the fields of theorem proving and declarative programming. Since classical rewriting is not correct for call-time choice, classical narrowing cannot be either (because rewriting is a particular case of narrowing). In [18] we proposed a notion of narrowing adequate to FO *let*-rewriting, and now we extend it to HO. As happens in [7,4], *HOlet*-narrowing may bind variables to HO-patterns.

Figure 3 contains the rules for the one-step *HOlet*-narrowing relation $e \leadsto^l_\theta e'$, expressing that e is narrowed to e' producing the substitution $\theta \in PSubst$. In *(X)* we collect those cases of *HOlet*-rewriting corresponding also to narrowing steps with empty substitution. *(Narr)* is the proper rule of narrowing for function application; it may produce HO bindings if the used program rule has HO patterns. Notice that, for the sake of generality, we do not require that θ is a mgu. *(VAct)* and *(VBind)* are rules producing HO bindings for flexible expressions (or subexpressions, in the case of *(VBind)*). We have preferred this pair of rules instead of the rule

$$\textbf{(VNarr)} \quad X\ e \leadsto^L_{[X/t]} t\ (e[X/t]), \text{ for any } t \in Pat$$

which is simpler, but also 'wilder' because it creates a larger search space. Finally, (Contxt) is a contextual rule where, as in [18], it is crucial to protect bound variables from narrowing (condition (i)) and to avoid variable capture (condition (ii), automatically fulfilled if mgu's are used in *(Narr)* and *(VAct)*, and fresh *shallow* patterns –i.e., of the form $h\ X_1 \ldots X_n$– in *(VBind)*).

Taking Example 1, a narrowing derivation for *fdouble F 0* would start with some *(X)* 'rewriting' steps:

$$fdouble\ F\ 0 \leadsto^l_\epsilon fadd\ F\ F\ 0 \leadsto^l_\epsilon F\ 0 + F\ 0 \leadsto^l_\epsilon let\ X{=}F\ 0\ in\ X + F\ 0$$

At this point, notice first that we cannot narrow on X, because it is a bound variable. Instead, we can apply *(VAct+Contx)*:

$$let\ X{=}F\ 0\ in\ X + F\ 0 \leadsto^l_{\{F/g\}} let\ X{=}0\ in\ X + g\ 0 \leadsto^{l^*}_\epsilon 0$$

Other similar derivations using *(VAct+Contx)* would bind F to h (with final result *s (s 0)*), or to f' (with possible results *0, s 0, s (s 0)*). Notice that the binding X/f is not legal, since f is not a pattern.

(X) $e \rightsquigarrow^l_\epsilon e'$ if $e \rightarrow^l e'$ using $\boldsymbol{X} \in \{Elim, Bind, Flat, LetIn, LetAp\}$ in Figure 2.

(Narr) $f\ \overline{t} \rightsquigarrow^l_\theta r\theta$, for any fresh variant $(f\ \overline{p} \rightarrow r) \in \mathcal{P}$ and $\theta \in PSubst$ such that $f\ \overline{t}\theta \equiv f\ \overline{p}\theta$.

(VAct) $X\ t_1 \ldots t_k \rightsquigarrow^l_\theta r\theta$, if $k > 0$, for any fresh variant $(f\ \overline{p} \rightarrow r) \in \mathcal{P}$ and $\theta \in PSubst$ such that $(X\ t_1 \ldots t_k)\theta \equiv f\ \overline{p}\theta$.

(VBind) $let\ X = e_1\ in\ e_2 \rightsquigarrow^l_\theta e_2\theta[X/e_1\theta]$, if $e_1 \notin Pat$, for any $\theta \in PSubst$ that makes $e_1\theta \in Pat$, provided that $X \notin (dom(\theta) \cup vRan(\theta))$.

(Contx) $\mathcal{C}[e] \rightsquigarrow^l_\theta \mathcal{C}\theta[e']$ for $\mathcal{C} \neq [\,]$, if $e \rightsquigarrow^l_\theta e'$ by any of the previous rules, and the following conditions hold:

i) $dom(\theta) \cap BV(\mathcal{C}) = \emptyset$

ii) • If the step is *(Narr)* or *(VAct)* using $(f\ \overline{p} \rightarrow r) \in \mathcal{P}$, then $vRan(\theta|_{\setminus var(\overline{p})}) \cap BV(\mathcal{C}) = \emptyset$

• If the step is *(VBind)* then $vRan(\theta) \cap BV(\mathcal{C}) = \emptyset$

Fig. 3. Higher order *let*-narrowing calculus $\rightsquigarrow^l$

Alternatively we could have applied *(VBind)*, obtaining:

$$let\ X{=}F\ 0\ in\ X + F\ 0 \rightsquigarrow^l_{\{F/s\}} s\ 0 + s\ 0 \rightsquigarrow^{l^*}_\epsilon s\ (s\ 0)$$

We remark that, in our untyped framework, other 'ill-typed' bindings could be tried, like $F/fadd\ 0$ or $F/fdouble$. This is a symptom of known problems [4,8] of the interaction with types of the intensional view of HO, that are partially alleviated in [4] by a typed version of a FO translation (see Sect. 6), but in general require (see [8]) bringing types to computations, a problem yet not well solved in practice. All these type-related issues are out of the scope of the paper.

A basic fact about completeness of *let*-narrowing in the FO case was that $e \rightsquigarrow^{l^*}_\theta e'$ implied $e\theta \rightarrow^{l^*} e'$, $\forall \theta \in CSubst$, which is closely related to the fact that FO *let*-rewriting is closed under c-substitutions. None of both facts hold with HO $\rightsquigarrow^l$, $\rightarrow^l$ and $\theta \in PSubst$: consider for instance $e \equiv s\ (Y\ 0) \rightarrow^l let\ X = Y\ 0\ in\ s\ X \equiv e'$ and $\theta = [Y/s]$, for which $e\theta \equiv s\ (s\ 0) \not\rightarrow^l let\ X = s\ 0\ in\ s\ X \equiv e'\theta$. Similarly, we have $e \equiv s\ (Y\ 0) \rightsquigarrow^L_\epsilon let\ X = Y\ 0\ in\ s\ X \rightsquigarrow^L_{[Y/s]} let\ X = s\ 0\ in\ s\ X \equiv e'$, but $e\theta \equiv s\ (s\ 0) \not\rightarrow^l e'$.

At this point the relation $\rightarrow^L$ of Sect. 3 becomes useful, because we have:

Lemma 5 (Closedness of $\rightarrow^L$ under $PSubst$). *For every $e, e' \in LExp, \theta \in PSubst$, $e \rightarrow^{L^*} e'$ implies $e\theta \rightarrow^{L^*} e'\theta$.*

Now we can prove soundness of HO *let*-narrowing wrt. $\rightarrow^L$:

Theorem 6 (Soundness or $\rightsquigarrow^l$ wrt $\rightarrow^L$). *For any $e, e' \in LExp$, $e \rightsquigarrow^{l^*}_\theta e'$ implies $e\theta \rightarrow^{L^*} e'$.*

And now, taking into account Th. 3 (which holds also for $\rightarrow^L$), we get:

Theorem 7 (Soundness of *let*-narrowing). *For any $e, e' \in LExp$, $t \in Pat$:*

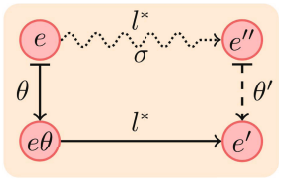
a) If $e \rightsquigarrow^{l^*}_\theta e'$ then $[\![e']\!] \subseteq [\![e\theta]\!]$ *b) If $e \rightsquigarrow^{l^*}_\theta t$ then $e\theta \rightarrow^{l^*} t$*

Regarding completeness, the following lemma shows how we can lift any $\rightarrow^l$ derivation to a $\rightsquigarrow^l$ derivation. This is surely the most involved result in the paper.

Lemma 6 (Lifting lemma for *HOlet*-rewriting). *Let $e, e' \in LExp$ such that $e\theta \rightarrow^{l^*} e'$ for some $\theta \in PSubst$, and let $\mathcal{W}, \mathcal{B} \subseteq \mathcal{V}$ with $dom(\theta) \cup FV(e) \subseteq \mathcal{W}$, $BV(e) \subseteq \mathcal{B}$ and $(dom(\theta) \cup vRan(\theta)) \cap \mathcal{B} = \emptyset$, and for each instance of a program rule $R\gamma \in [\mathcal{P}]$ used in an (Fapp) step of $e\theta \rightarrow^{l^*} e'$ then $vRan(\gamma|_{vExtra(R)}) \cap \mathcal{B} = \emptyset$. Then there exist a derivation $e \rightsquigarrow^{l^*}_{\sigma} e''$ and $\theta' \in PSubst$ such that:*

$$\textit{(i)}\ e''\theta' = e' \qquad \textit{(ii)}\ \sigma\theta' = \theta[\mathcal{W}] \qquad \textit{(iii)}\ (dom(\theta') \cup vRan(\theta')) \cap \mathcal{B} = \emptyset$$

Besides, the HOlet*-narrowing derivation can be chosen to use mgu's at each* **(Narr)** *or* **(VAct)** *step, and fresh shallow patterns in the range for each* **(VBind)** *step. Graphically:*



With the aid of this lemma we can reach our completeness result for $\rightsquigarrow^l$:

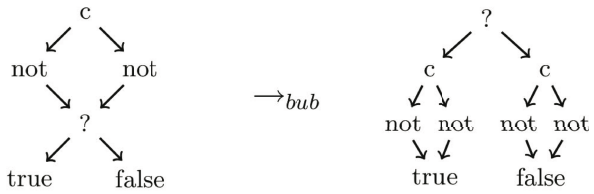
Theorem 8 (Completeness of *HOlet*-narrowing wrt. *HOlet*-rewriting). *Let $e, e' \in LExp$ and $\theta \in PSubst$. If $e\theta \rightarrow^{l^*} e'$, then there exist a* HOlet*-narrowing derivation $e \rightsquigarrow^{l^*}_{\sigma} e''$ and $\theta' \in PSubst$ such that $e''\theta' \equiv e'$ and $\sigma\theta' = \theta[FV(e)]$.*

5 A Case of Study: Correctness of Bubbling

Having equivalent notions of semantics and reduction allows to reason interchangeably at the rewriting and the semantic levels. We demonstrate the power of such technique by a case study where *let*-rewriting provides a good level of abstraction to formulate a new operational rule (*bubbling*), while the semantic point of view is appropriate for proving its correctness.

Bubbling, proposed in [3], is an operational rule devised to improve the efficiency of functional logic computations. Its correctness was formally studied in [2] in the framework of a variant [6] of term graph rewriting.

The idea of bubbling is to concentrate all non-determinism of a system into a *choice* operation ? defined by the rules $X\ ?\ Y \rightarrow X$ *and* $X\ ?\ Y \rightarrow Y$, and to lift applications of ? out of a surrounding context, as illustrated by the following graph transformation taken from [2]:



As it is shown in [3], bubbling can be implemented in such a way that many functional logic programs become more efficient, but we will not deal with these issues here.

Due to the technical particularities of term graph rewriting, not only the proof of correctness, but even the definition of bubbling in [3,2] are involved and need subtle care concerning the appropriate contexts over which choices can be bubbled. In contrast, bubbling can be expressed within our framework (moreover, generalized to HO) in a remarkably easy and abstract way as a new rewriting rule: **(Bub)** $\mathcal{C}[e_1?e_2] \rightarrow^{bub} \mathcal{C}[e_1]?\mathcal{C}[e_2]$, for $e_1, e_2 \in LExp$

With this rule, the bubbling step corresponding to the graph transformation of the example above is: *let X = true ? false in c (not X) (not X)* $\rightarrow^{bub}$ *let X = true in c (not X) (not X) ? let X = false in c (not X) (not X)*

Notice that the effect of this bubbling step is not a shortening of any existing *HOlet*-rewriting derivation; bubbling is indeed a genuine new rule, the correctness of which must be therefore subject of proof. Call-time choice is essential, since bubbling is not correct with respect to run-time choice: in Example 1, *fdouble (g?h) 0* can be reduced with run-time choice to $0, 1$ or 2, while *fdouble g 0 ? fdouble h 0* leads only to 0 and 2.

The fact that bubbling preserves $HOCRWL_{let}$-semantics has a simple formulation:

Theorem 9 (Correctness of bubbling). *If* $e \rightarrow^{bub} e'$*, then* $[\![\mathcal{C}[e]]\!] = [\![\mathcal{C}[e']]\!]$*. In other terms,* $[\![\mathcal{C}[e_1?e_2]]\!] = [\![\mathcal{C}[e_1]?\mathcal{C}[e_2]]\!]$ $(= [\![\mathcal{C}[e_1]]\!] \cup [\![\mathcal{C}[e_2]]\!])$*, for any* $e_1, e_2 \in LExp$ *and context* $\mathcal{C}$*.*

From this and the equivalence results of Sect. 3 we obtain as immediate corollary the correctness of bubbling in terms of rewriting:

Corollary 1. $e \rightarrow_l^* t \Leftrightarrow e\ (\rightarrow_l \cup \rightarrow_{bub})^*\ t$

It is interesting to observe that most of the proof of Th. 9 consists of direct calculations with denotation of expressions, in the form of chains of equalities of denotations, justified by general properties of the semantics like Th. 1. We find this methodology quite appealing and for this reason we include (a part of) the proof.

Proof (For Theorem 9, Correctness of bubbling). The proof uses the following easy (not proved here) lemma about semantics of ?, which justifies also the equation $[\![\mathcal{C}[e_1]?\mathcal{C}[e_2]]\!] = [\![\mathcal{C}[e_1]]\!] \cup [\![\mathcal{C}[e_2]]\!]$ stated in the Theor. 9.

Lemma 7. $[\![e_1?e_2]\!] = [\![e_1]\!] \cup [\![e_2]\!]$*, for any* $e_1, e_2 \in LExp_\bot$*.*

Now, we reason by induction on the number k of *let*'s occurring in $\mathcal{C}[e_1?e_2]$.

- $k = 0$: Since there is no *let* in $e_1?e_2$, we can apply Theor. 1 to obtain:

$$
\begin{array}{ll}
[\![\mathcal{C}[e_1?e_2]]\!] & = \textit{(by Theor. 1)} \\
\bigcup_{t \in [\![e_1?e_2]\!]} [\![\mathcal{C}[t]]\!] & = \textit{(by Lemma 7)} \\
\bigcup_{t \in ([\![\mathcal{C}[e_1]]\!] \cup [\![\mathcal{C}[e_2]]\!])} [\![\mathcal{C}[t]]\!] & = \textit{(set operations)} \\
\bigcup_{t \in [\![\mathcal{C}[e_1]]\!]} [\![\mathcal{C}[t]]\!] \ \cup \ \bigcup_{t \in [\![\mathcal{C}[e_2]]\!]} [\![\mathcal{C}[t]]\!] & = \textit{(by Theor. 1)} \\
[\![\mathcal{C}[e_1]]\!] \ \cup \ [\![\mathcal{C}[e_2]]\!] & = \textit{(by Lemma 7)} \\
[\![\mathcal{C}[e_1] \ ? \ \mathcal{C}[e_2]]\!] &
\end{array}
$$

- $k > 0$: We reason by induction on the structure of $\mathcal{C}$. The most interesting case is that of let bindings:

– $\mathcal{C} \equiv let\ x = e\ in\ \mathcal{C}'$: then

$$
\begin{array}{ll}
[\![\mathcal{C}[e_1 ?e_2]]\!] & = \\
[\![let\ x{=}e\ in\ \mathcal{C}'[e_1 ?e_2]]\!] & = \textit{(by Theor. 2,}\sigma \equiv \{x/t\}\textit{)} \\
\bigcup_{t\in[\![e]\!]}[\![\mathcal{C}'[e_1 ?e_2]\sigma]\!] & = \\
\bigcup_{t\in[\![e]\!]}[\![\mathcal{C}'\sigma[e_1\sigma ?e_2\sigma]]\!] & = \textit{(by IH on } k\textit{, that decreases)} \\
\bigcup_{t\in[\![e]\!]}[\![\mathcal{C}'\sigma[e_1\sigma]?\mathcal{C}'\sigma[e_2\sigma]]\!] & = \textit{(by Lemma 7)} \\
\bigcup_{t\in[\![e]\!]}([\![\mathcal{C}'\sigma[e_1\sigma]]\!] \cup [\![\mathcal{C}'\sigma[e_2\sigma]]\!]) & = \textit{(set operations)} \\
\bigcup_{t\in[\![e]\!]}[\![\mathcal{C}'\sigma[e_1\sigma]]\!] \cup \bigcup_{t\in[\![e]\!]}[\![\mathcal{C}'\sigma[e_2\sigma]]\!] & = \textit{(by Theor. 2)} \\
[\![let\ x{=}e\ in\ \mathcal{C}'[e_1]]\!] \cup [\![let\ x{=}e\ in\ \mathcal{C}'[e_2]]\!] & = \\
[\![\mathcal{C}[e_1]]\!] \cup [\![\mathcal{C}[e_2]]\!] & = \textit{(by Lemma 7)} \\
[\![\mathcal{C}[e_1]\ ?\ \mathcal{C}[e_2]]\!] &
\end{array}
$$

6 Translation to First Order

Since [29], a common technique to implement HO features in FO settings consists in a *HO-to-FO* translation introducing data constructors to represent partial applications and a special function @ (read *apply*) for reducing application of such constructors. This has been used within the context of *FLP* in [9,4]. Here we adapt such a transformation to our context and provide a correctness proof with respect to the semantics of the source and object programs, given by *HOCRWL* and *CRWL* [11,19] respectively.

Definition 2 (First order translation). *Given a HOCRWL-program* $\mathcal{P} = \{f\ \overline{p_1} \to e_1, \ldots, f\ \overline{p_m} \to e_m\}$ *built up over the signature* $\Sigma = FS \cup CS$*, its* first order translation P_{fo} *will be defined over the* **extended signature** $\Sigma_{fo} = FS_{fo} \cup CS_{fo}$ *where:*

$$FS_{fo} = FS \cup \{@\};\ \ CS_{fo} = \bigcup_{c\in CS^n, n\in\mathbb{N}}\{c_0, \ldots, c_n\} \cup \bigcup_{f\in FS^n, n\in\mathbb{N}}\{f_0, \ldots, f_{n-1}\}$$

being @ *a new function symbol of arity 2 and* $c_0, \ldots, c_n, f_0, \ldots, f_{n-1}$ *new symbols (with arities indicated by the sub-index). The set* $\mathcal{P}_@$ *of* $@-rules$ *is defined as:*

$$
\begin{array}{ll}
@(c_k(X_1, \ldots, X_k), Y) & = c_{k+1}(X_1, \ldots, X_k, Y), \textit{for each } c \in DC^n, k < n \\
@(f_k(X_1, \ldots, X_k), Y) & = f_{k+1}(X_1, \ldots, X_k, Y), \textit{for each } f \in FS^n, k+1 < n \\
@(f_{n-1}(X_1, \ldots, X_{n-1}), Y) & = f(X_1, \ldots, X_{n-1}, Y), \textit{for each } f \in FS^n
\end{array}
$$

The transforming function $fo : Exp_{\Sigma,\bot} \to Exp_{\Sigma_{fo},\bot}$ *is defined as:*

$$
\begin{array}{lll}
fo(\bot) = \bot & fo(X) = X & fo(h) = h_0, \textit{if } h \in CS \textit{ or } h \in FS^n, n > 0 \\
fo(f) = f, \textit{if } f \in FS^0 & & fo(e_1\ e_2) = @(fo(e_1), fo(e_2))
\end{array}
$$

The **transformed program** *is defined as* $P_{fo} = \{f(\overline{fo(p_1)\downarrow_@}) \to fo(e_1)\downarrow_@, \ldots, f(\overline{fo(p_m)\downarrow_@}) \to fo(e_m)\downarrow_@\} \cup P_@$*, where* $e\downarrow_@$ *stands for a* **normal form** *for* e *with respect to* $@-rules$ *defined above.*

The program rules obtained by the transformation are well defined: it is easy to prove that if p is a pattern then $fo(p)\downarrow_@$ is a FO constructor term.

For the program of Example 1 we have $FS_{fo} = \{+, f, g, h, f', fadd, fdouble, @\}$ and $CS_{fo} = \{0, s_0, s, +_0, +_1, g_0, h_0, {f'}_0, {fadd}_0, {fadd}_1, {fadd}_2, {fdouble}_0\}$. The translated rules are:

$$g(X) \to 0 \qquad f \to g_0 \qquad f \to h_0 \qquad f'(X) \to @(f, X) \qquad h(X) \to s(0)$$
$$fadd(F, G, X) \to @(F, X) + @(G, X) \qquad fdouble(F) \to {fadd}_2(F, F)$$

And the rules for @ are:

$$@(+_0, X) \to +_1(X) \qquad @(s_0, X) \to s(X) \qquad @(h_0, X) \to h(X)$$
$$@(+_1(X), Y) \to X + Y \qquad @(g_0, X) \to g(X) \qquad @(f'_0, X) \to f'(X)$$
$$@({fadd}_0, F) \to {fadd}_1(F) \qquad @({fadd}_2(F, G), X) \to fadd(F, G, X)$$
$$@({fadd}_1(F), G) \to {fadd}_2(F, G) \qquad @({fdouble}_0, F) \to fdouble(F)$$

The translation of the expressions to reduce in that example are:
$fo(fdouble\ f\ 0)\downarrow_@ = @(fdouble(f), 0) \qquad fo(fdouble\ f'\ 0)\downarrow_@ = @(fdouble(f'_0), 0)$

In general we cannot expect to prove a statement of the form $fo(e) \twoheadrightarrow fo(t)$ because $fo(t)$ can contain calls to the function @, i.e. $fo(t)$ might not be a FO constructor term. But the same statement makes sense in the form $fo(e) \twoheadrightarrow fo(t)\downarrow_@$ because $fo(t)\downarrow_@$ is a FO constructor term.

Proposition 2. *$[\![fo(e)\downarrow_@]\!]^{\mathcal{P}}_{CRWL} = [\![fo(e)]\!]^{\mathcal{P}}_{CRWL}$. Moreover $[\![fo(e)]\!] = [\![e']\!]$ where e' is any expression obtained from e by reducing some calls of @.*

According to this, when proving a statement $fo(e) \twoheadrightarrow t$ we can use any equivalent expression e' (in the sense of previous lemma) in the left hand side and prove $e' \twoheadrightarrow t$.

The correctness of the transformation can be stated then as follows:

Theorem 10 (Adequacy of HO-to-FO translation). *Let $\mathcal{P}$ be a program, $e \in Exp_\perp$, $t \in Pat_\perp$. Then: $\mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t \Leftrightarrow \mathcal{P}_{fo} \vdash_{CRWL} fo(e) \twoheadrightarrow fo(t)\downarrow_@$ Or, in terms of* HOlet*-rewriting: $e \to^{l^*} t \Leftrightarrow fo(e) \to^{l^*} fo(t)\downarrow_@$.*

7 Conclusions

Our paper addresses the broad question: *what means 'reduction' for functional logic programming?*, which had no previous satisfactory answer for the combination *HO + non-deterministic functions + call-time choice* supported by current systems in the mainstream of the field (Curry [16], Toy [21]). This leads to subtle behaviors well characterized from the point of view of a declarative semantics [7], but with no corresponding basic notion of one-step reduction. We have made a number of identifiable **contributions** in this sense:

- We propose a notion of rewriting with local bindings (*HOlet-rewriting*) suitable for a large class of HO systems (possibly non-confluent and non-terminating, allowing extra variables in right-hand sides and HO-patterns in left-hand sides).

- We have proved equivalence of *HOlet*-rewriting wrt to *HOCRWL* [7] declarative semantics. Along the way we have extended *HOCRWL* to cope with *let*s, and established new compositional properties of *HOCRWL* semantics.
- We have lifted *HOlet*-rewriting to a notion of *HOlet-narrowing* which is able to bind variables to patterns, even HO ones representing intensional descriptions of functions. We prove soundness and completeness of *HOlet*-narrowing wrt. *HOlet*-rewriting.
- We have recast within our framework the definition and proof of correctness of *bubbling*, an operational rule investigated in [3,2] using term graph rewriting techniques. Apart from extending it to HO, this case study illustrates quite well the power of using indistinctly rewriting and/or semantic-based reasoning.
- To close the panorama, we have formally proved that *translation from HO to FO*, a technique actually used in the implementations of FLP systems, still works properly when *let*-bindings with call-time choice are considered, while previous works [9,4] consider only deterministic functions.

The first three points have been conceived as an extension to HO of our previous work on the FO case [19,18]. However, adapting it has not been routine; on the contrary, some results have been indeed a technical challenge.

Our wish with this work, jointly with [19,18], is to have provided foundational pieces useful to understand how a FLP computation proceeds, serving also as suitable technical basis to address in the call-time choice context other operational issues (rewriting and narrowing strategies, residuation, program optimization, types in computations,...), all of which are lines of future work.

References

1. Albert, E., Hanus, M., Huch, F., Oliver, J., Vidal, G.: Operational semantics for declarative multi-paradigm languages. J. of Symb. Comp. 40(1), 795–829 (2005)
2. Antoy, S., Brown, D., Chiang, S.: On the correctness of bubbling. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 35–49. Springer, Heidelberg (2006)
3. Antoy, S., Brown, D., Chiang, S.: Lazy context cloning for non-deterministic graph rewriting. In: Proc. Termgraph 2006. ENTCS, vol. 176(1), pp. 61–70 (2007)
4. Antoy, S., Tolmach, A.P.: Typed higher-order narrowing without higher-order strategies. In: Middeldorp, A. (ed.) FLOPS 1999. LNCS, vol. 1722, pp. 335–353. Springer, Heidelberg (1999)
5. Ariola, Z.M., Felleisen, M., Maraist, J., Odersky, M., Wadler, P.: The call-by-need lambda calculus. In: Proc. POPL 1995, pp. 233–246 (1995)
6. Echahed, R., Janodet, J.-C.: Admissible graph rewriting and narrowing. In: Proc. JICSLP 1998, pp. 325–340. MIT Press, Cambridge (1998)
7. González-Moreno, J., Hortalá-González, M., Rodríguez-Artalejo, M.: A higher order rewriting logic for functional logic programming. In: Proc. ICLP 1997, pp. 153–167. MIT Press, Cambridge (1997)
8. González-Moreno, J., Hortalá-González, T., Rodríguez-Artalejo, M.: Polymorphic types in functional logic programming. J. of Functional and Logic Programming 2001/S01, 1–71 (2001)

9. González-Moreno, J.C.: A correctness proof for warren's ho into fo translation. In: Proc. GULP 1993, pp. 569–584 (1993)
10. González-Moreno, J.C., Hortalá-González, M.T., Rodríguez-Artalejo, M.: On the completeness of narrowing as the operational semantics of functional logic programming. In: Martini, S., Börger, E., Kleine Büning, H., Jäger, G., Richter, M.M. (eds.) CSL 1992. LNCS, vol. 702, pp. 216–230. Springer, Heidelberg (1993)
11. González-Moreno, J.C., Hortalá-González, T., López-Fraguas, F., Rodríguez-Artalejo, M.: An approach to declarative programming based on a rewriting logic. J. of Logic Programming 40(1), 47–87 (1999)
12. Hanus, M.: The integration of functions into logic programming: From theory to practice. J. of Logic Programming 19&20, 583–628 (1994)
13. Hanus, M.: Curry mailing list (March, 2007), `http://www.informatik.uni-kiel.de/~curry/listarchive/0497.html`
14. Hanus, M.: Multi-paradigm declarative languages. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 45–75. Springer, Heidelberg (2007)
15. Hanus, M., Prehofer, C.: Higher-order narrowing with definitional trees. J. of Functional Programming 9(1), 33–75 (1999)
16. Hanus, M. (ed.): Curry: An integrated functional logic language (version 0.8.2) (March, 2006), `http://www.informatik.uni-kiel.de/~curry/report.html`
17. Hussmann, H.: Non-Determinism in Algebraic Specifications and Algebraic Programs. Birkhäuser, Basel (1993)
18. López-Fraguas, F., Rodríguez-Hortalá, J., Sánchez-Hernández, J.: Narrowing for non-determinism with call-time choice semantics. In: Proc. WLP 2007 (2007)
19. López-Fraguas, F., Rodríguez-Hortalá, J., Sánchez-Hernández, J.: A simple rewrite notion for call-time choice semantics. In: Proc. PPDP 2007, pp. 197–208. ACM Press, New York (2007)
20. López-Fraguas, F., Rodríguez-Hortalá, J., Sánchez-Hernández, J.: Rewriting and call-time choice: the HO case (extended version). Tech. Rep. SIC-3-08 (2008), `http://gpd.sip.ucm.es/fraguas/papers/flops08long.pdf`
21. López-Fraguas, F., Sánchez-Hernández, J.: $\mathcal{TOY}$: A multiparadigm declarative system. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999)
22. Maraist, J., Odersky, M., Wadler, P.: The call-by-need lambda calculus. J. Funct. Program. 8(3), 275–317 (1998)
23. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. J. Log. Comput. 1(4), 497–536 (1991)
24. Nakahara, K., Middeldorp, A., Ida, T.: A complete narrowing calculus for higher-order functional logic programming. In: Leopold, H., Coulson, G., Danthine, A., Hutchison, D. (eds.) COST-237 1994. LNCS, vol. 882, pp. 97–114. Springer, Heidelberg (1994)
25. Peyton Jones, S.L. (ed.): Haskell 98 Language and Libraries. The Revised Report. Cambridge University Press, Cambridge (2003)
26. Plump, D.: Essentials of term graph rewriting. ENTCS 51 (2001)
27. van Raamsdonk, F.: Higher-order rewriting. In: Term Rewriting Systems, Cambridge University Press, Cambridge (2003)
28. Rodríguez-Artalejo, M.: Functional and constraint logic programming. In: Comon, H., Marché, C., Treinen, R. (eds.) CCL 1999. LNCS, vol. 2002, pp. 202–270. Springer, Heidelberg (2001)
29. Warren, D.H.: Higher-order extensions to prolog: Are they needed? Machine Intelligence 10, 441–454 (1982)