A Flexible Framework for Programming with Non-deterministic Functions *

Francisco J. López-Fraguas
Universidad Complutense de Madrid
fraguas@sip.ucm.es

Juan Rodríguez-Hortalá
Universidad Complutense de Madrid
juanrh@fdi.ucm.es

Jaime Sánchez-Hernández
Universidad Complutense de Madrid
jaime@sip.ucm.es

Abstract

The possibility of non-deterministic reductions is a distinctive feature of some declarative languages. Two semantics commonly adopted for non-determinism are *call-time choice*– a notion that at the operational level is related to the *sharing* mechanism of lazy evaluation in functional languages– and *run-time choice*, which corresponds closely to ordinary term rewriting. But there are practical situations where neither semantics is appropriate, if used in isolation. In this paper we propose to annotate functions in a program with the semantics most adequate to its intended use. Annotated programs are then mapped into a unified core language (but still high level), designed to achieve a careful but neat combination of ordinary rewriting –to cope with run-time choice– with local bindings via a *let*-construct devised to express call-time choice. The result is a flexible framework into which existing languages using pure run-time or call-time choice can be embedded, either directly –in the case of run-time choice– or by means of a simple program transformation introducing *let*s in function definitions –for the case of call-time choice–. We prove the adequacy of the embedding, as well as other relevant properties of the framework.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Nondeterministic languages

General Terms Theory, languages.

Keywords Term rewriting systems, non-deterministic functions, run-time-choice, call-time choice, local bindings.

1. Introduction

The possibility of performing non-deterministic reductions is a distinctive feature of some declarative languages, as happens with modern functional logic languages (see (15) for a recent survey) or other languages allowing non-confluent rewriting systems as programs (e.g. *Maude* (9)). It is known that the introduction of non-determinism in a functional setting gives rise to a variety of semantic decisions (see e.g. (27)). For term-rewriting based specifications, Hussmann (18) established a major distinction between *call-time choice* and *run-time choice*. Call-time choice is closely related to call-by-value and, in the case of strict semantics, it is easily implemented by innermost rewriting. In the case of non-strict semantics, things are more complicated, since the call-by-value view of call-time choice must include partial values. In operational terms, call-time choice cannot be achieved by imposing a particular strategy to ordinary rewriting (see (21) for a simple proof) but needs something similar to the sharing mechanism followed, because of efficiency reasons, in lazy functional languages like Haskell. In contrast, run-time choice does not share and can be directly realized by ordinary rewriting. For deterministic programs, run-time and call-time choice are able to produce the same set of values, but in general the set of values that are reachable by run-time choice is larger than in the case of call-time choice.

Non-deterministic functions with non-strict and call-time choice semantics were introduced in the functional logic setting with the *CRWL* framework (13; 14), in which programs are possibly non-confluent and non-terminating constructor-based term rewriting systems (*CTRS*). Since then, they are common part of daily programming in systems like *Curry* (16) or *Toy* (25). On the other hand, run-time choice is the implicit option taken by systems based directly on term rewriting like *Maude*, but has been rarely (3) considered in the functional-logic setting as a valuable global alternative to call-time choice.

The starting point of this work is the observation that, in practice, there might be parts in a program or individual functions for which run-time choice is the best option, while for others parts of the same program call-time choice is more appropriate. Monolithic languages supporting only one semantics do not deal well with such situations, for which less direct solutions must be adopted. A more convenient solution would be to have both possibilities (run-time/call-time choice) at programmer's disposal. Such a combination is not offered by any existing system and, as we will see in Sect. 5, it is not easy to achieve without adding really new constructs to existing languages. The purpose of this work is precisely proposing a clear, well-founded formal framework for doing that.

Our approach in a nutshell could be described as follows: programs will be constructor-based term rewriting systems, where each function in a program is annotated with the intended semantics (run-time choice or call-time choice) for it. Annotated programs are then transformed into a core language that essentially results of adding a *let*-construct to a run-time choice framework (i.e., to ordinary rewriting). For this core language we define a rewriting relation (called *rt-let*-rewriting) that mixes ordinary rewriting with suitable laws for the propagation of bindings contained in *let*s. Our language subsumes pure run-time choice and pure call-time choice (it is enough to annotate all functions with the corresponding semantics). Since those are well-established frameworks, we must

* This work has been partially supported by the Spanish projects TIN2005-09207-C03-03 (MERIT-FORMS-UCM), S-0505/TIC/0407 (PROMESAS-CAM) and TIN2008-06622-C03-01/TIN (FAST-STAMP).

PEPM'09, January 19–20, 2009, Savannah, Georgia, USA.

ensure that our transformation into the core language is harmless and respects the original semantics.

The rest of the paper is organized as follows. We start in Sect. 2 by giving some examples and further explanations trying to motivate in more concrete terms the interest of combining both semantics. Section 3 presents the language for combined semantics via annotations, the core language into which programs are transformed, and the *rt-let*-rewriting relation for performing reductions with core programs. Discussion of related work has been also deferred to this section. Section 4 proves that our new framework is a conservative extension of both pure run-time and pure call-time choice. In Section 5 we discuss in detail the question of whether our approach could be replaced by simpler ones; we point out some limits in the ability of run-time and call-time choice to simulate each other, and we show that our *rt-let*-rewriting compares advantageously to other alternative paths that might be followed. Finally Section 6 summarizes some conclusions. Fully detailed proofs, including many auxiliary results, can be found in (22).

2. First examples

Example 1. Modeling grammar rules for string generation can be directly done by CTRS like the following (non-confluent and non-terminating) one, in which we assume that texts (terminals) are represented as strings (lists of characters), that can be concatenated with ++ (defined in a standard way):

$$\begin{array}{lll} \mathit{letter} \to \text{"a"} & \ldots\ldots & \mathit{letter} \to \text{"z"} \\ \mathit{word} \to \text{" "} & & \mathit{word} \to \mathit{letter} \mathbin{++} \mathit{word} \end{array}$$

Disregarding concrete syntax, that CTRS is a valid program in functional logic systems like *Curry* (16) or *Toy* (25). The program acts as a non-deterministic generator of the strings in the language defined by the grammar. Each individual reduction leads to a string. Now imagine that we want to include the generation of palindromes in the specification. This could be done by the rewrite rules:

$$\begin{array}{l} \mathit{palindrome} \to \mathit{palAux(word)} \\ \mathit{palAux(X)} \to \mathit{X} \mathbin{++} \mathit{reverse(X)} \\ \mathit{palAux(X)} \to \mathit{X} \mathbin{++} \mathit{letter} \mathbin{++} \mathit{reverse(X)} \end{array}$$

where *reverse* is defined in any standard way.

It is important to remark that the definition of *palindrome/palAux* works fine only if call-time choice is adopted for non-determinism, meaning operationally that in the (partial) reduction

$$\mathit{palindrome} \to \mathit{palAux(word)} \to \mathit{word} \mathbin{++} \mathit{reverse(word)}$$

the two occurrences of *word* created by the rule of *palAux* must be shared. If run-time choice (i.e., ordinary rewriting) were used, the two occurrences of *word* could follow independent ways, and therefore *palindrome* could be reduced, for instance, to *"oops"*, which is not a palindrome.

Two useful operators to structure grammar specifications are the alternative '|' and Kleene's '$*$' for repetitions:

$$\begin{array}{ll} \mathit{X} \mid \mathit{Y} \to \mathit{X} & \mathit{X} \mid \mathit{Y} \to \mathit{Y} \\ \mathit{star(X)} \to \text{" "} & \mathit{star(X)} \to \mathit{X} {+}{+} \mathit{star(X)} \end{array}$$

With them *letter* and *word* could be redefined as follows:

$$\begin{array}{l} \mathit{letter} \to \text{"a"} \mid \text{"b"} \mid \ldots \mid \text{"z"} \\ \mathit{word} \to \mathit{star(letter)} \end{array}$$

The annoying fact is that this does not work! At least not under call-time choice, with which all the occurrences of *letter* created by *star* will be shared and therefore *word* will only generate words like *aaa* or *nnnn*, made with repetitions of the same letter.

The problem would be overcome if the function *star* would follow a *run-time choice* regime, so that the evaluation of each of the two occurrences of *letter* created in the rewrite sequence

$$\mathit{word} \to \mathit{star(letter)} \to \mathit{letter} \mathbin{++} \mathit{star(letter)}$$

is not shared, but could evolve independently.

We conclude that in this example neither call-time nor run-time choice are a good single option as semantics for the whole program. The definition of *palAux* requires call-time choice, while *star* requires run-time choice.

Discussion. Our proposal in this paper is that each function in a program should be annotated with the particular regime (call-time or run-time choice) adequate to it. Any way to declare the regime serves: for instance, with declarations like *rtc function_name*, and similar for *ctc*.

The dichotomy *sharing/not sharing* helps to understand in operational terms the difference between call-time and run-time choice. Another point of view that could guide the programmer while focusing more in the declarative meaning of both semantics is the following: If a function f (say of arity 1) follows call-time choice, the variables in the rules defining f range over the universe of *values*, where values are irreducible expressions made of data constructors. This means that the rules only specify the behaviour of $f(t)$ when t is a value. Of course, f can be applied to more general expressions, even to non-deterministic ones, but in this case the result of the evaluation of $f(e)$ is determined by the values to which e can be reduced. For instance, if the function *pair* is defined as *pair(X)* $\to$ *(X,X)* and declared as *ctc*, then the previous rule must be understood as stating how *pair* is applied to a value (e.g., *pair("a")* is *("a","a")*); the possible values of, e.g., *pair(letter)* come from the values of *letter* (which are *"a"*,...,*"z"*), yielding *("a","a")*,...,*("z","z")*. Therefore, non-deterministic expressions can be seen as *transparent* (or *open*) containers of values, that only serve for picking up values from them during evaluation. This view of call-time choice was made compatible with non-termination and lazy evaluation in the CRWL framework (14).

In contrast, if a function is declared to follow run-time choice, the variables in its rules are meant to range over the universe of all expressions. Expressions can be seen as *opaque* (or *closed*) containers (or generators) of values, that are in themselves useful pieces of information that can be passed as arguments or returned by functions without any previous manipulation (i.e., evaluation). For instance, if the function *pair* would have been declared as *rtc*, then the previous rule must be understood as stating how *pair* is applied to any expression (container) X, yielding a pair (X, X) that has two copies of the container X, which is then passed around without opening it. Containers are opened (evaluated) when required by a pattern matching operation or for completing the evaluation of the initial expression. For instance, the evaluation of *pair(letter)* would give *(letter,letter)* in one step; further evaluation will yield, non-deterministically, each of the possible combinations *("a","a")*, ...*("a","z")*,..., *("z","a")*,..., *("z","z")*.

It is interesting to remark that although the function *pair* in itself has nothing to do with non-determinism, we have seen that declaring it as *ctc* or *rtc* makes a great difference when it is applied to a non-deterministic argument. Typically, for those parts of a program expressing how to compute with individual values call-time choice will be the best option. Run-time choice will be adopted when we want to specify how to compute with sets of values as a whole (to be freely copied and unopened while not needed); to set a default option, we assume that a function follows *ctc* when nothing explicit has been said, but this is somehow arbitrary.

Example 2. To emphasize the proximity to practical programming, this example uses higher order (HO) functions and curried notation, although the formal framework to be presented below is first order (FO). Moving from a HO to a FO setting can be done following standard well-established techniques (28; 23; 5) that are used in current systems.

We want to specify the construction of *flags*, where each possible flag is given by a sequence of colors such that no contiguous colors are equal. This is not difficult, but combining call-time choice and run-time choice provides an interesting solution that respects to a large extent the 'separation of concerns principle', advocated for instance in (17) as a virtue of lazy functional languages, in which one can program independent 'bricks' to be 'glued' by lazy evaluation. In our case, some of the bricks are non-deterministic.

We first specify a non-deterministic generation of colors:

color → blue | red | yellow | pink | green | orange

Some standard manipulation of lists (of values) follows:

% checks absence of contiguous repetitions in a list
test [] → true
test [X] → true
test [X,Y | Ys] → if (X==Y) then false else test [Y| Ys]

% takes N elements from the front of a list
take N [] → []
take N [X|Xs] → if N==0 then [] else [X|take (N-1) Xs]

% checks if the value X passes the Test, and returns it
filterValue Test X → if (Test X) then X

We need also two generic *rtc*-annotated functions to be applied to non-deterministic generators of values. The first, *repeat E*, creates a (potentially) infinite number of copies of its argument, and the second, *some N Test E*, returns a list that fulfils *Test* and is made of *N* values of the generator *E*:

rtc repeat
repeat E → [E| repeat E]

rtc some
some N Test E → filterValue Test (take N (repeat E))

If *repeat* and *some* were not declared as *rtc*, their intended semantics would not be achieved, since all the created copies of *E* would be shared, and only lists made with the same value would be obtained.

Finally, building a flag of *N* colors can be defined by:

flag N → some N test color

3. A language with combined semantics

3.1 Syntax

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) 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 FS is partitioned into two sets FS_{rtc} and FS_{ctc} of functions following run-time choice and call-time choice respectively. In practice, constructor symbols are typically introduced by data type declarations, and we may assume that function symbols of FS_{rtc} (FS_{ctc} resp.) are introduced by declarations of the form *rtc f* (*ctc f* resp.).

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, not further evaluable. 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 $var(e)$.

A *program* $\mathcal{P} \in Program$ is a *constructor-based term rewriting system (CTRS)*, that is, a set of 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, linearity meaning that variables occur only once in $\overline{t}$. Notice that we allow e to contain *extra variables*, i.e., variables not occurring in $\overline{t}$. Although this complicates some parts of the formal treatment below, the generality obtained is well worth, since extra variables are useful for practical programming.

Sect. 2 already contained some program examples. In Ex. 1, the partition of function symbols would be $FS_{rtc} = \{star, |\}$, since they are operations intended to be applied to generators of strings (although in the case of |, the behavior would not change if declared as *ctc*). FS_{ctc} would consist of the remaining functions in the program.

3.2 A core language: run-time choice with local bindings

Our next step is defining the expected behavior of programs. Since programs are CTRS, this is best done by giving a precise notion of reduction step that generalizes adequately the standard notion of rewrite step.

If *rtc*-functions and *ctc*-functions did not interact, their combination would not be a challenge at all: *rtc*-function applications would be reduced using ordinary rewriting, and for *ctc*-functions we could use any of the existing formal descriptions of call-time choice (14; 2; 21). However, this is not enough if computations merge both kinds of functions; this happens easily, as in Example 1 where the evaluation of *palindrome* (a *ctc*-function) involves evaluation of *star* (a *rtc*-function).

At the technical level, we have found to be convenient to base our notion of reduction in a core (still high-level) annotation-free language, which essentially comes from enlarging standard TRS with a *let*-construct to express local bindings. Reduction in the core language will consist in a careful combination of ordinary rewriting –to cope with run-time choice– and *let*-management –to express sharing and call-time choice–.

In the *core language*, the syntax of expressions is extended to include *let*-bindings. *Let-expressions* are then 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 $e_1, \ldots, e_n \in LExp$. Recursive *let*s are not considered. In an expression $let\ X = e_1\ in\ e_2$, e_1 and e_2 are called the *defining* expression and the *body* of the *let*-expression, respectively. The notation $let\ \overline{X = a}\ in\ e$ abbreviates $let\ X_1 = a_1\ in\ \ldots\ in\ let\ X_n = a_n\ in\ e$.

Programs $\mathcal{P} \in Program_{let}$ in the core language are defined as above, with the exception that right-hand sides of program rules can contain *let*s, i.e., a program rule takes the form $f(\overline{t}) \to e$ with $e \in LExp$. Notice that $Program \subseteq Program_{let}$.

The sets $FV(e)$ and $BV(e)$ of *free* and *bound* variables of $e \in LExp$ are defined in the natural way. We assume a variable convention according to which the same variable symbol does not occur free and bound within an expression.

3.2.1 Mapping annotated programs into the core

The *rtc* or *ctc* annotation of a function in an annotated program intends to determine its behavior. This is replaced in the core language by explicit local bindings made up with *let*s. The exact

behavior of *let*s is given by the *rt-let*-rewriting relation defined in the next subsection, but the intuition is clear: in the reduction of $let\ X = e_1\ in\ e_2$, all occurrences of X in e_2 will share the same value, that will come from the evaluation of e_1. This gives the hint for the mapping $\tau : Program \mapsto Program_{let}$ that transforms annotated programs into core programs.

DEFINITION 1 (Sharing transformation τ).
Given a program rule $R \equiv f(\bar{t}) \to e$, its transformed rule is:

- $\tau(R) \equiv R$, *if* $f \in FS_{rtc}$.
- $\tau(R) \equiv f(\bar{t}) \to let\ \overline{Y = X}\ in\ e[\overline{X/Y}]$, *where* $FV(e) = \overline{X}$ *and* $\overline{Y}$ *is a linear tuple of fresh variables, if* $f \in FS_{ctc}$. *The notation* $e[\overline{X/Y}]$ *expresses the replacement in* e *of every free occurrence of the variables in* $\overline{X}$ *by the corresponding in* $\overline{Y}$.

The transformation is naturally extended to programs as

$$\tau(\mathcal{P}) = \{\tau(R) | R \in \mathcal{P}\}$$

This transformation leaves untouched the rules for *rtc*-functions (even if *ctc*-functions are invoked in the right-hand side), and introduces a *let*-binding for each variable in the right-hand side, in the case of program rules for *ctc*-functions. For instance, the transformed rule for the *ctc*-function *palAux* in Ex. 1 will be

palAux(X) → let Y = X in Y ++ reverse(Y).

We will prove in Sect. 4 that, for pure call-time choice programs, the behavior resulting of this transformation together with the definition of *rt-let*-rewriting given below corresponds exactly to the standard well-established semantics of call-time choice (14; 21).

3.2.2 *Rt-let*-rewriting

We define here how reduction of an expression $e \in LExp$ must proceed according to the rules of a core program $\mathcal{P} \in Program_{let}$. The essential notion is that of *one step of reduction*, given by the run-time rewriting relation with local bindings (or *rt-let*-rewriting), written $\to^{rt}$ (or $\to^{rt}_{\mathcal{P}}$ if the program $\mathcal{P}$ is made explicit). We first need some terminology.

Substitutions and contexts. *Substitutions* $\theta \in Subst$ are mappings $\theta : \mathcal{V} \longrightarrow Exp$, extending naturally to $\theta : Exp \longrightarrow Exp$. We write $e\theta$ for the application of θ to e, and $\theta\theta'$ for the composition, defined by $X(\theta\theta') = (X\theta)\theta'$. The domain and range of θ are defined as $dom(\theta) = \{X \in \mathcal{V} \mid X\theta \neq X\}$ and $vran(\theta) = \bigcup_{X \in dom(\theta)} var(X\theta)$. Substitutions extend also to $\theta : LExp \longrightarrow LExp$, assuming that whenever θ is applied to $e \in LExp$, the necessary renamings of bound variables are done in e to ensure that $BV(e) \cap (dom(\theta) \cup vran(\theta)) = \emptyset$. These conditions avoid variable capture when applying a substitution, which can be then defined by the rules:

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

C-substitutions $\theta \in CSubst$ verify that $X\theta \in CTerm$ for all $X \in dom(\theta)$. We consider also *Let-substitutions* $\theta \in LSubst$, that are are mappings $\theta : \mathcal{V} \longrightarrow LExp$.

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:

$$[\,][e] = e$$
$$(let\ X = \mathcal{C}\ in\ e')[e] = let\ X = \mathcal{C}[e]\ in\ e'$$
$$(let\ X = e'\ in\ \mathcal{C})[e] = let\ X = e'\ in\ \mathcal{C}[e]$$
$$h(e_1, \ldots, \mathcal{C}, \ldots, e_n)[e] = h(e_1, \ldots, \mathcal{C}[e], \ldots, e_n)$$

Notice that, in contrast to substitutions, application of contexts do not try to avoid variable capture.
Free variables of contexts are defined as for expressions, so that $FV(\mathcal{C}) = FV(\mathcal{C}[a])$, for any constant a. However, the set $BV(\mathcal{C})$ of *bound variables of a context* is defined quite differently because it consists only of those *let*-bound variables visible from the hole of $\mathcal{C}$. Formally:

$$BV([\,]) = \emptyset$$
$$BV(h(\ldots, \mathcal{C}, \ldots)) = BV(\mathcal{C})$$
$$BV(let\ X = e\ in\ \mathcal{C}) = \{X\} \cup BV(\mathcal{C})$$
$$BV(let\ X = \mathcal{C}\ in\ e) = BV(\mathcal{C})$$

We will also employ the notion of *c-contexts*, which are contexts whose holes appear only within a nested application of constructor symbols, that is, $\mathcal{C} ::= [\,] \mid c(e_1, \ldots, \mathcal{C}, \ldots, e_n)$, with $c \in CS^n$, $e_1, \ldots, e_n \in LExp$.

Figure 1 shows the rules of $\to^{rt}$. In it, $\mathcal{P}$ is a program, $X, Y, Z \in \mathcal{V}$, $f \in FS, h \in FS \cup CS, t \in CTerm, e, e_i, a \in LExp$, and $\mathcal{C}, \mathcal{C}' \in Cntx$.

Some explanations about the rules follow. Rule **(Fapp)** allows us to perform ordinary rewriting steps: when an expression matches the left-hand side of a program rule we can replace this expression with the right-hand side of the corresponding rule instance. Condition *i)* is imposed to avoid the capture of free extra variables introduced by σ. But we remark that in absence of extra variables in program rules, condition *i)* trivially holds and therefore (Fapp) (i.e., ordinary rewriting) can be done in any context.

The rest of the $\to^{rt}$-rules forget about the program and deal only with *let*-bindings. An important intuition is that if a step $e \to^{rt'} e'$ is performed using any of these rules that are independent from the program, then the set of $\to^{rt}$-reachable values (i.e. constructor terms) will be the same for e and e'. Therefore all nondeterminism involved in these rules is *don't care*; only (Fapp) is *don't know*. Furthermore, we will see (Prop. 1) that those rules are not a source of non-termination. Let us now comment each of them.

When the defining expression of a *let*-binding has been reduced to a value then the rule **(RBind)** (restricted bind) can be used to propagate this value to the body of the *let*.

The restriction expressed in condition $ii)$ is needed to be coherent with the fact that in $\to^{rt}$ we use $LSubst$ for parameter passing, and so any variable can be potentially instantiated with a $LExp$. Now, notice that if we dropped condition *ii)*, a step like $let\ Y = X\ in\ (Y, Y) \to^{rt} (X, X)$ would be allowed; however, some of its particular cases (replacing the free variable X by concrete expressions) are not valid, as happens for instance with $let\ Y = word\ in\ (Y, Y) \to^{rt} (word, word)$, which is forbidden because it does not respect sharing. The property that any reduction step performed from an expression is also possible with any of its instances (obtained by a substitution of the kind allowed in parameter passing) is a desirable property, for it is very useful to reason about the programs. For example replacing the program rule $(f(X) \to let\ Y = X\ in\ (Y, Y))$ with $(f(X) \to (X, X))$ is unsound, because they provide different levels of sharing: this could be easily detected in our setting because the step $let\ Y = X\ in\ (Y, Y) \to^{rt} (X, X)$ is forbidden.

(Elim) erases a *let*-binding when the bound variable does not appear in the body. The flattening rules **(Flat$_1$)** and **(Flat$_2$)** distribute the bindings, preventing derivations to become wrongly blocked. We remark that our variable convention ensures that application of (Flat$_1$) or (Flat$_2$) does not capture variables. The rule **(LetIn)** is designed to introduce *let*s only for expressions which are already shared, that is, which are present in a defining expression: introducing *let*s in more occasions would reduce the set of reachable values, causing incompleteness. Be-

The auxiliary relation $\rightarrow^{rt'}$ is defined by the rules:

(Fapp) $f(\bar{t})\sigma \rightarrow^{rt'} e\sigma$ if $f(\bar{t}) \rightarrow e$ is a rule of $\mathcal{P}$, $\sigma \in LSubst$

(RBind) $let\ X = t\ in\ e \rightarrow^{rt'} e[X/t]$

(Elim) $let\ X = e_1\ in\ e_2 \rightarrow^{rt'} e_2$ if $X \notin FV(e_2)$

(Flat$_1$) $h(\ldots, let\ X = e_1\ in\ e_2, \ldots) \rightarrow^{rt'} let\ X = e_1\ in\ h(\ldots, e_2, \ldots)$

(Flat$_2$) $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \rightarrow^{rt'} let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$

(LetIn) $let\ X = \mathcal{C}[e]\ in\ e' \rightarrow^{rt'} let\ Y = e\ in\ let\ X = \mathcal{C}[Y]\ in\ e'$
where Y is fresh, if $\mathcal{C} \neq [\,]$ is a c-context and $e \equiv f(\bar{e})$ or $e \in \mathcal{V}$.

The relation $\rightarrow^{rt}$ is defined as: for any $\mathcal{C} \in Cntx$, $\mathcal{C}[e] \rightarrow^{rt} \mathcal{C}[e']$ if $e \rightarrow^{rt'} e'$ and the following conditions hold:

i) If $e \rightarrow^{rt'} e'$ is $f(\bar{t})\sigma \rightarrow^{rt'} r\sigma$ by (Fapp) using $(f(\bar{t}) \rightarrow r) \in \mathcal{P}$ and $\sigma \in LSubst$, then $vran(\sigma|_{\setminus var(\bar{t})}) \cap BV(\mathcal{C}) = \emptyset$

ii) If $e \rightarrow^{rt'} e'$ is $let\ X = t\ in\ a \rightarrow^{rt'} a[X/t]$ by (RBind), then $var(t) \subseteq BV(\mathcal{C})$

iii) If $e \rightarrow^{rt'} e'$ is $let\ X = \mathcal{C}'[Y]\ in\ a \rightarrow^{rt'} let\ Z = Y\ in\ let\ X = \mathcal{C}'[Z]\ in\ a$ by (LetIn), then $Y \notin BV(\mathcal{C})$

Figure 1. Run-time let rewriting relation $\rightarrow^{rt}$

sides that, the context in which they appear must be a c-context because these (LetIn) steps are performed in order to enable a future (RBind) step, to propagate the partial value for the defining expression computed so far; the condition $C \neq [\,]$ avoids successive and useless applications of these rules. Specifically, the case $e \in \mathcal{V}$ in rule (LetIn) is needed to proceed in derivations blocked by the restrictions in (RBind), as illustrated by the program $\mathcal{P} = \{f(c(X)) \rightarrow true\}$ and the expression $let\ Y = c(X)\ in\ f(Y)$, to which (RBind) cannot be applied because X is free and therefore does not fulfil condition *ii)*. Without the case $e \in \mathcal{V}$ in (LetIn), that expression would be a normal form representing incorrectly a failed computation; but using (LetIn) as it is proposed we can do $let\ Y = c(X)\ in\ f(Y) \rightarrow^{rt} let\ Z = X\ in\ let\ Y = c(Z)\ in\ f(Y)$; now the computation can proceed successfully by applying (RBind,Fapp,Elim) yielding $let\ Z = X\ in\ f(c(Z)) \rightarrow^{rt} let\ Z = X\ in\ true \rightarrow^{rt} true$. The condition *iii)* affecting rule (LetIn) is only imposed to forbid useless steps of extraction of a bound variable, which are not needed to enable the application of (RBind).

One of our concerns has been the careful treatment of extra variables in program rules, which is another point where call-time choice and run-time choice greatly differ. In call-time choice, the *CRWL*-semantics instantiates extra variables only with c-terms, but our *rt-let*-rewriting, which in particular attempts to be a strict generalization of ordinary rewriting (see Sect. 4), will instantiate them with any expression. This is a good point to recall that, as argued also in (21), rewriting (either ordinary, run-time or call-time rewriting) by itself is an ineffective operational procedure in presence of rules with extra variables, because a rewriting step using such rules requires a 'magic guessing' of an appropriate substitution for the extra variables. The natural solution to this problem is to perform *narrowing* instead of rewriting in such situations; how to perform narrowing in a way coherent with *rt-let*-rewriting is not obvious, and we postpone it for future work. Nevertheless, to have a rewriting notion is important, since typically the narrowing rules are designed to lift rewriting reductions. Moreover, rewriting is enough if programs do not contain extra variables and the expressions to be reduced are ground.

As an example of derivation, consider the following program $\mathcal{P}$ defining some easy operations for natural numbers (represented with 0 and s in the standard way):

$$\begin{array}{ll} coin \rightarrow 0 & 0 + X \rightarrow X \\ coin \rightarrow s(0) & s(X) + Y \rightarrow s(X + Y) \\ pos(s(X)) \rightarrow true & double(X) \rightarrow let\ Y = X\ in\ Y + Y \end{array}$$

Notice the *let*-binding in the function *double*; due to its presence, $double(coin)$ can be evaluated to 0 or $s(s(0))$, but not to $s(0)$ (that could be obtained with $\rightarrow^{rt}$ if the binding were not present). The following is a possible $\rightarrow^{rt}$-derivation with $\mathcal{P}$ for the expression $pos(double(double(coin)))$. At each step, the redex is underlined and the applied $\rightarrow^{rt}$-rule is indicated on the right:

$$\begin{array}{ll} pos(\underline{double(double(coin))}) & (Fapp) \\ \rightarrow^{rt} \overline{pos(\underline{let\ Y = double(coin)\ in\ Y + Y})} & (Flat_1) \\ \rightarrow^{rt} \overline{let\ Y = \underline{double(coin)}\ in\ pos(Y + Y)} & (Fapp) \\ \rightarrow^{rt} \overline{let\ Y = \underline{(let\ Z = coin\ in\ Z + Z)}\ in\ pos(Y + Y)} & (Flat_2) \\ \rightarrow^{rt} \overline{let\ Z = \underline{coin}\ in\ let\ Y = Z + Z\ in\ pos(Y + Y)} & (Fapp) \\ \rightarrow^{rt} \overline{let\ Z = \underline{s(0)}\ in\ let\ Y = Z + Z\ in\ pos(Y + Y)} & (RBind) \\ \rightarrow^{rt} \overline{let\ Y = \underline{s(0) + s(0)}\ in\ pos(Y + Y)} & (Fapp) \\ \rightarrow^{rt} \overline{let\ Y = \underline{s(0 + s(0))}\ in\ pos(Y + Y)} & (LetIn) \\ \rightarrow^{rt} \overline{let\ V = 0 + s(0)\ in\ \underline{let\ Y = s(V)\ in\ pos(Y + Y)}} & (RBind) \\ \rightarrow^{rt} let\ V = 0 + s(0)\ in\ \overline{pos(\underline{s(V) + s(V)})} & (Fapp) \\ \rightarrow^{rt} let\ V = 0 + s(0)\ in\ \underline{pos(s(V + s(V)))} & (Fapp) \\ \rightarrow^{rt} \underline{let\ V = 0 + s(0)\ in\ true} & (Elim) \\ \rightarrow^{rt} true & \end{array}$$

This is not the only possible derivation, nor the shortest one, but it illustrates some interesting aspects of the run-time rewriting relation. After the first use of (Fapp) we obtain a *let* construction inside a function call, that is extracted by (Flat$_1$). The applications of (Flat$_N$) or (LetIn) enable the application of (RBind) and, ultimately, of (Fapp). The last (Fapp) step shows how lazy evaluation works, without evaluating the inner '+'. The final step erases residual bindings and obtain the expected value.

A first interesting property that we have pursued in the design of the relation $\rightarrow^{rt}$ is that the program rules, to be applied through (Fapp), should be the only potential source of non-termination. The following result shows that this is indeed so.

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

For some results coming below, we need some more terminology: as usual with non-strict languages, in order to express some aspects of the semantics of expressions and programs the signature is enhanced with a new constant (0-ary constructor) symbol $\bot$, to represent the undefined value. Each syntactic domain $\mathcal{D} \in \{CTerm, Exp, LExp, Subst, CSubst\}$ is enlarged to the corresponding $\mathcal{D}_\bot$ of partial c-terms, etc. Notice, however, that $\bot$ does not appear in programs, nor it is introduced by any of the rewriting relations considered in the paper. Expressions in $LExp_\bot$ 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 $e, e' \in LExp_\bot, \mathcal{C} \in Cntxt$.

The next result reflects the fact that all rules except (Fapp) are syntactic transformations that preserve the *shell* of the reduced expression, where the *shell* $|e|$ of an expression $e \in LExp_\bot$ represents the outer constructed part (maybe implicit in *let*-bindings) of the expression, that is, the information that cannot disappear by reduction. Its formal definition is:

$$|X| = X \qquad |c(e_1, \ldots, e_n)| = c(|e_1|, \ldots, |e_n|)$$
$$|f(e_1, \ldots, e_n)| = \bot \qquad |let\ X = e_1\ in\ e_2| = |e_2|[X/|e_1|]$$

PROPOSITION 2. *For any $e, e' \in LExp$, if $e \rightarrow^{rt\ *} e'$ does not use (Fapp), then $|e| = |e'|$.*

This is a first partial soundness result about the relation $\rightarrow^{rt}$.

3.3 Related work

In our work, *let*-bindings have been added to ordinary rewriting, and their laws have been tuned up to achieve a correct combination of run-time/call-time choice. This aim has not been pursued in other works that consider also local bindings, and therefore comparison is difficult. Nevertheless, we can say something about similarities and differences:

• *Local definitions* of existing functional logic languages ((16; 25)): as in the functional case, they can be eliminated by lifting. Since those languages only support call-time choice, nothing really new is achieved with such *let*s, except program readability.

• *Let*s of $lambda$-calculus with sharing ((7; 6)): they formalize sharing in lambda-calculus, but have nothing to do with nondeterminism. Moreover, the underlying formalism is $lambda$-calculus instead of term rewriting.

• *Let*s of *ct-let*-rewriting[1], proposed in (21) as a notion of one-step reduction adequately reflecting *CRWL*'s lazy call-time choice while avoiding the complexity of term graph rewriting (26; 10).

First of all, the framework presented in (21) does not contemplate combination of semantics, but only call-time choice.

Furthermore, the use of *let*s in (21) follows a somehow complementary view to which is done here: in *ct-let*-rewriting, *let*s are introduced by the computation, even if the program does not contain *let*s at all, and must be combined with a restrictive function application rule that avoids the potential duplication of arguments caused by ordinary rewriting, in order to avoid run-time choice behavior. In contrast, in *rt-let*-rewriting function applications (by rule (Fapp)) are liberal (as ordinary rewriting is), and computations do not introduce *let*s, except those explicitly written in program rules to intentionally express call-time choice. As a consequence, the rules for function application and for *let*-management in (21) and in this paper are greatly different.

• Heap-based bindings *à la* Launchbury (19), which have been adapted to call-time choice in (2). Run-time choice is not considered in that work, and we find several reasons to guess that trying to cover run-time choice by extending that paper would present technical disadvantages when compared to our approach:

- Our syntax extends TRS, while (2) does not, but uses instead a lower level representation encoding a way of guiding lazy evaluation. As a consequence, comparison to ordinary rewriting would be far less obvious than in our case (see next section).

- Our rules for *rt-let*-rewriting allow *let*s to disappear, creating constructor nestings, and finished derivations reach contructor terms. In contrast, heaps in (2) never disappear, only grow, and derivations reach only head normal forms. Again, all this would make more difficult establishing the relationship to pure run-time or call-time choice.

[1] For the sake of clarity, we rename the '*let*-rewriting' relation of (21) to '*ct-let*-rewriting', to distinguish it from *rt-let*-rewriting.

4. *Rt-let*-rewriting as a conservative extension

We have presented a (run-time choice) rewriting notion able to express sharing by means of an explicit *let* construction in program rules. The purpose of this section is to show with technical care that the resulting framework indeed generalizes pure run-time choice – as realized by ordinary rewriting– and pure call-time choice –as realized by the *CRWL* approach (14; 21)–.

Regarding the first statement – *rt-let*-rewriting generalizes ordinary rewriting – first of all let us recall the formal definition of term rewriting. For any program $\mathcal{P}$ its associated *rewrite relation* $\rightarrow_\mathcal{P}$ is defined as $\mathcal{C}[l\sigma] \rightarrow_\mathcal{P} \mathcal{C}[r\sigma]$ for any context $\mathcal{C}$, rule $l \rightarrow r \in \mathcal{P}$ and $\sigma \in Subst$. Notice that σ can instantiate extra variables to any expression. In the following, we will usually omit the reference to $\mathcal{P}$. Now proving that first statement is fairly straighforward: if *let*s do not appear in a program $\mathcal{P}$, then every step of ordinary rewriting is a valid *rt-let*-rewriting step performed by the rule (Fapp) of Fig. 1, because the absence of *let*s implies that $BV(\mathcal{C}) = \emptyset$ for any context $\mathcal{C}$, which guarantees the condition *i)* in Fig. 1. Besides, if a step $e \rightarrow^{rt} e'$ has been performed for $e, e' \in Exp$, then the only rule which may have been applied is (Fapp), and besides no *let* could have been used to instantiate the extra variables: thus the step is also an ordinary rewriting step. Therefore, we have:

THEOREM 1 (*Rt-let*-rewriting extends rewriting). *If $\mathcal{P}$ is a program without lets (i.e., $\mathcal{P} \in Program$), then:*

$$e \rightarrow_\mathcal{P} e' \Leftrightarrow e \rightarrow^{rt}_\mathcal{P} e', \textit{ for any } e, e' \in Exp.$$

To compare *rt-let*-rewriting with the call-time choice semantics provided by *CRWL* is more complicated, even having at our disposal an equivalent rewrite notion for call-time choice like the *ct-let*-rewriting relation of (21). As we saw in Section 3.3, despite their rough similarity, both relations are quite different; as a matter of fact, they are incomparable step by step. Nevertheless, in (21) a conservative extension of *CRWL* called $CRWL_{let}$ was also presented, a logic for call-time choice applicable to programs containing *let*s, i.e., to our core programs. This will be the main tool we will use to show that, for any program $\mathcal{P}$, its transformed program $\tau(\mathcal{P})$ behaves, under *rt-let*-rewriting, as $\mathcal{P}$ with respect to call-time choice.

The semantics of a program $\mathcal{P}$ is determined in $CRWL_{let}$ by means of a proof calculus able to derive reduction statements of the form $e \twoheadrightarrow t$, with $e \in LExp_\bot$ and $t \in CTerm_\bot$, 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 $CRWL_{let}$-proof calculus is presented in Fig. 2.

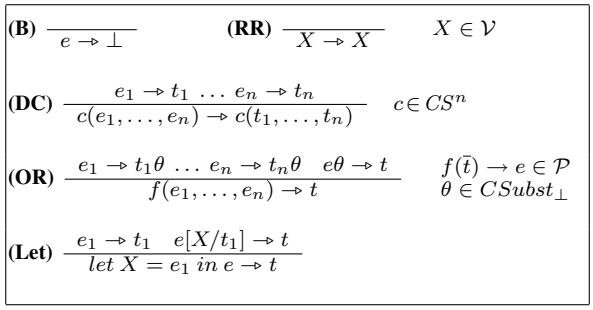

Figure 2. Rules of $CRWL_{let}$

Rule **B** (bottom) allows any expression to be undefined or not evaluated (non-strict semantics). Rule **OR** (outer reduction) ex-

presses that to evaluate a function call we must choose a compatible program rule, perform parameter passing (by means of a $CSubst_\bot\ \theta$) and then reduce the right-hand side. The use of partial c-substitutions in **OR** is essential to express call-time choice, as only single partial values are used for parameter passing; 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 $\to_{\mathcal{P}}$. Finally rule **Let** expresses that to evaluate an expression $let\ X = e_1\ in\ e_2$ first we must calculate a partial value for e_1, then propagate it to e_2 through a substitution, and continue the evaluation on the resulting expression. This way e_1 is evaluated only once even when its corresponding variable X may appear more than once in the body e_2, thus achieving the sharing of the value for e_1.
We write $\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t$ to express that $e \twoheadrightarrow t$ is derivable in the $CRWL_{let}$-calculus using the program $\mathcal{P}$. Given a program $\mathcal{P}$, the *CRWL$_{let}$-denotation* of an expression $e \in LExp_\bot$ is defined as $[\![e]\!]^{\mathcal{P}} = \{t \in CTerm_\bot \mid \mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t\}$. The *hypersemantics* gives a more active role to variables in the expression; it is the function $[\![e]\!]^{\mathcal{P}} : CSubst_\bot \to \mathcal{P}(CTerm_\bot)$ defined by $[\![e]\!]^{\mathcal{P}}\theta = [\![e\theta]\!]^{\mathcal{P}}$. Hypersemantics are useful to characterize the meaning of expressions present in a context in which some of its variables may get bound, like in the body of a *let* or in the right hand side of a program rule. In the following, we will usually omit the reference to $\mathcal{P}$. Semantics of expressions can be ordered by set inclusion, and hypersemantics are ordered by:

$$[\![e]\!] \Subset [\![e']\!] \Leftrightarrow \forall\theta.[\![e]\!]\theta \subseteq [\![e']\!]\theta\ \ (\Leftrightarrow \forall\theta.[\![e\theta]\!] \subseteq [\![e'\theta]\!]).$$

The expected property of τ is that $\tau(\mathcal{P})$, if executed under *rt-let*-rewriting, behaves as $\mathcal{P}$, if executed under call-time choice (as given by $CRWL_{let}$). In other words, τ serves to simulate call-time choice within a run-time choice framework enhanced with a *let* primitive to express sharing of values. To prove it we start by showing that τ is harmless when performed in a call-time choice ambient, i.e., τ preserves *CRWL$_{let}$*-(hyper)semantics:

THEOREM 2 (Adequacy of τ under $CRWL_{let}$).
For any $\mathcal{P} \in Program$ and $e \in LExp$ we have $[\![e]\!]^{\mathcal{P}} = [\![e]\!]^{\tau(\mathcal{P})}$. In particular, $[e]^{\mathcal{P}} = [e]^{\tau(\mathcal{P})}$.

We now address the *soundness* of τ as simulation of call-time choice: we show that $\tau(\mathcal{P})$, executed with *rt-let*-rewriting, does not produce new results when compared to $\mathcal{P}$ with call-time choice. To that purpose, the basic technical result is the following one, stating that at each step $e \to^{rt} e'$ done with $\tau(\mathcal{P})$, the hypersemantics of the reduced expression e does not grow (it might decrease due to non-determinism if (Fapp) was used for the step).

LEMMA 1. *For any $\mathcal{P} \in Program$, $e, e' \in LExp$,*

$$e \to^{rt}_{\tau(\mathcal{P})} e' \Rightarrow [\![e']\!]^{\tau(\mathcal{P})} \Subset [\![e]\!]^{\tau(\mathcal{P})}.$$

As a consequence, chaining several $\to^{rt}$-steps and taking into account that $[\![e]\!] \Subset [\![e']\!]$ implies $[\![e]\!] \subseteq [\![e']\!]$, we obtain the following:

THEOREM 3. *For any $\mathcal{P} \in Program$, $e, e' \in LExp$,*

$$e \to^{rt\,*}_{\tau(\mathcal{P})} e' \Rightarrow [\![e']\!]^{\tau(\mathcal{P})} \subseteq [\![e]\!]^{\tau(\mathcal{P})}.$$

With this and the aid of Theorem 2, it is now straightforward to formulate our desired soundness result:

THEOREM 4 (Soundness of τ).
For any $\mathcal{P} \in Program$, $e \in LExp$, $t \in CTerm$,

$$e \to^{rt\,*}_{\tau(\mathcal{P})} t \ \ \Rightarrow\ \ \mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t.$$

The next goal is proving *completeness* of the simulation, i.e., the reciprocal of Th. 4. The technical key for it is the following result, ensuring that any value in the *CRWL$_{let}$*-semantics of an expression e can be covered by a $\to^{rt}$ derivation starting from e.

LEMMA 2 (Completeness lemma for $\to^{rt}$).
For any $\mathcal{P} \in Program_{let}$, $e \in LExp, t \in CTerm_\bot$,

$$\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t \ \ \Rightarrow\ \ e \to^{rt\,*}_{\mathcal{P}} e'$$

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

Notice that the lemma, being a completeness result, does not mention the transformed program, and therefore constitutes a formal proof of the intuitive fact that the *CRWL$_{let}$*-semantics, designed to express call-time choice, cannot give more results than the more liberal *rt-let*-rewriting, a result which is interesting in itself.

If we apply Lemma 2 to $t \in CTerm$ (i.e., t is total), then $t \sqsubseteq |e'|$ means $t \equiv |e'|$, which in particular implies that there is no function application in $|e'|$. One could expect then that the *let*-bindings that could remain in e' could be eliminated by some $\to^{rt}$-steps, and therefore that for t total $\mathcal{P} \vdash e \twoheadrightarrow t$ implies $e \to^{rt\,*}_{\mathcal{P}} t$. However, this cannot be guaranteed for total but not ground t, because a variable X in t, which is free, can appear in e' inside a *let*-binding $let\ Y = X\ in\ \ldots$ that cannot be dropped off because of the condition *ii)* imposed to $\to^{rt}$ in Fig. 1.

Which can be proved is the following:

THEOREM 5 (Completeness of $\to^{rt}$ wrt *CRWL$_{let}$*).
For any $\mathcal{P} \in Program$, $e \in LExp$, and $t \in CTerm$,

$$\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t \ \ \Rightarrow\ \ e \to^{rt\,*}_{\tau(\mathcal{P})} let\ \overline{Y = X}\ in\ t'$$

for some $t' \in CTerm$ such that $t'[\overline{Y/X}] \equiv t$ and $\overline{X} \subseteq FV(t)$. If in addition t is ground, then $e \to^{rt\,*}_{\tau(\mathcal{P})} t$.

Joining all these completeness results with the previous soundness results and the equivalence of $\mathcal{P}$ and $\tau(\mathcal{P})$ wrt. *CRWL$_{let}$*, it is not difficult now to obtain the adequacy (soundness + completeness) of the transformation τ to express call-time choice under an overall run-time choice regime.

THEOREM 6 (Adequacy of τ).
For any $\mathcal{P} \in Program$, $e \in LExp$, $t \in CTerm_\bot$,

a) $\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t \ \Leftrightarrow\ e \to^{rt\,*}_{\tau(\mathcal{P})} e'$, for some e' such that $|e'| \sqsupseteq t$.
b) If $t \in CTerm$ (i.e., t is total), then:

$$\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t \ \Leftrightarrow\ e \to^{rt\,*}_{\tau(\mathcal{P})} let\ \overline{Y = X}\ in\ t'$$

for some $t' \in CTerm$ with $t'[\overline{Y/X}] \equiv t$ and $\overline{X} \subseteq FV(t)$.
c) If t is total and ground, then $\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t \Leftrightarrow e \to^{rt\,*}_{\tau(\mathcal{P})} t$.

5. Discussion: could it be simpler?

In this section we examine with some detail other possibilities to achieve the integration of run-time and call-time choice. First of all, we showed in (21) that no program transformation can perfectly mimic call-time choice within ordinary rewriting (i.e., within run-time choice without *let*s) due to their different closedness properties under substitutions. We show here that the opposite perfect imitation (run-time choice within call-time choice) is not possible either, in this case due to different compositionality properties of both kind of choices. We include the proof because of its remarkable simplicity, thanks to the strength of some essential results about semantics of call-time choice ((21; 23)).

THEOREM 7. *There are programs $\mathcal{P}$ for which no program $\mathcal{P}'$ can verify the following property* (P)*:*

$$e \to_{\mathcal{P}}^{rt^*} t \Leftrightarrow \mathcal{P}' \vdash_{CRWL_{let}} e \twoheadrightarrow t$$

for any ground $e \in Exp$, $t \in Cterm$.

Proof. The following simple program suffices:

$$\mathcal{P} \equiv \{f(X) \to (X, X),\ coin \to 0,\ coin \to 1\}$$

Assume there exists $\mathcal{P}'$ verifying *(P)*. Since $f(coin) \to_{\mathcal{P}}^{rt^*} (0,1)$, *(P)* implies that $\mathcal{P}' \vdash_{CRWL_{let}} f(coin) \twoheadrightarrow (0,1)$. By a compositionality property of call-time choice (see e.g. (23), Th. 1), there must exist a $t \in CTerm_{\bot}$ such that $\mathcal{P}' \vdash_{CRWL_{let}} coin \twoheadrightarrow t$ and $\mathcal{P}' \vdash_{CRWL_{let}} f(t) \twoheadrightarrow (0,1)$. Now we distinguish some cases depending on the value of t (notice that t might be partial):

(a) If $t \equiv \bot$, then monotonicity of $CRWL$-derivability ((14)) proves that $\mathcal{P}' \vdash_{CRWL_{let}} f(s) \twoheadrightarrow (0,1)$ for any $s \in CTerm_{\bot}$, in particular $\mathcal{P}' \vdash_{CRWL_{let}} f(0) \twoheadrightarrow (0,1)$. Then, again by *(P)*, $f(0) \to_{\mathcal{P}}^{rt^*} (0,1)$, but this is not true.

(b) If $t \equiv 0$, then $\mathcal{P}' \vdash_{CRWL_{let}} f(t) \twoheadrightarrow (0,1)$ leads to a contradiction as in (a). The cases $t \equiv 1$, $t \equiv Y$ or $t \equiv c(\overline{s})$ for a constructor c different from $0, 1$ lead to similar contradictions.

Some facts to be noticed: first, the program used in the proof is an ordinary CTRS (it does not use *let*s at all), and therefore the relation $\to^{rt}$ could by replaced by ordinary rewriting $\to$ (Th. 1) along Th. 7 and its proof. Second, the groundness restriction for e, t in the theorem is not a weakness, but quite the opposite (since Th. 7 trivially implies the proposition dropping the groundness restriction). Third, the result is true even if transformed programs $\mathcal{P}'$ are allowed to be HO in the sense of (11), since the properties of *CRWL*-semantics used in the proof are also true for such HO extension.

Theorem 7 does not preclude the existence of other more sophisticated program transformations that, by changing the representation of expressions, could be suitable to express run-time choice within existing systems that use call-time choice (e.g., *Curry* (16) or *Toy* (25)). In particular, it is a kind of folkclore of the functional logic community that an old well-known HO technique (1) for delaying evaluation, based on the fact that partial applications are not evaluated, can help in the simulation of run-time choice within call-time choice. We discuss it now with the aid of Example 1, where we encountered the problem of achieving run-time choice behavior for *star*. To clarify the discussion we use HO syntax and types (as existing systems do). The trick consists in replacing the original definitions of *String* generators like *letter, word, palindrome*, which had type *String* (an alias for *[Char]*), by new functions of type *() → String* (here *()* plays the role of a *dummy* type). The type of *star* would be changed also to *star:: (() → String) → (() → String)*, and the program will be recoded as (we show only a part):

letter () → "a" ..... *letter () → "z"*
word () → star letter ()
star X () → "" *star X () → (X ()) ++ star X ()*

Now *letter* and *(star letter)* are partial applications, and *word ()* evaluates to *"ab"*, among other values, so that a run-time choice behavior for *star* has been achieved. This is a nice trick, used for parsing in (8), but has some noticeable drawbacks and limitations, when compared to our approach:
(i) It requires to change the natural type of functions: moreover that change is global, and not localized in the functions for which one desires run-time choice behavior. If one wants generality and allows the inclusion of run-time functions at any point in the program, then the types of *every* function f need to be artificially changed with dummy arguments, and consequently every expression to be evaluated has to be adapted to f's new type and definition. For example the expression *star "a"* is not well typed anymore and we must use *star aux*, where *aux* is a new symbol defined by the rule *aux () → "a"*. Thus the resulting code is much less natural.
(ii) An even more serious problem is that the trick is not general enough as to deal with matching. Consider, for instance, that we want a run-time choice regime for a function f defined as *f['a' | Xs] → (Xs,Xs)*, so that *f(word)* can be reduced to *("a","b")*, among (infinitely many) other values. What type should be assigned to f in the HO-encoding? If we keep the 'original' type *f:: String → (String,String)*, then f cannot be applied directly to *word*; instead, we must consider *f (word ())*, but this can be reduced to *("a","a")* or *("b","b")* but not to *("a","b")*. Switching to the type *f:: (() → String) → (String,String)* does not solve the problem, because any suitable definition for f needs to do some evaluation work with its argument (in order to match it with *['a' | X]*); but, at this point, what else can be done with an argument of type *() → String* except applying it to *()*, thus losing run-time choice behavior for f? Trying to overcome the problem we could think of a re-revision of all types, but it is fairly unclear how to do that, and anyhow it shows that a general technique to encode run-time choice in a host HO typed language following call-time choice can be rather cumbersome (if possible at all, which remains as an interesting open question).
(iii) It requires to use HO to express FO run-time, thus mixing unnecessarily two concerns. Moreover, it is known (see e.g. (23)) that HO functions with call-time choice have subtle behaviors, so their use cannot be alleged to be free of surprises for the programmer.

In contrast to all this, our approach:
(i) seamlessly integrates types (the distinction run-time/call-time is irrelevant for types) and matching (nothing special must be done),
(ii) is more modular due to its local flavor (adopting call-time for a function affects only to its definition),
(iii) keeps the concerns FO/HO separated, and therefore could be more easily adapted to existing systems or frameworks that are directly based in FO rewriting (e.g., Maude (9)). The extension of the framework to HO can be addressed as an independent matter, realizable in standard ways followed in other works: adapting the theory to HO (11), adopting a FO translation (12; 5), or both (23). In such a HO extension the management of call-time choice could be made even more modular and abstract through a HO polymorphic function *call_time F X → let Y = X in F Y*. With this function (that can be generalized to greater arities) we can get call-time versions of functions following other regimes,
(iv) last but not least, we give formal foundations to our approach, while nothing similar does exist for the HO-approach to simulation of run-time choice within call-time choice (and the question is not trivial, as argued before).

A final comment: after developing this approach, we have tried in (24) an alternative approach to the combination of run-time and call-time choice. Instead of starting form ordinary rewriting and enhance it with a construct to express call-time choice, we have followed in (24) the opposite way: add to a call-time choice based language annotations of the form $rt(e)$ to indicate that evaluation of e is kept out of the sharing regime of call-time choice. However, the resulting framework seems to be less expresive and with no so clean formal properties, when compared to the approach here. On the positive side, the proposal of (24) has been easy to implement on top of the system *Toy* (25).

6. Final summary and discussion

We have proposed a new formal framework for (first order) programming with non-deterministic functions. The novelty is that, in contrast to existing languages where a decision is taken a pri-

ori about the semantics of non-determinism adopted for functions (run-time or call-time choice), with our approach both semantics can be used within the same program, which reveals itself as a very useful resource in many cases.

This has been done through a language with annotated functions (each one with its intended semantics) which are transformed into a core language of term rewriting systems with local bindings. Reductions in the core language are made by *rt-let*-rewriting, a one-step rewriting relation specifically designed to realize the combination of semantics. We have ensured that the framework is a safe extension of both pure run-time choice and pure call-time choice. All the notions and results are supported by rigorous definitions and technically detailed proofs.

The main underlying ideas being quite simple –annotate functions, translate programs into ordinary rewriting enhanced with local bindings–, two false impressions might arise: that existing frameworks are sufficient to achieve the combination of semantics, or that proving properties of the combined framework is a routine task. Sect. 5 gets rid of the first illusion; regarding the second issue, it is interesting to observe that the proof of adequacy of our simulation of call-time choice (Sect. 4), besides of not being trivial, relies heavily on semantic properties of $CRWL_{let}$ (some of them new, see (22)), in a new strong evidence of the power, argued in (23), of using semantics to prove results about functional logic reductions.

The decision of annotating each function as *rtc* or *ctc* has been taken for the sake of clarity in the conception of the language, and as a kind of discipline that could help programmers to use the combination of semantics. However, we remark that the syntax of the core language, using explicit *let*s, could be put at programmer's disposal when writing programs. Notice that the results in the paper cover the case of programs using this core syntax, and therefore its use is technically justified. The disadvantage of proceeding so is that programs become more obscure to read and with less predictable behavior if one makes a wild use of *let*s. But we find also some *pros*: the explicit use of *let*s gives a finer control of sharing, since we can choose specific behavior (shared/non shared) to each piece in an expression. For instance, we could write a defining rule with the form $f(X, [Y|Ys]) \rightarrow \ let\ U = X\ in\ let\ V = Ys\ in\ e$, indicating that the first argument of f and a part, but not the whole second one, are shared.

Having on hand simultaneously run-time choice and call-time choice (a non-sharing and a sharing procedure, respectively) is useful not only for programming purposes, but also for devising and justifying in a formal basis program transformations or implementation techniques. As an example consider the function *repeat'*, similar to *repeat* of Sect. 1, but in this case programmed to follow call-time choice:

repeat'(X) → let Y=X in [Y|repeat'(Y)]

With this definition, an expression of the form *repeat'(e)* reduces to the expression *let Y=e in [Y|repeat'(Y)]*, and therefore recursive invocations to *repeat'* (and there might be an arbitrarily large number of them in a lazy computation) generate successive *let*-bindings *let Z=Y in [Z|repeat'(Z)]*, etc. However, intuitively only the first *let Y=e* is really needed, since then Y is already a shared value for which new sharing is useless. This suggests (automatically) replacing the original definition of *repeat'* by an optimized variant

repeat'(X) → let Y=X in [Y|repeat(Y)]

where *repeat* is the *rtc* function of Sect. 1, i.e., is defined by the rule *repeat(X) → [X|repeat(X)]*, thus avoiding the useless *let*s. We see some analogy between these *let*-binding savings described here and the implementation of sharing in some *Curry* systems (4) that try to avoid unnecessary creation of *suspensions*. A thorough investigation of these issues is left for future work. We simply remark here the potential applicability of our framework as a suitable formalism for making and proving precise statements.

We contemplate other relevant subjects of future work:

- *Rt-let*-rewriting, as presented here, still misses two pieces to become a fully practical operational procedure. One is an evaluation strategy to be imposed in the space of possible $\rightarrow^{rt}$-derivations. It seems, however, that adapting standard strategies of existing functional logic languages based on *definitional trees* (see e.g. (15)) would not be difficult. The second one is that, as commented in Sect. 3.2, the notion of rewriting given here should be lifted to a notion of narrowing, as was done in (20; 23) for the case of call-time choice. Notice, however, that narrowing is needed only in the case of initial expressions having variables, or when programs have extra variables in right-hand sides.
- We must invest some effort in producing a complete implementation of our approach (including HO functions and types) and a larger collection of program examples and programming patterns that make sensible use of the combination of run-time choice and call-time choice. From them, we should gain more insights about how, when and why making use of the combination.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[2] 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.

[3] S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. Algebraic and Logic Programming (ALP'97)*, 16–30. Springer LNCS 1298, 1997.

[4] S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. Frontiers of Combining Systems (FroCoS'00)*, 171–185, Springer LNCS 1794, 2000.

[5] S. Antoy and A. P. Tolmach. Typed higher-order narrowing without higher-order strategies. In *Proc. Fuji Int. Symp. on Functional and Logic Programming (FLOPS'99)*, 335–353, Springer LNCS 1722, 1999.

[6] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, 1997.

[7] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. In *Proc. ACM Symposium on Principles of Programming Languages (POPL'95)*, 233–246, 1995.

[8] R. Caballero-Roldán and F. López-Fraguas. Parsing with non-deterministic functions. In *Proc. APPIA-GULP-PRODE'98*, 1–16, 1998.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude*. Springer LNCS 4350, 2007.

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

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

[12] J. C. González-Moreno. A correctness proof for Warren's HO into FO translation. In *Proc. GULP'93*, 569–584, 1993.

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

[14] 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.

[15] M. Hanus. Multi-paradigm declarative languages. In *Proc. Int. Conf. on Logic Programming (ICLP'07)*, 45–75. Springer LNCS 4670, 2007.

[16] 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.

[17] J. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107,1989.

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

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

[20] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Narrowing for non-determinism with call-time choice semantics. In *Proc. Workshop on Logic Programming (WLP'07), Tech. Rep. 434 Univ. Wurzburg*, 224–233, 2007.

[21] 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 (PPDP'07)*, 197–208. ACM Press, 2007.

[22] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A Flexible Framework for Programming with Non-deterministic Functions (Extended version) `http://gpd.sip.ucm.es/juanrh/pubs/tchrRTCT08.pdf`, 2008.

[23] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In *Proc. Fuji Int. Symp. on Functional and Logic Programming (FLOPS'08)*, 147–162, Springer LNCS 4989, 2008.

[24] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A Lightweight Combination of Semantics for Non-deterministic Functions. In *Proc. Workshop on Logic Programming Environments (WLPE'08)*, 2008.

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

[26] D. Plump. Essentials of term graph rewriting. *ELec. Notes on Theoretical Computer Science*, 51, 2001.

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

[28] D. H. Warren. Higher-order extensions to Prolog: are they needed? In J. Hayes, D. Michie, and Y.-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd., 1982.