A Flexible Framework for Programming with Non-deterministic Functions*

(Extended version)

Tech. Rep. SIC-9-08, 2008

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

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

Abstract. The use of non-deterministic functions is a distinctive feature of modern functional logic languages. The semantics commonly adopted is *call-time choice*, a notion that at the operational level is related to the *sharing* mechanism of lazy evaluation in functional languages. However, there are situations where *run-time choice*, closer to ordinary rewriting, is more appropriate. In this paper we propose a unified formal framework where both semantics can co-exist for the same progam. This is done through 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 call-time choice based languages can be embedded by means of a simple program transformation introducing *let*s in function definitions. We prove the adequacy of the embedding, as well as other relevant properties of the framework.

1 Introduction

Non-strict non-deterministic functions are a distinctive feature of modern functional logic languages (see [16] for a recent survey). It is known that the introduction of non-determinism in a functional setting gives rise to a variety of semantic decisions (see e.g. [26]). 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. Operationally, this needs something similar to the sharing mechanism followed, by efficiency reasons, in (deterministic) functional languages under lazy evaluation. In contrast, run-time choice does not share, corresponds rather to call-by-name, and is realized by ordinary rewriting. For deterministic programs, run-time and call-time are able to produce the same set of values, but in general the set of values reachable by run-time choice is larger than that of call-time choice.

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

Non-deterministic functions with non-strict and call-time choice semantics were introduced in the functional logic setting with the *CRWL* framework [14, 15], 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 [17] or Toy [23]. Run-time choice has been rarely [2] considered as a valuable global alternative to call-time choice.

However, there might be parts in a program or individual functions for which run-time choice could be a better option, and therefore it would be convenient to have both possibilities (run-time/call-time) at programmer's disposal. The purpose of this work is precisely proposing a clear, well-founded formal framework for doing that. The following example illustrates the interest of combining both semantics.

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):

$$letter \to \text{"a"} \;\; .... \;\; letter \to \text{"z"} \qquad word \to \text{""} \qquad word \to letter{+}{+}word$$

Disregarding syntax, that CTRS is a valid program in functional logic systems like Curry or Toy. Each individual reduction leads to a string. The generation of palindromes (of even length) could be done by the rewrite rules:

$$palindrome \to palAux(word) \qquad palAux(X) \to X \;{+}{+}\; reverse(X)$$

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

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

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:

$$X \mid Y \to X \qquad X \mid Y \to Y \qquad star(X) \to \text{""} \qquad star(X) \to X{+}{+}star(X)$$

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

$$letter \to \text{"a"} \mid \text{"b"} \mid ... \mid \text{"z"} \qquad word \to star(letter)$$

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. This problem was pointed out in [7], where a 'higher order trick' was suggested to overcome it. We discuss in depth that trick in Sect. 5. Notice, however, that it would be much simpler to consider that *star* follows run-time choice regime, so that the occurrences of *letter* created by *word* 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 *palindrome* requires call-time, while *star* requires run-time.

To the best of our knowledge, no existing proposal offers the possibility of combining in the same program both kind of semantics. This paper addresses that problem at

a foundational level, deferring for the future the matters of implementing a concrete system or developing larger practical applications.

Our approach to combining run-time/call-time develops a natural idea: enhance run-time choice (i.e., ordinary rewriting) with a *let*-construction for local bindings to be governed by operational rules expressing the kind of sharing needed by call-time choice. We will call the enhanced rewriting relation *rt-let-rewriting*. In Example 1, the definitions of *letter*, *word* or *star* would remain the same. However, *palAux* would be encoded as $palAux(X) \to let\ Y = X\ in\ Y\ ++\ reverse(Y)$ to ensure call-time choice for it. Or better, we can dispense with *palAux* and define directly $palindrome \to let\ Y = word\ in\ Y\ ++\ reverse(Y)$.

In spite of obvious general similarities, the use of *let*s in *rt-let*-rewriting must not be confused with other uses of local bindings in related scenarios, although of course some general similarities remain:
• *local definitions* of existing functional logic languages ([17, 23]): 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 ([6, 5]): they formalize sharing in lambda-calculus, but have nothing to do with non-determinism. Moreover, the underlying formalism is *lambda*-calculus instead of term rewriting.
• *let*s of *ct-let*-rewriting[1], proposed in [20] as a notion of one-step reduction adequately reflecting *CRWL*'s lazy call-time choice while avoiding the complexity of term graph rewriting [24, 11]. That use of *let*s 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 will be liberal (as ordinary rewriting is), and computations will 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 [20] and in this paper are clearly different.

The rest of the paper is organized as follows. Section 2 contains some technical preliminaries and notations about term rewriting systems. Section 3 introduces rewrite systems with *let*-bindings, presents the precise notion of *rt-let*-rewriting, and prove some of its properties. Section 4 shows that our new framework is a conservative extension of both pure run-time and pure call-time choice. The former will be almost obvious, while for the latter we will propose a program transformation $\mathcal{P} \mapsto \tau(\mathcal{P})$ introducing the necessary *let*s in $\mathcal{P}$, so that the behavior of $\mathcal{P}$ under call-time choice (as determined by the CRWL-semantics) and the behavior of $\tau(\mathcal{P})$ under run-time choice (as determined by *rt-let*-rewriting) coincide. In Section 5 we discuss in detail the question of wether our approach could be replaced by simpler ones; we point out some limits in the ability of run-time and call-time 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 [21].

[1] For the sake of clarity, we rename the '*let*-rewriting' relation of [20] to '*ct-let*-rewriting'.

2 Preliminaries

Constructor-based term rewrite systems

We consider a first order signature $\Sigma = CS \cup FS$, where CS and FS are two disjoint set of *constructor* and defined *function* symbols respectively, all them with associated arity. We write CS^n (FS^n resp.) for the set of constructor (function) symbols of arity n. We write $c, d, \ldots$ for constructors, $f, g, \ldots$ for functions and $X, Y, \ldots$ for variables of a numerable set $\mathcal{V}$. The notation $\overline{o}$ stands for tuples of any kind of syntactic objects.

The set Exp of *expressions* is defined as $Exp \ni e ::= X \mid h(e_1, \ldots, e_n)$, where $X \in \mathcal{V}$, $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. The set $CTerm$ of *constructed terms* (or *c-terms*) is defined like Exp, but with h restricted to CS^n (so $CTerm \subseteq Exp$). The intended meaning is that Exp stands for evaluable expressions, i.e., expressions that can contain function symbols, while $CTerm$ stands for data terms representing values. We will write $e, e', \ldots$ for expressions and $t, s, \ldots$ for c-terms. The set of variables occurring in an expression e will be denoted as $var(e)$. We will frequently use *one-hole contexts*, defined as $Cntxt \ni \mathcal{C} ::= [\,] \mid h(e_1, \ldots, \mathcal{C}, \ldots, e_n)$, with $h \in CS^n \cup FS^n$. The application of a context $\mathcal{C}$ to an expression e, written by $\mathcal{C}[e]$, is defined inductively as $[\,][e] = e$ and $h(e_1, \ldots, \mathcal{C}, \ldots, e_n)[e] = h(e_1, \ldots, \mathcal{C}[e], \ldots, e_n)$.

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 $vran(\theta) = \bigcup_{X \in dom(\theta)} var(X\theta)$. *C-substitutions* $\theta \in CSubst$ verify that $X\theta \in CTerm$ for all $X \in dom(\theta)$.

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

Local bindings. The $CRWL_{let}$ framework

As explained in Section 1, in [20] we already considered local bindings in programs and expressions, but only for the purpose of characterizing call-time choice as a one-step reduction relation, *ct-let*-rewriting, that was proved to be equivalent to the semantics given by *CRWL*. As an auxiliary tool, we needed to extend the *CRWL* logic of [15] to the more general $CRWL_{let}$, a logic for call-time choice applicable to programs containing *let*s. In this section we briefly recall syntactic aspects of local bindings, as well as the $CRWL_{let}$ logic, that will be used later on. *Let-expressions* are 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 $let\ X_1 = a_1\ in\ \ldots\ in\ let\ X_n = a_n\ in\ e$. The notion of context is also extended to the new syntax: $\mathcal{C} ::= [] \mid let\ X = \mathcal{C}\ in\ e \mid let\ X = e\ in\ \mathcal{C} \mid h(\ldots, \mathcal{C}, \ldots)$.

From this point on, we assume 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 $e \in LExp$. However, we must stress the fact that in the $CRWL_{let}$ framework of [20] all programs, irrespective to the fact of using explicit *let*s or not, are to be assigned a call-time choice semantics. The main role of *let*s there is that they are introduced in *ct-let*-rewriting steps (see [20, 21] for the rules) precisely to ensure call-time choice. If *let*s are allowed in $CRWL_{let}$-programs is just for the sake of generality, but it is not difficult to realize that, within $CRWL_{let}$, any program using explicit *let*s has a semantically equivalent one with no *let*s at all.

The sets $FV(e)$ and $BV(e)$ of *free* and *bound* variables of $e \in LExp$ are defined as usual (see [20]). We assume a variable convention according to which the same variable symbol does not occur free and bound within an expression. Moreover, we assume 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:

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

Free variables of contexts are defined as for expressions, so that $FV(\mathcal{C}) = FV(\mathcal{C}[\bot])$ $(= FV(\mathcal{C}[a])$, 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 $c \in CS^n$, $e_1,\ldots,e_n \in LExp$.

As usual with non-strict languages, in order to express the semantics of expressions and programs the signature is enhanced with a new constant constructor symbol $\bot$, to represent the undefined value. Each syntactic domain $\mathcal{D} \in \{Subst, CSubst, Exp, LExp\}$ can be enlarged to the corresponding $\mathcal{D}_\bot$ of partial substitutions, 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 $CRWL_{let}$-logic defines the derivability relation $\mathcal{P} \vdash e \twoheadrightarrow t$, where $\mathcal{P}$ is a program, $e \in LExp_\bot$, $t \in CTerm_\bot$, indicating that t is an $\sqsubseteq$-approximation to a possible value for e, calculated with $\mathcal{P}$ according to call-time choice semantics. The inference rules defining this relation can be found in [20, 21].

Given a program $\mathcal{P}$, the approximated values of an expression $e \in LExp_\bot$ are collected in its $CRWL_{let}$-denotation $[\![e]\!]^{\mathcal{P}} = \{t \in CTerm_\bot \mid \mathcal{P} \vdash e \twoheadrightarrow t\}$. 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 $[\![e]\!]^{\mathcal{P}}\theta = [\![e\theta]\!]^{\mathcal{P}}$. The mention to $\mathcal{P}$ is frequently omitted. 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 *shell* $|e|$ of an expression $e \in LExp_\bot$ is a partial c-term representing 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:

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

Shells verify $|e| \in [\![e]\!]$, for any program and $e \in LExp_\bot$.

The *ct-let*-rewriting relation $\rightarrow^{ct}$ is proposed in [20] (where it was written $\rightarrow^{l}$) as a one-step rewriting relation corresponding to the $CRWL_{let}$-semantics. The rules governing $\rightarrow^{ct}$ can be found in [20, 21]. The main result in [20] is the equivalence of $CRWL_{\rm let}$-derivability and $\rightarrow^{ct}$-reachability:

Theorem 1 ([20]). *$e \rightarrow^{ct^*}_{\mathcal{P}} t \Leftrightarrow \mathcal{P} \vdash e \twoheadrightarrow t$ ($\Leftrightarrow t \in [\![e]\!]^{\mathcal{P}}$), for any $\mathcal{P}, e \in LExp, t \in CTerm$.*

3 Run time choice with local bindings

We present here our framework for run-time choice with *let*-bindings. Syntactically, the family of programs is the same of $CRWL_{let}$, but the point of view changes completely, as argued in Section 1. In *rt-let*-rewriting –to be defined below– the reduction process does not create new *let*s, but only manages them conveniently. Therefore, writing explicit *let*s is essential in those points where the programmer wants a call-time choice behavior. Explicit *let*s in programs provide a great flexibility to the programmer, who can choose a specific behavior (shared/non shared) for each piece in a program rule. 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.

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 [20], 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; that issue has been addressed for *ct-let*-rewriting in [19, 22], but for the case of *rt-let*-rewriting 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.

Now we will define the run-time rewriting relation with local bindings (or *rt-let*-rewriting), written $\rightarrow^{rt}$ (or $\rightarrow^{rt}_{\mathcal{P}}$ if the program $\mathcal{P}$ is made explicit). To do this we will first define an auxiliary relation $\rightarrow^{rt'}$ for rewriting steps at the root position of an expression; a $\rightarrow^{rt}$-step is then defined as a $\rightarrow^{rt'}$-step put in context and fulfilling some additional conditions. In the following definition $\mathcal{P}$ is a program, $X, Y, Z \in \mathcal{V}$, $f \in FS, h \in FS \cup CS, t \in CTerm, e, e_i, a \in LExpr$, and $\mathcal{C}, \mathcal{C}' \in Cntx$.

Definition 1 (Run-time let rewriting relation $\rightarrow^{rt}$). *The auxiliary relation $\rightarrow^{rt'}$ is defined by the following 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* $e \in \mathcal{V}$.

Now, for any $\mathcal{C} \in Cntx$ we define $\mathcal{C}[e] \rightarrow^{rt} \mathcal{C}[e']$, if $e \rightarrow^{rt'} e'$ using any of the previous rules, and the following conditions hold, depending on the form of $e \rightarrow^{rt'} e'$:

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 $Y \notin BV(\mathcal{C})$.

Some explanations about the rules follow. Rule **(Fapp)** allows 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 $\rightarrow^{rt}$-rules forget about the program and deal only with *let*-bindings. An important intuition is that if a step $e \rightarrow^{rt'} e'$ is performed using any of these rules that are independent from the program, then the set of $\rightarrow^{rt}$-reachable values (i.e. constructor terms) will be the same for e and e'. Therefore all non-determinism 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 $\rightarrow^{rt}$ 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 $let\ Y = X\ in\ (Y, Y) \rightarrow^{rt} (X, X)$ would be allowed; however, some of its particular cases (replacing the free variable X by concrete expressions) are not valid, as happens with $let\ Y = coin\ in\ (Y, Y) \rightarrow^{rt} (coin, coin)$, 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 $(f(X) \rightarrow let\ Y = X\ in\ (Y, Y))$ with $(f(X) \rightarrow (X, X))$ is unsound, because they provide different levels of sharing: this could be easily detected in our setting because the step $let\ Y = X\ in\ (Y, Y) \rightarrow^{rt} (X, X)$ 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 to prevent 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. Besides 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 $\mathcal{P} = \{f(c(X)) \to true\}$ 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 $let\ Y = c(X)\ in\ f(Y) \to^{rt} let\ Z = X\ in\ let\ Y = c(Z)\ in\ f(Y)$; now the computation can proceed successfully by applying (RBind,Fapp,Elim) yielding $let\ Z = X\ in\ f(c(Z)) \to^{rt} let\ Z = X\ in\ true \to^{rt} true$. 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).

As an example of derivation, consider the program $\mathcal{P} = \{coin \to 0, coin \to s(0), 0{+}X \to X, s(X){+}Y \to s(X{+}Y), double(X) \to let\ Y = X\ in\ Y{+}Y, pos(s(X)) \to true\}$ defining some easy operations for natural numbers (represented with 0 and s in the standard way). Notice the *let*-binding in the function *double*; it allows for example to evaluate $double(coin)$ to 0 or $s(s(0))$, but not to $s(0)$ (that could be obtained with $\to^{rt}$ if the binding were not present). The following is a possible $\to^{rt}$-derivation with $\mathcal{P}$ for the expression $pos(double(double(coin)))$. At each step, the redex is underlined and the applied $\to^{rt}$-rule is indicated on the right:

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

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 Def. 1 except (Fapp) is terminating.*

The next result reflects the fact that all rules except (Fapp) are syntactic transformations that preserve the outer constructed part of the expressions. This is in fact a first partial soundness result about the relation $\rightarrow^{rt}$.

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

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 [15, 20]–.

The first statement – *rt-let*-rewriting generalizes ordinary rewriting – 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 Def. 1, because the absence of *let*s implies that $BV(\mathcal{C}) = \emptyset$ for any context $\mathcal{C}$, which guarantees the condition *i)* in Def. 1. Therefore, we have:

Theorem 2 (*Rt-let*-rewriting extends rewriting).
If $\mathcal{P}$ *is a program with no lets, then* $e \rightarrow_{\mathcal{P}} e' \Leftrightarrow e \rightarrow^{rt}_{\mathcal{P}} e'$, *for any* $e, e' \in Exp$.

To compare *rt-let*-rewriting with the *ct-let*-rewriting relation of [20] is more complicated since, despite their rough similarity, both relations are quite different; as a matter of fact, they are incomparable step by step. However, we will show how a program $\mathcal{P}$ can be transformed into another $\tau(\mathcal{P})$ that behaves, under *rt-let*-rewriting, as $\mathcal{P}$ with respect to *ct-let*-rewriting It is interesting to remark in advance that we will base the proof of adequacy of τ in semantic properties of $CRWL_{let}$, instead of reasoning directly about *ct-let*-rewriting derivations.

The transformation τ introduces *let*-bindings in the rules of a program in order to simulate call-time choice semantics, and is defined as follows:

Definition 2 (Sharing transformation τ). *Given a program rule* $R \equiv f(\overline{t}) \rightarrow e$, *its transformed rule is* $\tau(R) \equiv (f(\overline{t}) \rightarrow let\ \overline{Y = X}\ in\ e[\overline{X/Y}])$ *where* $FV(e) = \overline{X}$ *and* $\overline{Y}$ *is a linear tuple of fresh variables.*
The transformation is naturally extended to programs as $\tau(\mathcal{P}) = \{\tau(R) | R \in \mathcal{P}\}$.

This transformation introduces a *let*-binding for each variable in the right-hand side of a program rule. For example, for the program $\mathcal{P} = \{coin \rightarrow 0, coin \rightarrow 1, pair(X) \rightarrow (X, X)\}$ the last rule is transformed as $pair(X) \rightarrow let\ Y = X\ in\ (Y, Y)$, and if we evaluate now $pair(coin)$ we can obtain $(0, 0)$ and $(1, 1)$ but not $(0, 1)$ or $(1, 0)$; that reflects the evaluation of $pair(coin)$ with call-time choice.

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 terms, τ serves to simulate call-time choice within run-time choice. 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 3 (Adequacy of τ under $CRWL_{let}$). *For any program $\mathcal{P}$ 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 $e \rightarrow^{rt} e'$ 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 (One-step hyper-soundness of $\rightarrow^{rt}$ for $\tau(\mathcal{P})$). *For any program $\mathcal{P}$, $e, e' \in LExp$, if $e \rightarrow^{rt}_{\tau(\mathcal{P})} e'$ then $[\![e']\!]^{\tau(\mathcal{P})} \Subset [\![e]\!]^{\tau(\mathcal{P})}$.*

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

Theorem 4. *For any program $\mathcal{P}$, $e, e' \in LExp$, $t \in CTerm$:*

a) $e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} e'$ *implies* $[\![e']\!]^{\tau(\mathcal{P})} \subseteq [\![e]\!]^{\tau(\mathcal{P})}$
b) $e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} t$ *implies* $e \rightarrow^{ct\,*}_{\tau(\mathcal{P})} t$

Part *b)*, which follows from *a)* and the equivalence of $\rightarrow^{ct}$ and the $CRWL_{let}$ semantics (Th. 1), establishes already a close relationship between $\rightarrow^{rt}$ and $\rightarrow^{ct}$, but it is not yet our final soundness result, because it mentions only the transformed program $\tau(\mathcal{P})$. With the aid of Theorem 3, it is now straightforward to formulate our desired soundness result:

Theorem 5 (Soundness of τ as simulation of call-time choice). *For any program $\mathcal{P}$, $e \in LExp$, $t \in CTerm$ we have that $e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} t$ implies $e \rightarrow^{ct\,*}_{\mathcal{P}} t$.*

The next goal is proving *completeness* of the simulation, i.e., the reciprocal of Th. 5. 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 $\rightarrow^{rt}$ derivation starting from e.

Lemma 2 (Completeness lemma for $\rightarrow^{rt}$). *For any program $\mathcal{P}$, $e \in LExp, t \in CTerm_{\perp}$, if $\mathcal{P} \vdash e \twoheadrightarrow t$ then $e \rightarrow^{rt\,*}_{\mathcal{P}} e'$ 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 $t = |e'|$, 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 $\rightarrow^{rt}$-steps, and therefore that for t total $\mathcal{P} \vdash e \twoheadrightarrow t$ implies $e \rightarrow^{rt\,*}_{\mathcal{P}} t$. 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 $let\ Y = X\ in\ ...$ that cannot be dropped off because of the condition *i)* imposed to $\rightarrow^{rt}$ in Def. 1. What can be proved is the following:

Theorem 6 (Completeness of $\rightarrow^{rt}$ wrt $CRWL_{let}$).
For any program $\mathcal{P}$, $e \in LExp$, and $t \in CTerm$, if $\mathcal{P} \vdash e \twoheadrightarrow t$, then:
a) $e \rightarrow^{rt\,*}_{\tau(\mathcal{P})}$ $let\ \overline{Y = X}\ in\ t'$, for some $t' \in CTerm$ such that $t'[\overline{Y/X}] \equiv t$ and $\overline{X} \subseteq FV(t)$.
b) If in addition t is ground, then $e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} t$.

Joining all these completeness results with the previous soundness results, the equivalence of $\mathcal{P}$ and $\tau(\mathcal{P})$ wrt $CRWL_{let}$, and the equivalence of $CRWL_{let}$-semantics and *ct-let*-rewriting, 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 7 (Adequacy of the simulation of call-time-choice).
For any program $\mathcal{P}$, $e \in LExp$, $t \in CTerm_\bot$:

a) $\mathcal{P} \vdash e \twoheadrightarrow t \ \Leftrightarrow\ e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} e'$, for some $|e'| \sqsupseteq t$.

b) If t is total, then $\mathcal{P} \vdash e \twoheadrightarrow t \ \Leftrightarrow\ e \rightarrow^{ct^*}_{\mathcal{P}} t \ \Leftrightarrow\ e \rightarrow^{rt\,*}_{\tau(\mathcal{P})}$ $let\ \overline{Y = X}\ in\ t'$ 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 $\mathcal{P} \vdash e \twoheadrightarrow t \Leftrightarrow e \rightarrow^{ct^*}_{\mathcal{P}} t \Leftrightarrow e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} t$

5 Discussion: could it be done 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 [20] 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 remarkably simplicity, thanks to the equivalence of semantics and reduction for call-time choice ([20]) and the strength of some essential results about semantics.

Theorem 8. *There are programs $\mathcal{P}$ for which no program $\mathcal{P}'$ can verify the following property* (P)*: $e \rightarrow^{rt^*}_{\mathcal{P}} t \Leftrightarrow e \rightarrow^{ct^*}_{\mathcal{P}'} t$ for any ground $e \in Exp$, $t \in Cterm$.*

Proof. The following simple program suffices: $\mathcal{P} \equiv \{f(X) \rightarrow (X, X),\ coin \rightarrow 0,\ coin \rightarrow 1\}$. Assume there exists $\mathcal{P}'$ verifying *(P)*. Since $f(coin) \rightarrow^{rt^*}_{\mathcal{P}} (0, 1)$, *(P)* implies that $f(coin) \rightarrow^{ct^*}_{\mathcal{P}'} (0, 1)$ and then, because of the equivalence of *ct-let*-rewriting $\rightarrow^{ct^*}$ and *CRWL*-derivability (Th. 1), we have $\mathcal{P}' \vdash f(coin) \twoheadrightarrow (0, 1)$. By a compositionality property of call-time choice (see e.g. [22], Th. 1), there must be $t \in CTerm_\bot$ such that $\mathcal{P}' \vdash coin \twoheadrightarrow t$ and $\mathcal{P}' \vdash f(t) \twoheadrightarrow (0, 1)$. 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 ([15]) proves that $\mathcal{P}' \vdash f(s) \twoheadrightarrow (0,1)$ for any $s \in CTerm_\bot$, in particular $\mathcal{P}' \vdash f(0) \twoheadrightarrow (0,1)$, and therefore $f(0) \rightarrow^{ct^*}_{\mathcal{P}'} (0,1)$, by Th. 1. Then, again by *(P)*, $f(0) \rightarrow^{rt^*}_{\mathcal{P}} (0,1)$, but this is not true.

(b) If $t \equiv 0$, then $\mathcal{P}' \vdash f(t) \twoheadrightarrow (0,1)$ 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 $\rightarrow^{rt}$ could by replaced by ordinary rewriting $\rightarrow$ (Th. 2) along Th. 8 and its proof. Second, the groundness restriction for e, t in the theorem is not a weakness, but quite the opposite (since the proposition as it is 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 [12], since the properties of *CRWL*-semantics used in the proof are also true for such HO extension.

Theorem 8 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 [17] or Toy [23]). At a first sight, an old well-known HO technique [1] for delaying evaluation, based on the fact that partial applications are not evaluated, could help. 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 a 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 of it):

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 [7, 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 *all* functions f need to be artificially changed with dummy arguments, and 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(['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's 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.
(iii) It requires to use HO to express FO run-time, thus mixing unnecessarily two concerns. Moreover, it is known (see e.g. [22]) 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 [12], adopting a FO translation [13, 4], or both [22]. 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 within call-time (and the question might be not trivial, as argued before).

6 Conclusions

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 priori about the semantics (run-time choice/call-time choice) of non-determinism adopted for functions, our approach allows using different semantics within the same program, which reveals itself as a very useful resource in many cases.This is achieved with great flexibility, because the selection of semantics can be done at the level of individual arguments or subexpressions, not only at the level of the complete definition of a function.

Our approach in a nutshell could be described as follows: to combine run-time choice and call-time choice, add a *let*-construct to a run-time choice framework (e.g., ordinary rewriting), and impose appropriate laws to the propagation of bindings contained in *let*s. Pure run-time choice (call-time choice resp.) is then achieved by not using *let*s at all (introducing *let*s for all function defining rules, resp.). Being the ideas so simple, two false impressions might arise: that existing frameworks are sufficient to cope with 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 [21]), in a new strong evidence of the power, argued in [22], of using semantics to prove results about functional logic reductions.

The syntax presented here for our framework can be thought as a core syntax, that could be put in practice in different (mixable) ways:

(i) As syntactic sugar, each function can be declared as run-time or call-time, and its definition must be interpreted (and transformed, in the case of call-time) accordingly. A default declaration (call-time, most probably) could be assumed.

(ii) We can program using the core syntax, that is, with explicit *let*s. This gives a finer control, since we can choose specific behavior (shared/non shared) to each piece in an expression.

(iii) In a HO setting, the introduction of *let*s for call-time choice can be hidden in the function $call\_time\ F\ X \to let\ Y = X\ in\ F\ Y$ introduced in Sect. 5. .

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*, programmed to follow call-time choice: $repeat(X) \to let\ Y{=}X\ in\ [Y|repeat(Y)]$.
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 sharings are useless. This suggests (automatically) replacing the original definition of *repeat* by an optimized variant $repeat(X) \to let\ Y{=}X\ in\ [Y|repeat'(Y)]$, where the auxiliary *repeat'* is defined as $repeat'(X) \to [X|repeat'(X)]$, 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 [3] 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:

- The notion of rewriting given here should be lifted to a notion of narrowing, as was done in [19, 22] for the case of call-time choice.

- We must build an implementation of our framework. It should include types and HO functions, but we do not expect important novelties in this extension with respect to similar tasks performed in previous frameworks.

- We must invest some effort in producing a 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. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.
3. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into prolog. In *Proc. of the 3rd International Workshop on Frontiers of Combining Systems (FroCoS 2000)*, pages 171–185, Nancy, France, March 2000. Springer LNCS 1794.
4. S. Antoy and A. P. Tolmach. Typed higher-order narrowing without higher-order strategies. In *Fuji International Symposium on Functional and Logic Programming*, pages 335–353, 1999.
5. Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3):265–301, 1997.
6. Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. In *POPL*, pages 233–246, 1995.
7. R. Caballero-Roldán and F. López-Fraguas. Parsing with non-deterministic functions. In *Proc. Joint Conference on Declarative Programming, APPIA-GULP-PRODE'98*, pages 1–16, 1998.
8. R. Caballero-Roldán and F. López-Fraguas. A functional-logic perspective on parsing. In *FLOPS '99: Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, pages 85–99, London, UK, 1999. Springer-Verlag.
9. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The maude 2.0 system. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, pages 76–87. Springer LNCS 2706, 2003.
10. J. Dios-Castro and F. López-Fraguas. Extra variables can be eliminated from functional logic programs. *Electronic Notes in Theoretical Computer Science 188*, pages 3–19, 2007.
11. R. Echahed and J.-C. Janodet. On constructor-based graph rewriting systems. Research Report 985-I, IMAG, 1997.
12. J. González-Moreno, M. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. In *Proc. International Conference on Logic Programming (ICLP'97)*, pages 153–167. MIT Press, 1997.
13. J. C. González-Moreno. A correctness proof for Warren's ho into fo translation. In *GULP*, pages 569–584, 1993.
14. J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. European Symposium on Programming (ESOP'96)*, pages 156–172. Springer LNCS 1058, 1996.
15. 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.
16. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
17. 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.
18. H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.
19. 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*, pages 224–233, 2007.

20. 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*, pages 197–208. ACM Press, 2007.
21. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A flexible framework for programming with non-deterministic functions. http://gpd.sip.ucm.es/fraguas/papers/iclp08long.pdf, 2008.
22. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In *Proc. 9th International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *LNCS*, pages 147–162. Springer, 2008.
23. F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
24. D. Plump. Essentials of term graph rewriting. *Electr. Notes Theor. Comput. Sci.*, 51, 2001.
25. J. Rodríguez-Hortalá. El indeterminismo en programacin lógico-funcional: un enfoque basado en reescritura. Trabajo de Investigación de Tercer Ciclo, Dpto. de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Jun. 2007.
26. H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, 1992.

A Proofs of the results

Figures 1 and 2 show the $CRWL_{let}$ calculus and *ct-let*-rewriting relation, respectively, defined in [20]. In Figure 3 we show an extended definition of the rule of the run-time let rewriting relation $\rightarrow^{rt}$ in which some conditions has been made explicit. This formulation is equivalent to the one in Def. 1 and is used intensively in the proofs.

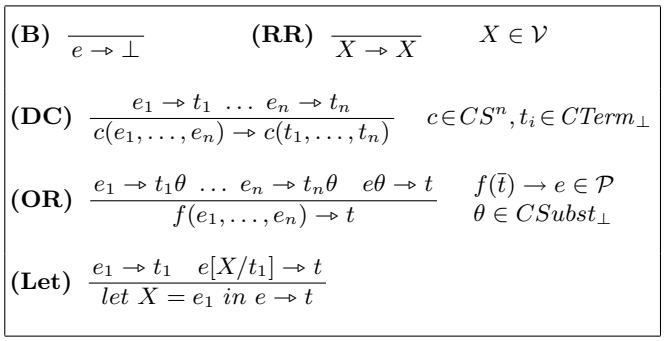

Fig. 1. Rules of $CRWL_{let}$

For the derivations over $\rightarrow^{rt}$ sometimes we will do a derivation assuming that the rule was applied at the top of the expression, thus making a case distinction over the rule of $\rightarrow^{rt\,\prime}$, and then we will see how this result can be propagated to a rewriting step in any subexpression. We will call this latter step (Contx), which is just an application of $\mathcal{C}[e] \rightarrow^{rt} \mathcal{C}[e']$ if $e \rightarrow^{rt\,\prime} e'$, while the former cases are applications of $\mathcal{C}[e] \rightarrow^{rt} \mathcal{C}[e']$ if $e \rightarrow^{rt\,\prime} e'$ for $\mathcal{C} = []$. We will also use $\rightarrow^{rt}$ instead of $\rightarrow^{rt\,\prime}$ by an

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

Fig. 2. Rules of *ct-let*-rewriting

abuse of notation.

The following technical lemmas will be useful:

Lemma 3. *For any $\mathcal{C} \in Contx, e \in LExp_\perp$:*

i) $|\mathcal{C}[e]| \equiv |\mathcal{C}[|e|]|$
ii) $|e_1[X/e_2]| \equiv |e_1|[X/|e_2|]$

Proof (For lemma 3).

i) By the definition of shells.
ii) See [25].

Lemma 4. *For any $e \in Exp_\perp$, $t \in CTerm_\perp$ and program $\mathcal{P}$, if $\mathcal{P} \vdash e \twoheadrightarrow t$ then there is a derivation for $\mathcal{P} \vdash e \twoheadrightarrow t$ in which every free variable used belongs to $FV(e \twoheadrightarrow t)$.*

Proof (For lemma 4). A simple extension of the proof in [10].

Lemma 5. *For every $CRWL_{let}$ derivation $e \twoheadrightarrow t$ there exists $e' \in LExp_\perp$ which is syntactically equivalent to e module α-conversion, and a $CRWL_{let}$ derivation for $e' \twoheadrightarrow t$ such that if $\mathcal{B}$ is the set of bound variables used in $e' \twoheadrightarrow t$ and $\mathcal{E}$ is the set of free variables used in the instantiation of extra variables in $e' \twoheadrightarrow t$ then $\mathcal{B} \cap (\mathcal{E} \cup var(t)) = \emptyset$.*

Proof (For lemma 5). By lemma 4, if $\mathcal{F}$ is the set of free variables used in $e' \twoheadrightarrow t$, then $\mathcal{F} \subseteq FV(e' \twoheadrightarrow t)$, in fact $\mathcal{F} = FV(e' \twoheadrightarrow t)$, as $FV(e')$ and $FV(t)$ are used in the top derivation of the derivation tree for $e' \twoheadrightarrow t$. As by definition $\mathcal{E} \cup var(t) \subseteq \mathcal{F}$, if we prove $\mathcal{B} \cap \mathcal{F} = \emptyset$ then $\mathcal{B} \cap (\mathcal{E} \cup var(t)) = \emptyset$ is a trivial consequence. To prove that we will prove that for every $a \in LExp_\perp$ used in the derivation for $e' \twoheadrightarrow t$ we have $BV(a) \cap FV(a) = \emptyset$. We can build e' using α-conversion to ensure that $BV(e') \cap FV(e') = \emptyset$. This can be easily maintained as an invariant during the derivation, as the new *let* bindings that appear during the derivation are those introduced in the instances of the rule used during the **OR** steps, and be can ensure by α-conversion that $BV(a) \cap FV(a) = \emptyset$ for these instances too, as α-conversion leaves the hypersemantics untouched.

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

(Fapp) $f(\bar{t})\sigma \rightarrow^{rt\,\prime} r\sigma$, if $(f(\bar{t}) \rightarrow r) \in \mathcal{P}$, $\sigma \in LSubst$
(RBind) $let\ X = t\ in\ e \rightarrow^{rt\,\prime} e[X/t]$, if $t \in CTerm$
(Elim) $let\ X = e_1\ in\ e_2 \rightarrow^{rt\,\prime} e_2$, if $X \notin FV(e_2)$
($\mathbf{Flat_1}$) $h(\ldots, let\ X = e_1\ in\ e_2, \ldots) \rightarrow^{rt\,\prime} let\ X = e_1\ in\ h(\ldots, e_2, \ldots)$, with $h \in \Sigma$, if $X \notin FV(h(\ldots, [], \ldots))$
($\mathbf{Flat_2}$) $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \rightarrow^{rt\,\prime} let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$, if $Y \notin FV(e_3)$
($\mathbf{LetIn_1}$) $let\ X = \mathcal{C}[f(\bar{e})]\ in\ e \rightarrow^{rt\,\prime} let\ Y = f(\bar{e})\ in\ let\ X = \mathcal{C}[Y]\ in\ e$, with $f \in FS$, $Y \in \mathcal{V}$ fresh and $\mathcal{C} \neq []$ a c-context
($\mathbf{LetIn_2}$) $let\ X = \mathcal{C}[Y]\ in\ e \rightarrow^{rt\,\prime} let\ Z = Y\ in\ let\ X = \mathcal{C}[Z]\ in\ e$, with $Y, Z \in \mathcal{V}$, Z fresh and $\mathcal{C} \neq []$ a c-context

Now, for any $\mathcal{C} \in Cntx$ we have $\mathcal{C}[e] \rightarrow^{rt} \mathcal{C}[e']$, if $e \rightarrow^{rt\,\prime} e'$ using any of the previous rules, and in case $e \rightarrow^{rt\,\prime} e'$ is of the form:

i) $f(\bar{t})\sigma \rightarrow^{rt\,\prime} r\sigma$ 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) $let\ X = t\ in\ a \rightarrow^{rt\,\prime} a[X/t]$ by (RBind), then $var(t) \subseteq BV(\mathcal{C})$.
iii) $let\ X = \mathcal{C}'[Y]\ in\ a \rightarrow^{rt\,\prime} let\ Z = Y\ in\ let\ X = \mathcal{C}'[Z]\ in\ a$ by ($\mathrm{LetIn_2}$), then $Y \notin BV(\mathcal{C})$.

Fig. 3. Run-time let rewriting relation

Free and *bound* variables of $e \in LExp$ are defined as:

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

We remark that the given definition of FV implies that recursive *let*-bindings simply do not exist. For instance, the binding in the expression $let\ X = s(X)\ in\ f(X)$ is not seen as recursive, despite of its aspect, because the occurrence of X in $s(X)$ is a free occurrence, and so it is 'different' from the bound X. Renaming the bound X to Y in the expression would give $let\ Y = s(X)\ in\ f(Y)$.

The following lemmas related to the sharing transformation $\tau()$ defined in Def. 2 will be useful later on. The first one states that $\tau()$ preserves free variables.

Lemma 6. *For any program rule $(l \rightarrow r)$ we have $FV(l \rightarrow r) = FV(\tau(l \rightarrow r))$, where $FV(f(\overline{p}) \rightarrow r)$ is defined as $var(\overline{p}) \cup FV(r)$.*

Proof (For lemma 6).

$$\begin{aligned}
&FV(\tau(l \rightarrow r)) = FV(l) \cup FV(let\ \overline{Y = X}\ in\ r[\overline{X/Y}]) = && \text{Def. of } \tau \\
&= FV(l) \cup \overline{X} \cup (FV(r[\overline{X/Y}]) \setminus \overline{Y}) && \\
&= FV(l) \cup \overline{X} \cup (\overline{Y} \setminus \overline{Y}) && \text{as } FV(r) = \overline{X} \\
&= FV(l) \cup \overline{X} \cup \emptyset = FV(l) \cup FV(r) = FV(l \rightarrow r) &&
\end{aligned}$$

The following result shows how we can introduce arbitrary *let*-bindings in expressions (for instance, those introduced by τ) while preserving their $CRWL_{let}$ semantics, and moreover their hypersemantics.

Lemma 7. *Let $\mathcal{P}$ be any program, $e \in LExp_\bot$ and $\overline{X}$ a linear n-tuple of arbitrary variables. Then*

$$[\![e]\!]^{\mathcal{P}} = [\![let\ \overline{Y = X}\ e[\overline{X/Y}]]\!]^{\mathcal{P}}$$

where $\overline{Y}$ is a linear n-tuple of fresh variables.

As a direct consequence, for any program $\mathcal{P}$ and $e \in LExp_\bot$ $[\![e]\!]^{\mathcal{P}} = [\![\tau(e)]\!]^{\mathcal{P}}$.

Proof (For lemma 7). Notice that as $\overline{X}$ and $\overline{Y}$ are linear tuples, the substitutions $[X_1/Y_1], \ldots [X_n/Y_n]$ can be reordered in any way obtaining the same substitution $[\overline{X/Y}]$. The notation $\overline{[X_{1..i}/Y_{1..i}]}$ stands for the substitution $[X_1/Y_1, \ldots, X_i/Y_i]$. The proof of the lemma proceed by induction on the number of variables in $\overline{X}$. The base case is trivial because there is no let and the induction step is $(i \Rightarrow i+1)$:

$$\begin{array}{l}
[\![let\ \overline{Y_{1..i+1} = X_{1..i+1}}\ in\ e[\overline{X_{1..i+1}/Y_{1..i+1}}]]\!] =_{(1)} \\
[\![let\ \overline{Y_{1..i} = X_{1..i}}\ in\ e[\overline{X_{1..i+1}/Y_{1..i+1}}][Y_{i+1}/X_{i+1}]]\!] =_{(2)} \\
[\![let\ \overline{Y_{1..i} = X_{1..i}}\ in\ e[\overline{X_{1..i}/Y_{1..i}}]]\!] =_{IH} \\
[\![e]\!]
\end{array}$$

The step (1) is justified because it is a step with **(Bind)**, that preserves the hyper-semantics of the expression (see [20]); and the step (2) is also sound because $e[\overline{X_{1..i+1}/Y_{1..i+1}}][Y_{i+1}/X_{i+1}] = e[\overline{X_{1..i}/Y_{1..i}}]$ as Y_{i+1} is fresh and does not appear in e.

In this lemma we see how applying $\tau()$ to a single program rule does not change the denotation of expressions, the key to prove Theorem 3.

Lemma 8. *Let $\mathcal{P}$ be a program, $(l \to r) \in \mathcal{P}$ and $\mathcal{P}' = (\mathcal{P} \setminus \{l \to r\}) \cup \{l \to \tau(r)\}$, then for any $e \in LExp_\bot$ we have $[\![e]\!]^{\mathcal{P}} = [\![e]\!]^{\mathcal{P}'}$.*

Proof (For lemma 8). We must prove that for any $\sigma \in CSubst_\bot$ and $e \in LExp_\bot$:

$$\mathcal{P} \vdash e\sigma \twoheadrightarrow t \Leftrightarrow \mathcal{P}' \vdash e\sigma \twoheadrightarrow t$$

We proceed by induction on the size k of the derivation for $\mathcal{P} \vdash e\sigma \twoheadrightarrow t$:

- $k = 0$: the derivations with respect to $\mathcal{P}$ or $\mathcal{P}'$ are the same as they do not use any rule of the program.
- $k \Rightarrow k+1$: For proving the $(\Rightarrow)$ part, if the derivation $\mathcal{P} \vdash e\sigma \twoheadrightarrow t$ starts with a (DC) or (Let) step, the proof is a direct application of I.H., and similarly for the $(\Leftarrow)$ part. The most interesting case is when the derivation starts with a (OR) step using the rule $(l \to r) \equiv (f(\bar{t}) \to r)$. Then $e\sigma$ must be of the form $f(e_1, \ldots, e_n)$ and the derivation with respect to $\mathcal{P}$ is:

$$\frac{e_1 \twoheadrightarrow t_1\theta\ \ \ldots\ \ e_n \twoheadrightarrow t_n\theta \quad r\theta \twoheadrightarrow t}{f(e_1, \ldots, e_n) \twoheadrightarrow t}\ (OR)$$

using $\theta \in CSubst_\bot$ with $dom(\theta) = FV(f(\bar{t}) \to r)$. By I.H. we have all the derivations $\mathcal{P}' \vdash e_i \twoheadrightarrow t_i\theta$ and we must search for a derivation for $\mathcal{P}' \vdash \tau(r)\theta \twoheadrightarrow t$. For the last we have $\tau(r)\theta \equiv (let\ \overline{Y = X}\ in\ r[\overline{X/Y}])\theta$, where $\overline{X} = FV(r)$ and $\overline{Y}$ are fresh variables. Applying θ, this expression is equivalent to $let\ \overline{Y = X\theta}\ in\ r[\overline{X/Y}]\theta$, and as θ does not affect the variables $\overline{Y}$, this is also $let\ \overline{Y = X\theta}\ in\ r[\overline{X/Y}]$. For

this expression we can perform a derivation applying the rule (Let) once for each binding $Y_i = X_i$ reaching a derivation for $r[\overline{X/Y}][\overline{Y/X\theta}] \equiv r\theta \to t$, which can be done by I.H.
For ($\Leftarrow$) in the case of (OR) (using the program rule $f(\bar{t}) \to \tau(r)$), if $(let\ \overline{Y = X\theta}\ in\ r[\overline{X/Y}]) \twoheadrightarrow t$ then $X_i\theta \twoheadrightarrow s_i$ and $r[\overline{X/Y}][\overline{Y/s}] \twoheadrightarrow t$ ($\overline{X}$ and $\overline{Y}$ are linear by hypothesis), for some tuple of c-terms $\overline{s}$. But $X_i\theta \in CTerm_\perp$, and then $X_i\theta \twoheadrightarrow s_i$ implies $s_i \sqsubseteq X_i\theta$ and in fact $[\overline{X/s}] \sqsubseteq \theta$. Then $r[\overline{X/s}] = r[\overline{X/Y}][\overline{Y/s}] \twoheadrightarrow t$ (as $FV(r) = \overline{X}$) implies that the derivation $r\theta \twoheadrightarrow t$ can be done with smaller size.

Now we are ready to prove Theorem 3.

Proof (For Theorem 3). Given $\mathcal{P} = \{\rho_1, \ldots, \rho_n\}$ and $e \in Exp_\perp$ then

$$[\![e]\!]^{\mathcal{P}} = [\![e]\!]^{\{\rho_1,\ldots,\rho_n\}} = [\![e]\!]^{\{\tau(\rho_1),\ldots,\rho_n\}} = \ldots = [\![e]\!]^{\{\tau(\rho_1),\ldots,\tau(\rho_n)\}}$$

applying lemma 8 n times.

Now we will focus on proving lemma 1, to do this we firstly need the following technical results:

Lemma 9. *Let $t \in CTerm$ linear and $\sigma \in LSubst_\perp$ such that for any $X_i \in \overline{X} = var(t)$ we have $\mathcal{P} \vdash X_i\sigma \twoheadrightarrow s_i$ for some $s_i \in CTerm_\perp$. Then $\mathcal{P} \vdash t\sigma \twoheadrightarrow t[\overline{X_i/s_i}]$.*

Proof (For lemma 9). By induction on the structure of t:

Base cases
- $t \equiv X$: Then $\overline{X} = \{X\}$ and $\mathcal{P} \vdash_{CRWL_{let}} X\sigma \twoheadrightarrow s$, so $t\sigma \equiv X\sigma \twoheadrightarrow s \equiv X[X/s] \equiv t[\overline{X_i/s_i}]$
- $t \equiv c \in CS^0$: Then $\overline{X} = \emptyset$ and $t\sigma \equiv c\sigma \equiv c \twoheadrightarrow c \equiv c\epsilon \equiv t\epsilon \equiv t[\overline{X_i/s_i}]$

Inductive step $t \equiv c(t_1, \ldots, t_n)$: As t is linear then we assume $\overline{X} = \overline{X_1} \uplus \ldots \uplus \overline{X_n}$, where $\overline{X_j} = var(t_j)$. Now we can build:

$$\frac{\dfrac{IH}{t_1\sigma \twoheadrightarrow t_1[\overline{X_1/s_1}]} \quad \ldots \quad \dfrac{IH}{t_n\sigma \twoheadrightarrow t_n[\overline{X_n/s_n}]}}{t\sigma \equiv c(t_1\sigma, \ldots, t_n\sigma) \twoheadrightarrow c(t_1[\overline{X_1/s_1}], \ldots, t_n[\overline{X_n/s_n}])}\ DC$$

Note how, by the linearity of t, the premises corresponding to each t_j are independent and so the induction hypothesis can be applied independently too. Besides $c(t_1[\overline{X_1/s_1}], \ldots, t_n[\overline{X_n/s_n}]) \equiv c(t_1, \ldots, t_n)\ [\overline{X_1/s_1}, \ldots, \overline{X_n/s_n}]$ for the same reason.

Lemma 10 (Weak compositionality of $CRWL_{let}$). *For any $\mathcal{P}$ and $e \in LExp_\perp$: $[\![\mathcal{C}[e]]\!] = \bigcup_{t \in [\![e]\!]} [\![\mathcal{C}[t]]\!]$, if $BV(\mathcal{C}) \cap FV(e) = \emptyset$. In particular, $[\![let\ X = e_1\ in\ e_2]\!] = \bigcup_{t \in [\![e_1]\!]} [\![e_2[X/t]]\!]$*

Proof (For lemma 10). It follows the same schema of weak compositionality of [22].

The notion of *hypersemantics of a context* and its associated compositionality result are powerful proving tools that we will use to prove lemma 1.

Definition 3 (Hypersemantics of a context). *Given $\mathcal{C} \in Contx$ its hypersemantics $[\![\mathcal{C}]\!]$ is a transformer of hypersemantics of expressions. Given $\varphi : CSubst_\perp \to \mathcal{P}(CTerm_\perp)$ and $\theta \in CSubst_\perp$, $[\![\mathcal{C}]\!]$ is defined by induction over the structure of $\mathcal{C}$:*

- $[\![[]]\!]\varphi = \varphi$
- $[\![h(e_1, \ldots, \mathcal{C}, \ldots, e_n)]\!]\varphi\theta = \bigcup_{t \in [\![\mathcal{C}]\!]\varphi\theta} [\![h(e_1\theta, \ldots, t, \ldots, e_n\theta)]\!]$
- $[\![let\ X = \mathcal{C}\ in\ e]\!]\varphi\theta = \bigcup_{t \in [\![\mathcal{C}]\!]\varphi\theta} [\![let\ X = t\ in\ e\theta]\!]$
- $[\![let\ X = e\ in\ \mathcal{C}]\!]\varphi\theta = \bigcup_{t \in [\![e]\!]\theta} [\![\mathcal{C}]\!]\varphi(\theta[X/t])$

With this notion we can prove the following abstract and powerful compositionaly result for hypersemantics, generalizing (and simplifying the aspect of) lemma 10, which was formulated in terms of semantics.

Lemma 11 (Compositionality of hypersemantics). $[\![\mathcal{C}[e]]\!] = [\![\mathcal{C}]\!][\![e]\!]$

This result implies that in any context one can replace any subexpression by another one having the same hypersemantics (and therefore also the same semantics) without changing the hypersemantics (hence the semantics) of the global expression.

Proof (For lemma 11). By induction over the structure of contexts.

Base case $\mathcal{C} = []$: Then $[\![\mathcal{C}[e]]\!] = [\![e]\!] = [\![[]]\!][\![e]\!] = [\![\mathcal{C}]\!][\![e]\!]$, as $[\![[]]\!]$ is the identity function, by definition.

Inductive step

- $\mathcal{C} = h(e_1, \ldots, \mathcal{C}', \ldots, e_n)$: Then

$$
\begin{aligned}
&[\![\mathcal{C}]\!][\![e]\!] = \lambda\theta. \bigcup_{t \in [\![\mathcal{C}']\!][\![e]\!]\theta} [\![h(e_1\theta, \ldots, t, \ldots, e_n\theta)]\!] \\
&= \lambda\theta. \bigcup_{t \in [\![\mathcal{C}'[e]]\!]\theta} [\![h(e_1\theta, \ldots, t, \ldots, e_n\theta)]\!] && \text{by IH} \\
&= \lambda\theta. \bigcup_{t \in [\![(\mathcal{C}'[e])\theta]\!]} [\![h(e_1\theta, \ldots, t, \ldots, e_n\theta)]\!] && \text{by definition} \\
&= \lambda\theta.[\![h(e_1\theta, \ldots, (\mathcal{C}'[e])\theta, \ldots, e_n\theta)]\!] && \text{by lemma 10} \\
&= \lambda\theta.[\![(\mathcal{C}[e])\theta]\!] = [\![\mathcal{C}[e]]\!]
\end{aligned}
$$

- $\mathcal{C} = let\ X = \mathcal{C}'\ in\ s$: Then

$$
\begin{aligned}
&[\![\mathcal{C}]\!][\![e]\!] = \lambda\theta. \bigcup_{t \in [\![\mathcal{C}']\!][\![e]\!]\theta} [\![let\ X = t\ in\ s\theta]\!] && \text{by definition} \\
&= \lambda\theta. \bigcup_{t \in [\![\mathcal{C}']\!][\![e]\!]\theta} [\![s\theta[X/t]]\!] && \text{by rule (Bind) of } \to^{ct} \\
&= \lambda\theta. \bigcup_{t \in [\![\mathcal{C}'[e]]\!]\theta} [\![s\theta[X/t]]\!] && \text{by IH} \\
&= \lambda\theta. \bigcup_{t \in [\![(\mathcal{C}'[e])\theta]\!]} [\![s\theta[X/t]]\!] && \text{by definition} \\
&= \lambda\theta.[\![let\ X = (\mathcal{C}'[e])\theta\ in\ s\theta]\!] && \text{by lemma 10} \\
&= [\![\mathcal{C}[e]]\!]
\end{aligned}
$$

– $\mathcal{C} = let\ X = s\ in\ \mathcal{C}'$: Then

$$\begin{aligned}
&[\![\mathcal{C}]\!][\![e]\!] = \lambda\theta.\ \bigcup_{t\in[\![s]\!]\theta} [\![\mathcal{C}']\!][\![e]\!](\theta[X/t]) \\
&= \lambda\theta.\ \bigcup_{t\in[\![s]\!]\theta} [\![\mathcal{C}'[e]]\!](\theta[X/t]) && \text{by IH} \\
&= \lambda\theta.\ \bigcup_{t\in[\![s]\!]\theta} [\![(\mathcal{C}'[e])(\theta[X/t])]\!] && \text{by definition} \\
&= \lambda\theta.\ \bigcup_{t\in[\![s\theta]\!]} [\![(\mathcal{C}'[e])(\theta[X/t])]\!] && \text{by definition} \\
&= \lambda\theta.\ \bigcup_{t\in[\![s\theta]\!]} [\![((\mathcal{C}'[e])\theta)[X/t]]\!] \\
&= \lambda\theta.[\![let\ X = s\theta\ in\ (\mathcal{C}'[e])\theta]\!] && \text{by lemma 10} \\
&= [\![\mathcal{C}[e]]\!]
\end{aligned}$$

The following lemma, combined with lemma 11, will be one of the keys to prove lemma 1.

Lemma 12. *If $BV(\mathcal{C}) \cap FV(e_1) = \emptyset$ and $X \notin FV(\mathcal{C})$ then $[\![\mathcal{C}[let\ X = e_1\ in\ []]]\!] = [\![let\ X = e_1\ in\ \mathcal{C}]\!]$*

Proof (For lemma 12). By induction on the structure of $\mathcal{C}$:

Base case $\mathcal{C} = []$: This case is trivial as then $\mathcal{C}[let\ X = e_1\ in\ []] \equiv let\ X = e_1\ in\ \mathcal{C}$
Inductive steps

• $\mathcal{C} = h(a_1, \ldots, \mathcal{C}', \ldots, a_n)$: Then

$$\begin{aligned}
&[\![\mathcal{C}[let\ X = e_1\ in\ []]]\!]\varphi\theta \\
&= [\![h(a_1, \ldots, \mathcal{C}'[let\ X = e_1\ in\ []], \ldots, a_n)]\!]\varphi\theta \\
&= \bigcup_{t\in[\![\mathcal{C}'[let\ X=e_1\ in\ []]]\!]\varphi\theta} [\![h(a_1\theta, \ldots, t, \ldots, a_n\theta)]\!] \\
&=_{IH} \bigcup_{t\in[\![let\ X=e_1\ in\ \mathcal{C}']\!]\varphi\theta} [\![h(a_1\theta, \ldots, t, \ldots, a_n\theta)]\!] \\
&= \bigcup_{t\in(\bigcup_{s\in[\![e_1]\!]\theta} [\![\mathcal{C}']\!]\varphi(\theta[X/s]))} [\![h(a_1\theta, \ldots, t, \ldots, a_n\theta)]\!] \\
&= \bigcup_{s\in[\![e_1]\!]\theta}\ \bigcup_{t\in[\![\mathcal{C}']\!]\varphi(\theta[X/s])} [\![h(a_1\theta, \ldots, t, \ldots, a_n\theta)]\!] \\
&=_{(1)} \bigcup_{s\in[\![e_1]\!]\theta}\ \bigcup_{t\in[\![\mathcal{C}']\!]\varphi(\theta[X/s])} [\![h(a_1\theta[X/s], \ldots, t, \ldots, a_n\theta[X/s])]\!] \\
&= \bigcup_{s\in[\![e_1]\!]\theta} ([\![h(a_1, \ldots, \mathcal{C}', \ldots, a_n)]\!]\varphi(\theta[X/s])) \\
&= [\![let\ X = e_1\ in\ h(a_1, \ldots, \mathcal{C}', \ldots, a_n)]\!]\varphi\theta \\
&= [\![let\ X = e_1\ in\ \mathcal{C}]\!]\varphi\theta
\end{aligned}$$

where:

- (1) : As $FV(a_1) \cup \ldots \cup FV(a_n) \subseteq FV(h(a_1, \ldots, \mathcal{C}', \ldots, a_n)) = FV(\mathcal{C})$ and $X \notin FV(\mathcal{C})$ by hypothesis, then $\forall i,\ X \notin FV(a_i)$. Besides $X \notin vran(\theta)$ by the variable convention, thus $X \notin FV(a_i\theta)$ and $a_i\theta[X/s] \equiv a_i\theta$ for any i.

• $\mathcal{C} = let\ Y = \mathcal{C}'\ in\ s$: Then

$$
\begin{aligned}
&[\![\mathcal{C}[let\ X = e_1\ in\ []]]\!]\varphi\theta \\
&= [\![let\ Y = \mathcal{C}'[let\ X = e_1\ in\ []]\ in\ s]\!]\varphi\theta \\
&= \bigcup_{t \in [\![\mathcal{C}'[let\ X = e_1\ in\ []]]\!]\varphi\theta} [\![let\ Y = t\ in\ s\theta]\!] \\
&=_{IH} \bigcup_{t \in [\![let\ X = e_1\ in\ \mathcal{C}']\!]\varphi\theta} [\![let\ Y = t\ in\ s\theta]\!] \\
&= \bigcup_{t \in (\bigcup_{r \in [\![e_1]\!]\theta} [\![\mathcal{C}']\!]\varphi(\theta[X/r]))} [\![let\ Y = t\ in\ s\theta]\!] \\
&= \bigcup_{r \in [\![e_1]\!]\theta} \bigcup_{t \in [\![\mathcal{C}']\!]\varphi(\theta[X/r])} [\![let\ Y = t\ in\ s\theta]\!] \\
&=_{(1)} \bigcup_{r \in [\![e_1]\!]\theta} \bigcup_{t \in [\![\mathcal{C}']\!]\varphi(\theta[X/r])} [\![let\ Y = t\ in\ s\theta[X/r]]\!] \\
&= \bigcup_{r \in [\![e_1]\!]\theta} ([\![let\ Y = \mathcal{C}'\ in\ s]\!]\varphi(\theta[X/r])) \\
&= [\![let\ X = e_1\ in\ let\ Y = \mathcal{C}'\ in\ s]\!]\varphi\theta \\
&= [\![let\ X = e_1\ in\ \mathcal{C}]\!]\varphi\theta
\end{aligned}
$$

where:

- (1) : As $FV(s) \subseteq FV(\mathcal{C}) \cup \{Y\}$, and because we may assume $X \neq Y$ by α-conversion, and we also have $X \notin FV(\mathcal{C})$ by hypothesis, then $X \notin FV(s)$. Besides $X \notin vran(\theta)$ by the variable convention, thus $X \notin FV(s\theta)$ and $s\theta[X/r] \equiv s\theta$.

• $\mathcal{C} = let\ Y = s\ in\ \mathcal{C}'$: Then

$$
\begin{aligned}
&[\![\mathcal{C}[let\ X = e_1\ in\ []]]\!]\varphi\theta \\
&= [\![let\ Y = s\ in\ \mathcal{C}'[let\ X = e_1\ in\ []]]\!]\varphi\theta \\
&= \bigcup_{t \in [\![s]\!]\theta} [\![\mathcal{C}'[let\ X = e_1\ in\ []]]\!]\varphi(\theta[Y/t]) \\
&=_{IH} \bigcup_{t \in [\![s]\!]\theta} [\![let\ X = e_1\ in\ \mathcal{C}']\!]\varphi(\theta[Y/t]) \\
&= \bigcup_{t \in [\![s]\!]\theta} \bigcup_{r \in [\![e_1]\!](\theta[Y/t])} [\![\mathcal{C}']\!]\varphi(\theta[Y/t][X/r]) \\
&=_{(1)} \bigcup_{t \in [\![s]\!]\theta} \bigcup_{r \in [\![e_1]\!]\theta} [\![\mathcal{C}']\!]\varphi(\theta[Y/t][X/r]) \\
&= \bigcup_{r \in [\![e_1]\!]\theta} \bigcup_{t \in [\![s]\!]\theta} [\![\mathcal{C}']\!]\varphi(\theta[Y/t][X/r]) \\
&=_{(2)} \bigcup_{r \in [\![e_1]\!]\theta} \bigcup_{t \in [\![s]\!]\theta} [\![\mathcal{C}']\!]\varphi(\theta[X/r][Y/t]) \\
&=_{(3)} \bigcup_{r \in [\![e_1]\!]\theta} \bigcup_{t \in [\![s]\!](\theta[X/r])} [\![\mathcal{C}']\!]\varphi(\theta[X/r][Y/t]) \\
&= \bigcup_{r \in [\![e_1]\!]\theta} [\![let\ Y = s\ in\ \mathcal{C}']\!]\varphi(\theta[X/r]) \\
&= [\![let\ X = e_1\ in\ let\ Y = s\ in\ \mathcal{C}']\!]\varphi\theta \\
&= [\![let\ X = e_1\ in\ \mathcal{C}]\!]\varphi\theta
\end{aligned}
$$

where:

- (1) : As $Y \in BV(\mathcal{C})$ and $BV(\mathcal{C}) \cap FV(e_1) = \emptyset$ by hypothesis, then $Y \notin FV(e_1)$. Besides $Y \notin vran(\theta)$ by the variable convention, thus $Y \notin FV(e_1\theta)$ and $e_1\theta[Y/t] \equiv e_1\theta$. But then we can chain $[\![e_1]\!](\theta[Y/t]) = [\![e_1\theta[Y/t]]\!] = [\![e_1\theta]\!] = [\![e_1]\!]\theta$.
- (2) : We may assume $X \neq Y$ by α-conversion, so $X \notin dom([Y/t])$; we may assume $X \notin var(t)$ by lemma 5, so $X \notin vran([Y/t])$. But then we can

apply the substitution lemma to get, for any $e \in LExp_\perp$, $e[X/r][Y/t] \equiv e[Y/t][X/r[Y/t]] \equiv e[Y/t][X/r]$, as $Y \notin var(r)$ by lemma 5 . Hence $[X/r][Y/t] = [Y/t][X/r]$.

- (3) : As $FV(s) \subseteq FV(\mathcal{C})$ and $X \notin FV(\mathcal{C})$ by hypothesis, then $X \notin FV(s)$. Besides $X \notin vran(\theta)$ by the variable convention, thus $X \notin FV(s\theta)$ and $s\theta[X/r] \equiv s\theta$.But then we can chain $[\![s]\!]\theta = [\![s\theta]\!] = [\![s\theta[X/r]]\!] = [\![s]\!](s\theta[X/r])$.

Now we are ready to prove lemma 1.

Proof (For lemma 1). By a case distinction. It is not a surprise that the most difficult step was (Fapp), as the essence of the transformation is concentrated in this step.

(Fapp) As we are working with the transformed program the rule used in this step must be of the shape $R = (f(\overline{p}) \to let\ \overline{Y = X}\ in\ r[\overline{X/Y}])$ such that $\overline{X} = FV(r)$, where $(f(\overline{p}) \to r)$ is the original rule. Assume the step was:

$$f(\overline{p})\sigma \to^{rt} (let\ \overline{Y = X}\ in\ r[\overline{X/Y}])\sigma$$

Without loss of generality we may assume $dom(\sigma) \subseteq FV(R) = FV(f(\overline{p}) \to r)$, as free variables are preserved by $\tau()$, as stated by lemma 6. Then $dom(\sigma) \cap \overline{Y} \subseteq FV(R) \cap \overline{Y} = \emptyset$, as the variables in $\overline{Y}$ are fresh wrt the variables in R, by definition of $\tau()$. Besides, $FV(r[\overline{X/Y}]) = \overline{Y}$, as $\overline{X} = FV(r)$, so $r[\overline{X/Y}]\sigma \equiv r[\overline{X/Y}]$. Hence we can reformulate the step as:

$$f(\overline{p})\sigma \to^{rt} let\ \overline{Y = X\sigma}\ in\ r[\overline{X/Y}]$$

– If $\overline{X} = \emptyset$ then R remains the same as in the original program, $R = (f(\overline{p}) \to r)$, and r is ground, so the step was $f(\overline{p})\sigma \to^{rt} r\sigma \equiv r$. Then given $\theta \in CSubst_\perp$ such that $\vdash_{CRWL_{let}} r\theta \twoheadrightarrow t$, as r is ground then $\vdash_{CRWL_{let}} r \equiv r\theta \twoheadrightarrow t$. Besides, given $\overline{Z} = var(\overline{p})$ it is easy to prove that $\forall \gamma \in LSubst_\perp$, $i \in \{1, \ldots, n\}$, it happens $\vdash_{CRWL_{let}} p_i\gamma \twoheadrightarrow p_i[\overline{Z/\perp}]$ (by induction on the structure of $CTerm$), and we can do:

$$\frac{\overline{p_1\sigma\theta \twoheadrightarrow p_1[\overline{Z/\perp}]}\ \ldots\ \overline{p_n\sigma\theta \twoheadrightarrow p_n[\overline{Z/\perp}]}\quad \overline{r[\overline{Z/\perp}] \equiv r \twoheadrightarrow t}}{f(\overline{p})\sigma\theta \twoheadrightarrow t}\ OR$$

using the instance $(f(\overline{p}) \to r)[\overline{Z/\perp}] \in [\mathcal{P}]_\perp$.

– If $\overline{X} \neq \emptyset$, given $\theta \in CSubst_\perp$ such that $\vdash_{CRWL_{let}} (let\ \overline{Y = X\sigma}\ in\ r[\overline{X/Y}])\theta \twoheadrightarrow t$, by the variable convention $\overline{Y} \cap dom(\theta) = \emptyset$, and besides $FV(r[\overline{X/Y}]) = \overline{Y}$, as $\overline{X} = FV(r)$, hence $r[\overline{X/Y}]\theta \equiv r[\overline{X/Y}]$ and the derivation was:

$$\frac{X_1\sigma\theta \twoheadrightarrow s_1\quad \dfrac{\ldots}{(let\ Y_2 = X_2\sigma\theta\ in\ \ldots\ in\ r[\overline{X/Y}][Y_1/s_1] \twoheadrightarrow t}}{(let\ \overline{Y = X\sigma}\ in\ r[\overline{X/Y}])\theta \equiv let\ \overline{Y = X\sigma\theta}\ in\ r[\overline{X/Y}] \twoheadrightarrow t}\ Let$$

But as $\overline{X}$ is linear and so does $\overline{Y}$, every $Y_j \in \overline{Y}$ is different from every $X_i \in \overline{X}$, and $\overline{Y} \cap (vran(\sigma) \cup vran(\theta)) = \emptyset$ by the variable convention, then $\forall i, j\ X_i\sigma\theta[Y_j/s_j] \equiv X_i\sigma\theta$, and so $\vdash_{CRWL_{let}} let\ \overline{Y = X\sigma\theta}\ in\ r[\overline{X/Y}] \twoheadrightarrow t$ iff for every $X_i \in \overline{X}$ exists some $s_i \in CTerm_\perp$ such that $\vdash_{CRWL_{let}} X_i\sigma\theta \twoheadrightarrow s_i$, and

$\vdash_{CRWL_{let}} r[\overline{X/Y}][\overline{Y/s}] \twoheadrightarrow t$.
Besides, as $FV(r[\overline{X/Y}]) = \overline{Y}$, as $\overline{X} = FV(r)$, then $r[\overline{X/Y}]\ [\overline{Y/s}] \equiv r[\overline{X/s}]$, hence $\vdash_{CRWL_{let}} r[\overline{X/s}] \twoheadrightarrow t$ and we can do:

$$\dfrac{\dfrac{\forall X_i \in \overline{X} \cap var(\overline{p}),\ X_i\sigma\theta \twoheadrightarrow s_i \text{ + lemma 9}}{\overline{p}\sigma\theta \twoheadrightarrow \overline{p}([\overline{X/s}])|_{var(\overline{p})} \equiv \overline{p}[\overline{X/s}]} \ (*)}{f(\overline{p})\sigma\theta \twoheadrightarrow t}\ OR$$

where (*) is the derivation:

$$\dfrac{\dfrac{\dfrac{r[\overline{X/s}] \twoheadrightarrow t}{\text{lemma 7 + } t \in [\![r]\!][\overline{X/s}]}}{t \in [\![let\ \overline{Y = X}\ in\ r[\overline{X/Y}]]\!][\overline{X/s}]}}{(let\ \overline{Y = X}\ in\ r[\overline{X/Y}])[\overline{X/s}] \twoheadrightarrow t}$$

using the instance $(f(\overline{p}) \to let\ \overline{Y = X}\ in\ r[\overline{X/Y}])[\overline{X/s}]$ of $[\mathcal{P}]_\perp$.

(RBind) This is a particular case of the rule (Bind) of $\to^{ct}$, see proof in [20].

(Elim) This is a particular case of the rule (Elim) of $\to^{ct}$, see proof in [20].

(Flat$_1$) Let us define a new rule:

(Dist) $\mathcal{C}[let\ X = e_1\ in\ e_2] \to let\ X = e_1\ in\ \mathcal{C}[e_2]$ for every $\mathcal{C} \neq []$ such that $BV(\mathcal{C}) \cap FV(e_1) = \emptyset$ and $X \notin FV(\mathcal{C})$

This rule introduces non-termination for every program, but it will be useful because it generalizes the let distribution rules (Flat$_1$) and (Flat$_2$). We will see that a (Dist) step leaves the hypersemantics untouched. But now, as $[\![\mathcal{C}[e]]\!] = [\![\mathcal{C}]\!][\![e]\!]$ by lemma 11, we can chain $[\![\mathcal{C}[let\ X = e_1\ in\ e_2]]\!] = [\![\mathcal{C}[let\ X = e_1\ in\ []]\!][\![e_2]\!] = [\![let\ X = e_1\ in\ \mathcal{C}]\!][\![e_2]\!] = [\![let\ X = e_1\ in\ \mathcal{C}[e_2]]\!]$, applying lemma 12 in the third step. Now as every (Flat$_1$) step is a particular case of a (Dist) step then (Flat$_1$) leaves the hypersemantics untouched.

(Flat$_2$) As every (Flat$_2$) step is a particular case of a (Dist) step then (Flat$_2$) leaves the hypersemantics untouched.

(LetIn$_1$) Let us define a new rule:

(CLetIn) $\mathcal{C}[e_1] \to_l let\ X = e_1\ in\ \mathcal{C}[X]$, $\forall \mathcal{C} \neq []$, if $BV(\mathcal{C}) \cap FV(e_1) = \emptyset$, for $X \in \mathcal{V}$ fresh

This rule introduces non-termination for every program, but it will be useful to reason about the programs, as it leaves the hypersemantics untouched; and because it generalizes the let distribution rules (LetIn$_1$) and (LetIn$_2$). Given $\theta \in CSubst_\perp$:

$$\begin{array}{ll}
[\![(let\ X = e_1\ in\ \mathcal{C}[X])\theta]\!] & \\
= [\![let\ X = e_1\theta\ in\ \mathcal{C}\theta[X]]\!] & \text{variable convention} \\
= \bigcup_{t_1 \in [\![e_1\theta]\!]} [\![(\mathcal{C}\theta[X])[X/t_1]]\!] & \text{lemma 10} \\
= \bigcup_{t_1 \in [\![e_1\theta]\!]} [\![\mathcal{C}\theta[t_1]]\!] & \text{variable convention} \\
= [\![\mathcal{C}\theta[e_1\theta]]\!] & BV(\mathcal{C}) \cap FV(e_1) = \emptyset \\
= [\![(\mathcal{C}[e_1])\theta]\!] & \text{variable convention}
\end{array}$$

But then, as every (LetIn$_1$) step is a particular case of a (CLetIn) step then (LetIn$_1$) leaves the hypersemantics untouched as (CLetIn) does.

(LetIn$_2$) As every (LetIn$_2$) step is a particular case of a (CLetIn) step then (LetIn$_2$) leaves the hypersemantics untouched as (CLetIn) does.
(Contx) By the monotonicity under contexts of the hypersemantics (see [20]).

Now the tools for proving the main results concerning soundness of the simulation are available.

Proof (For Theorem 4). It is straightforward to extend lemma 1 to any number of steps by a simple induction on the length of the derivation. But then $\tau(\mathcal{P}) \vdash e \to^{rt\,*} e'$ implies $[\![e']\!]^{\tau(\mathcal{P})}_{CRWL_{let}} \Subset [\![e]\!]^{\tau(\mathcal{P})}_{CRWL_{let}}$, so $[\![e']\!]^{\tau(\mathcal{P})}_{CRWL_{let}} = [\![e']\!]^{\tau(\mathcal{P})}_{CRWL_{let}}\epsilon \subseteq [\![e]\!]^{\tau(\mathcal{P})}_{CRWL_{let}}\epsilon = [\![e]\!]^{\tau(\mathcal{P})}_{CRWL_{let}}$. On the other hand *b)* is a consequence of *a)*, as $\forall t \in CTerm_\perp$ we have $t \in [\![t]\!]$, so $t \in [\![t]\!] \subseteq [\![e]\!]$, by *a)*.

Proof (For Theorem 5). Assume $e \to^{rt\,*}_{\tau(\mathcal{P})} t$, then by Theorem 4 that implies $e \to^{ct\,*}_{\tau(\mathcal{P})} t$, in other words, $t \in [\![e]\!]^{\tau(\mathcal{P})}$. But $[\![e]\!]^{\tau(\mathcal{P})} = [\![e]\!]^{\mathcal{P}}$ by Theorem 3, hence $t \in [\![e]\!]^{\mathcal{P}}$. In other words, $e \to^{ct\,*}_{\mathcal{P}} t$.

Regarding completeness of $\to^{rt}$ wrt $\to^{ct}$, we will base on the following technical lemmas:

Lemma 13. *For every $e \in Exp$, $t \in CTerm_\perp$ and $p \in CTerm$ linear:*

a) $|e| \sqsubseteq e$.
b) If $t \sqsubseteq |e|$ *then* $t \sqsubseteq e$.
c) Given $\theta \in CSubst_\perp$ *such that* $dom(\theta) \subseteq FV(p)$, *if* $p\theta \sqsubseteq |e|$ *then* $\exists \sigma \in Subst$ *such that* $dom(\sigma) = dom(\theta)$, $p\sigma \equiv e$ *and* $\theta \sqsubseteq \sigma$.

Proof (For lemma 13).

a) By induction on the structure of e:
Base cases
- $e \equiv X$: Then $|e| \equiv X \sqsubseteq X \equiv e$.
- $e \equiv c \in CS^0$: Then $|e| \equiv c \sqsubseteq c \equiv e$.
- $e \equiv f \in FS^0$: Then $|e| \equiv \perp \sqsubseteq e$.

Inductive steps
- $e \equiv c(e_1, \ldots, e_n)$ for $c \in CS$: Then $|e| \equiv c(|e_1|, \ldots, |e_n|) \sqsubseteq_{IH} c(e_1, \ldots, e_n) \equiv e$.
- $e \equiv f(e_1, \ldots, e_n)$ for $f \in FS$: Then $|e| \equiv \perp \sqsubseteq e$.

b) Then $t \sqsubseteq |e| \sqsubseteq e$, by *a)*.
c) By induction on the structure of $p\theta$:
Base cases
- $p\theta \equiv Y \in \mathcal{V}$: Then $p\theta \equiv Y \sqsubseteq |e|$ implies $Y \equiv |e|$ and so $Y \equiv e$. But then we can take $\sigma = \theta$ to get $p\sigma \equiv p\theta \equiv Y \equiv e$, $\theta \sqsubseteq \sigma$ as $\sigma \sqsubseteq \sigma$, and $dom(\sigma) = dom(\theta)$.
- $p\theta \equiv c \in CS^0$: Then $p\theta \equiv c \sqsubseteq |e|$ implies $c \equiv |e|$ and so $c \equiv e$. But then we can take $\sigma = \theta$ to get $p\sigma \equiv p\theta \equiv c \equiv e$, $\theta \sqsubseteq \sigma$ as $\sigma \sqsubseteq \sigma$, and $dom(\sigma) = dom(\theta)$.
- $p\theta \equiv \perp$: Then $p \equiv X \in \mathcal{V}$, and $\theta = [X/\perp]$, as $dom(\theta) \subseteq FV(p) = \{X\}$. But then we can choose $\sigma = [X/e]$ to get $p\sigma \equiv X[X/e] \equiv e$, $\theta = [X/\perp] \sqsubseteq [X/e] \equiv \sigma$, and $dom(\sigma) = \{X\} = dom(\theta)$.

Inductive steps

- $p\theta \equiv c(s_1, \ldots, s_n)$ with $p \equiv X \in \mathcal{V}$: Then $dom(\theta) \subseteq FV(p)$ implies $dom(\theta) = \{X\}$, so $\theta = [X/X\theta] = [X/p\theta]$. As $\theta \in CSubst_\perp, p \in CTerm$ then $p\theta \in CTerm_\perp$ and so $p\theta \sqsubseteq |e|$ implies $p\theta \sqsubseteq e$ by *b)*. But then we can choose $\sigma = [X/e]$ to get $p\sigma \equiv X[X/e] \equiv e$, $\theta = [X/p\theta] \sqsubseteq [X/e] \equiv \sigma$, and $dom(\sigma) = \{X\} = dom(\theta)$.
- $p\theta \equiv c(p_1\theta, \ldots, p_n\theta)$ with $p \equiv c(p_1, \ldots, p_n)$. Then $p\theta \equiv c(p_1\theta, \ldots, p_n\theta) \sqsubseteq |e|$ implies $|e| \equiv c(|e_1|, \ldots, |e_n|)$ for $e \equiv c(e_1, \ldots, e_n)$ such that $\forall i,\ p_i\theta \sqsubseteq |e_i|$. As p is linear and $dom(\theta) \subseteq FV(p)$ then if for every i we define $\theta_i = \theta|_{FV(p_i)}$ then $\theta = \theta_1 \uplus \ldots \uplus \theta_n$. But then for every i we have $p_i\theta_i \sqsubseteq |e_i|$ to which we can apply the IH to get $\exists \sigma_i \in Subst$ such that $p_i\sigma_i \equiv e_i$, $\theta_i \sqsubseteq \sigma_i$ and $dom(\sigma_i) = dom(\theta_i)$. But as p is linear then $\sigma = \sigma_1 \uplus \ldots \uplus \sigma_n$ is correctly defined and $p\sigma \equiv c(p_1\sigma_1, \ldots, p_n\sigma_n) \equiv c(e_1, \ldots, e_n) \equiv e$, $\theta \sqsubseteq \sigma$ and $dom(\sigma) = dom(\sigma_1) \cup \ldots \cup dom(\sigma_n) = dom(\theta_1) \cup \ldots \cup dom(\theta_n) = dom(\theta)$.

Note that lemma 13 *a)* is not true in general for $e \in LExp$ as $c(\perp) \equiv |let\ X = loop\ in\ c(X)| \not\sqsubseteq let\ X = loop\ in\ c(X)$, so it happens for lemma 13 *b)*, as a consequence. Again lemma 13 is not true in general for $e \in LExp$, just taking $p \equiv c(X), \theta = [X/\perp], e \equiv let\ X = loop\ in\ c(X)$: $p\theta \equiv c(\perp) \sqsubseteq c(\perp) \equiv |let\ X = loop\ in\ c(X)|$, but $\nexists \sigma \in LSusbt$ such that $p\sigma \equiv e$.

The following lemma shows that using the rules for $\rightarrow^{rt}$ except **(Fapp)**, any expression $e \in LExp$ can be transformed to a 'flat' fully developed form with respect to *let*s.

Lemma 14 (Peeling lemma for $\rightarrow^{rt}$). *For every $e \in LExp$ one has*

$$e \rightarrow^{rt\,*} let\ \overline{X = a}\ in\ b$$

such that:

- $\forall a_i \in \overline{a}, a_i \in Exp$ *(i.e., there are no nested lets)*
- $b \in Exp$ *(i.e., it is a let-free body)*
- $\forall a_i \in \overline{a}$, a_i *is function rooted or* $a_i \in FV(let\ \overline{X = a}\ in\ b)$ *(i.e., no applicable binding remains)*

Besides (Fapp) was not used in that derivation (and therefore the $CRWL_{let}$-hypersemantics and the shell remain untouched).

Proof (For lemma 14). Through this proof we will assume α-conversion when needed to fulfil the conditions of application of rules of $\rightarrow^{rt}$. We proceed by induction on the structure of e:

Base cases If $e \equiv Y \in \mathcal{V}$ or $e \equiv h \in \Sigma^0$ then the lemma holds for $e \rightarrow^{rt\,0} e$ with $\overline{X} = \emptyset$.

Inductive steps

- $e \equiv h(e_1, \ldots, e_n)$: Then by IH over each e_i we have $e_i \rightarrow^{rt\,*} let\ \overline{X_i = a_i}\ in\ b_i$ under the conditions stipulated, so we can do:

$$
\begin{array}{ll}
& h(e_1, \ldots, e_n) \\
& \rightarrow^{rt\,*} h(let\ \overline{X_1 = a_1}\ in\ b_1, \ldots, let\ \overline{X_n = a_n}\ in\ b_n) \qquad (1) \\
& \rightarrow^{rt\,*} let\ \overline{X_1 = a_1}\ in\ \ldots\ let\ \overline{X_n = a_n}\ in\ h(b_1, \ldots, b_n) \quad (2)
\end{array}
$$

(1) by IH; (2) by (Flat_1^*). But then:

- $\forall a \in \overline{a_1} \cup \ldots \cup \overline{a_n},\ a \in Exp$ by IH.
- $b_1, \ldots, b_n \in Exp$ by IH, so $h(b_1, \ldots, b_n) \in Exp$.
- $\forall a \in \overline{a_1} \cup \ldots \cup \overline{a_n}$, by IH we have that a is function rooted or $a \in FV(let\ \overline{X_i = a_i}\ in\ b_i)$, for the corresponding i. In the latter case that implies we have $a \in FV(h(let\ \overline{X_1 = a_1}\ in\ b_1, \ldots, let\ \overline{X_n = a_n}\ in\ b_n))$ by definition, hence $a \in FV($ $let\ \overline{X_1 = a_1}\ in\ \ldots\ let\ \overline{X_n = a_n}\ in\ h(b_1, \ldots, b_n))$ as free variables are preserved by (Flat_1) steps, even when put in non trivial contexts (easy to check).

• $e \equiv let\ X = e_1\ in\ e_2$: Then by IH over e_1 and e_2 we can do:

$$
\begin{array}{ll}
& let\ X = e_1\ in\ e_2 \rightarrow^{rt\,*} let\ X \\
& = (let\ \overline{X_1 = a_1}\ in\ b_1)\ in\ (let\ \overline{X_2 = a_2}\ in\ b_2) \qquad (1) \\
& \rightarrow^{rt\,*} let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ let\ \overline{X_2 = a_2}\ in\ b_2 \quad (2)
\end{array}
$$

(1) by IH; (2) by (Flat_2^*). Then as $b_1 \in Exp$ by IH, we have the following possibilities:

a) b_1 is function rooted or $b_1 \in FV(let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ let\ \overline{X_2 = a_2}\ in\ b_2)$: Then $b_1 \in Exp$ by IH, and it is easy to check that the other conditions of the lemma are also fulfilled by IH, as (LetIn_1) also preserves free variables.

b) b_1 is constructor rooted or $b_1 \notin FV(let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ let\ \overline{X_2 = a_2}\ in\ b_2)$. In other words, b_1 is constructor rooted or $b_1 \in BV(let\ \overline{X_1 = a_1}\ in\ let\ X = [\,]\ in\ let\ \overline{X_2 = a_2}\ in\ b_2)$. Then we have the following possibilities:

   i) $b_1 \in CTerm$ such that every variable in $var(b_1)$ is bound in its context : Then we can perform a (RBind) step:

$$
\begin{array}{l}
let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ let\ \overline{X_2 = a_2}\ in\ b_2 \\
\rightarrow^{rt} let\ \overline{X_1 = a_1}\ in\ let\ \overline{X_2 = a_2[X/b_1]}\ in\ b_2[X/b_1]
\end{array}
$$

   But then:

   - $\forall a \in \overline{a_1},\ a \in Exp$ by IH. Besides $\forall a \in \overline{a_2},\ a \in Exp$ by IH, hence $a[X/b_1] \in Exp$ as $[X/b_1] \in CSubst$.
   - $b_2 \in Exp$ by IH, so $b_2[X/b_1] \in Exp$, as $[X/b_1] \in CSubst$.
   - $\forall a \in \overline{a_1}$, by IH we have that a is function rooted or $a \in FV(let\ \overline{X_1 = a_1}\ in\ b_1)$. In the latter case that implies $a \in FV(let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ let\ \overline{X_2 = a_2}\ in\ b_2)$ by definition, hence $a \in FV(let\ \overline{X_1 = a_1}\ in\ let\ \overline{X_2 = a_2[X/b_1]}\ in\ b_2[X/b_1])$ as free variables are preserved by (RBind) steps, even when put in non trivial contexts (easy to check).
     Besides $\forall a \in \overline{a_2}$, by IH we have that a is function rooted or $a \in FV(let\ \overline{X_2 = a_2}\ in\ b_2)$. In the first case $a_2[X/b_1]$ obviously remains function rooted, and in the latter $a_2[X/b_1] \equiv a_2$, as a_2 was a free in its context, and so it is also free in the new context established by (RBind), which also preserves free variables.

ii) $b_1 \equiv \mathcal{C}[s_1, \ldots, s_n]$ for $\mathcal{C} \neq []$ (otherwise we would be in case *a)*) a many hole c-context and $s_1, \ldots, s_n \in Exp$ the maximal (in the order of positions) subexpressions of b_1 which are function rooted or variables free in their contexts. Those free s_i are also not bound in their contexts, and so we can perform several (LetIn_1) or (LetIn_2) steps:

$$\begin{array}{l} \mathit{let}\ \overline{X_1 = a_1}\ \mathit{in}\ \mathit{let}\ X = b_1\ \mathit{in}\ \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ b_2 \\ \equiv \mathit{let}\ \overline{X_1 = a_1}\ \mathit{in}\ \mathit{let}\ X = \mathcal{C}[s_1, \ldots, s_n]\ \mathit{in}\ \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ b_2 \\ \to^{rt\,*} \mathit{let}\ \overline{X_1 = a_1}\ \mathit{in}\ \mathit{let}\ \overline{Y = s}\ \mathit{in}\ \mathit{let}\ X = \mathcal{C}[\overline{Y}]\ \mathit{in}\ \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ b_2 \end{array}$$

But then we are in the previous case with $\mathcal{C}[\overline{Y}]$ playing the role of b_1, because every $s_i \in Exp$ and besides is function rooted or a free variable, and besides (LetIn_1) and (LetIn_2) preserve free variables.

The following pair of technical but interesting lemmas will be needed to cope with the renaming implicitly introduced by (LetIn_2).

Lemma 15. *For every $p \in CTerm$ linear, $\sigma \in LSubst_\perp$, $\overline{X}, \overline{Y}$ linear and finite tuples of variables and $e \in Exp$ such that $dom(\sigma) \subseteq FV(p)$ if $p\sigma \equiv e[\overline{X/Y}]$ then $\exists \sigma' \in LSubst_\perp$ such that $dom(\sigma') = dom(\sigma)$, $p\sigma' \equiv e$ and $\sigma'[\overline{X/Y}] = \sigma\ [FV(p)]$. Besides σ' is in the same subset of $LSubst_\perp$ as σ (is total when σ is, is constructed when σ is, ...).*

Proof (For lemma 15). Note that $dom(\sigma') = dom(\sigma) \subseteq FV(p)$ and $\sigma'[\overline{X/Y}] = \sigma\ [FV(p)]$ does not imply $\sigma'[\overline{X/Y}] = \sigma$, as in general $dom(\sigma'[\overline{X/Y}]) \neq dom(\sigma')$. We proceed by induction on the structure of p:

Base cases

- $p \equiv U \in \mathcal{V}$: Then by hypothesis $U\sigma \equiv p\sigma \equiv e[\overline{X/Y}]$ and $dom(\sigma) \subseteq FV(p) = \{U\}$, hence $\sigma = [U/e[\overline{X/Y}]]$. But then we can take $\sigma' = [U/e]$, with which $p\sigma' \equiv e$, $dom(\sigma') = \{U\} = dom(\sigma)$, and given $Z \in FV(p) = \{U\}$ then $Z = U$ and so $Z\sigma'[\overline{X/Y}] \equiv U[U/e][\overline{X/Y}] \equiv U\sigma \equiv Z\sigma$.
- $p \equiv c \in CS^0$: Then by hypothesis $dom(\sigma) \subseteq FV(p) = \emptyset$, hence $\sigma = \epsilon$. Besides by hypothesis $c \equiv c\epsilon \equiv p\sigma \equiv e[\overline{X/Y}]$, therefore $e \equiv c$. But then we can take $\sigma' = \epsilon$, with which $p\sigma' \equiv c \equiv e$, $dom(\sigma') = \emptyset = dom(\sigma)$, and $\sigma'[\overline{X/Y}] = \sigma\ [FV(p)]$ trivially as $FV(p) = \emptyset$.

Inductive step Then $p \equiv c(p_1, \ldots, p_n)$: As by hypothesis $p\sigma \equiv e[\overline{X/Y}]$ then it must happen that $e \equiv c(e_1, \ldots, e_n)$ such that $p\sigma \equiv c(p_1\sigma, \ldots, p_n\sigma) \equiv c(e_1[\overline{X/Y}], \ldots, e_n[\overline{X/Y}]) \equiv e[\overline{X/Y}]$. As p is linear then if for each $i \in \{1, \ldots, n\}$ we define $\sigma_i = \sigma|_{FV(p_i)}$ then $\sigma_1 \uplus \ldots \uplus \sigma_n$ is correctly defined and besides $\sigma = \sigma_1 \uplus \ldots \uplus \sigma_n$, as $dom(\sigma) \subseteq FV(p)$ by hypothesis, and $p_i\sigma_i \equiv e_i[\overline{X/Y}]$ for each i. But then we can apply the IH to each i to get some $\sigma'_i \in Subst$ such that $p_i\sigma' \equiv e_i$, $dom(\sigma'_i) = dom(\sigma_i) \subseteq FV(p_i)$ and $\sigma'_i[\overline{X/Y}] = \sigma_i\ [FV(p_i)]$. So $\sigma' = \sigma'_1 \uplus \ldots \uplus \sigma'_n$ is correctly defined and besides:

- $p\sigma' \equiv c(p_1\sigma', \ldots, p_n\sigma') \equiv c(p_1\sigma'_1, \ldots, p_n\sigma'_n) \equiv$ $c(e_1[\overline{X/Y}], \ldots, e_n[\overline{X/Y}]) \equiv e[\overline{X/Y}]$.
- $dom(\sigma') = dom(\sigma'_1) \uplus \ldots \uplus dom(\sigma'_n) = dom(\sigma_1) \uplus \ldots \uplus dom(\sigma_n) = dom(\sigma)$.
- $\sigma'[\overline{X/Y}] = \sigma\ [FV(p)]$ because given $U \in FV(p)$ then, as p is linear, $\exists!\ i \in \{1, \ldots, n\}$ such that $U \in FV(p_i)$. But then $U\sigma'[\overline{X/Y}] \equiv U\sigma'_i[\overline{X/Y}] \equiv U\sigma_i \equiv U\sigma$, as $\sigma'_i[\overline{X/Y}] = \sigma_i\ [FV(p_i)]$ by IH.

Lemma 16. *For any program $\mathcal{P}$, $e \in LExp$, $t \in CTerm_\perp$, $\overline{X},\overline{Y}$ linear and finite tuples of variables, if $\overline{X} \cap var(t) = \emptyset$ and $\mathcal{P} \vdash e[\overline{X/Y}] \twoheadrightarrow t$ then $\exists t' \in CTerm_\perp$ such that $\mathcal{P} \vdash e \twoheadrightarrow t'$ with a derivation of the same size and structure that $t'[\overline{X/Y}] \equiv t$.*

Proof (For lemma 16). We will prove this lemma for $\overline{X} = \{X\}$ and $\overline{Y} = \{Y\}$, that is "For every $e \in LExp$, $t \in CTerm_\perp$, $X, Y \in \mathcal{V}$ and under any program, if $X \notin var(t)$ and $\mathcal{P} \vdash_{CRWL_{let}} e[X/Y] \twoheadrightarrow t$ then $\exists t' \in CTerm_\perp$ such that $\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t'$ with a derivation of the same size and structure and $t'[X/Y] \equiv t$.". The extension to finite sets of variables is a trivial induction on the cardinal of those sets. We proceed by induction on the structure of the derivation for $\vdash_{CRWL_{let}} e[X/Y] \twoheadrightarrow t$:

Base cases

B If $\vdash_{CRWL_{let}} e[X/Y] \twoheadrightarrow \perp$ then $\vdash_{CRWL_{let}} e \twoheadrightarrow \perp$ by **B** and $t'[X/Y] \equiv \perp$ $[X/Y] \equiv \perp \equiv t$.

RR Then $e[X/Y] \in \mathcal{V}$ and we have the following possibilities:

- $e \equiv X$: Then $\vdash_{CRWL_{let}} e[X/Y] \equiv Y \twoheadrightarrow Y \equiv t$, but then $\vdash_{CRWL_{let}} e \equiv X \twoheadrightarrow X \equiv t'$ by **RR** and $t'[X/Y] \equiv X[X/Y] \equiv Y \equiv t$.
- $e \equiv Z \in \mathcal{V}$ such that $Z \neq X$: Then $\vdash_{CRWL_{let}} e[X/Y] \equiv Z \twoheadrightarrow Z \equiv t$, but then $\vdash_{CRWL_{let}} e \equiv Z \twoheadrightarrow Z \equiv t'$ by **RR** and $t'[X/Y] \equiv Z[X/Y] \equiv Z \equiv t$.

Inductive steps

DC Then it must happen $e \equiv c(e_1, \ldots, e_n)$ and the derivation is:

$$\frac{e_1[X/Y] \twoheadrightarrow t_1 \ \ldots \ e_n[X/Y] \twoheadrightarrow t_n}{e[X/Y] \equiv c(e_1[X/Y], \ldots, e_n[X/Y]) \twoheadrightarrow c(t_1, \ldots, t_n) \equiv t} \ DC$$

By IH, for each $i \in \{1, \ldots, n\}$ there must exists some $t'_i \in CTerm_\perp$ such that $\vdash_{CRWL_{let}} e_i \twoheadrightarrow t'_i$ and $t'_i[X/Y] \equiv t_i$. But then we can do:

$$\frac{e_1 \twoheadrightarrow t'_1 \ \ldots \ e_n \twoheadrightarrow t'_n}{e \equiv c(e_1, \ldots, e_n) \twoheadrightarrow c(t'_1, \ldots, t'_n) \equiv t'} \ DC$$

But then $t'[X/Y] \equiv c(t'_1[X/Y], \ldots, t'_n[X/Y]) \equiv c(t_1, \ldots, t_n) \equiv t$.

OR Then it must happen $e \equiv f(e_1, \ldots, e_n)$ such that for some $R = (f(\overline{p}) \to r) \in \mathcal{P}$ and $\theta \in CSusbt_\perp$ we have a derivation like:

$$\frac{e_1[X/Y] \twoheadrightarrow p_1\theta \ \ldots \ e_n[X/Y] \twoheadrightarrow p_n\theta \quad r\theta \twoheadrightarrow t}{e[X/Y] \equiv f(e_1[X/Y], \ldots, e_n[X/Y]) \twoheadrightarrow t} \ OR$$

We assume that R is fresh and $dom(\theta) \subseteq FV(R)$ without loss of generality. But then, as p is linear we can decompose θ as $\theta = \theta_1 \uplus \ldots \theta_n \uplus \theta_{vE}$, where $\theta_i = \theta|_{FV(p_i)}$ and $\theta_{vE} = \theta|_{vExtra(R)}$. Besides, by IH, for each $i \in \{1, \ldots, n\}$ there must exists some $s_i \in CTerm_\perp$ such that $\vdash_{CRWL_{let}} e_i \twoheadrightarrow s_i$ and $s_i[X/Y] \equiv p_i\theta \equiv p_i\theta_i$. Then we can apply lemma 15 to each i to get some $\theta'_i \in CSubst_\perp$ such that $p_i\theta'_i \equiv s_i$, $\theta'_i[X/Y] = \theta_i$ $[var(p_i)]$ and $dom(\theta'_i) = dom(\theta_i) \subseteq FV(p_i)$. But then

$$\begin{aligned} r\theta &\equiv r(\theta_1 \uplus \ldots \uplus \theta_n \uplus \theta_{vE}) \\ &\equiv r((\theta'_1[X/Y])|_{var(p_1)} \uplus \ldots \uplus (\theta'_n[X/Y])|_{var(p_n)} \\ &\qquad\qquad\qquad\qquad\qquad\qquad \uplus \theta_{vE}) \end{aligned}$$

Furthermore by lemma 4 we assume that the set $\mathcal{U}$ of the free variables used in $\vdash_{CRWL_{let}} e[X/Y] \twoheadrightarrow t$ fulfils $\mathcal{U} \subseteq FV(e[X/Y]) \cup FV(t)$. As obviously $X \notin FV(e[X/Y])$, and $X \notin FV(t)$ by hypothesis, then $X \notin \mathcal{U}$. Besides $FV(r\theta) = FV(r(\theta_1 \uplus \ldots \uplus \theta_n \uplus \theta_{vE})) \subseteq \mathcal{U}$, as $r\theta$ is used in the derivation for $e[X/Y] \twoheadrightarrow t$, and $X \notin FV(r)$ as r is part of the fresh instance, hence $X \notin vran(\theta_{vE})$. With this we will prove that $r((\theta'_1[X/Y])|_{var(p_1)} \uplus \ldots \uplus (\theta'_n[X/Y])|_{var(p_n)} \uplus \theta_{vE}) \equiv r(\theta'_1 \uplus \ldots \uplus \theta'_n \uplus \theta_{vE})[X/Y]$. Given $Z \in FV(r) \subseteq var(p_1) \uplus \ldots \uplus var(p_n) \uplus vExtra(R)$, we have the following possibilities:

- If $Z \in var(p_i)$ for some $i \in \{1, \ldots, n\}$ then

$$\begin{array}{l} Z((\theta'_1[X/Y])|_{var(p_1)} \uplus \ldots \uplus (\theta'_n[X/Y])|_{var(p_n)} \\ \qquad\qquad\qquad\qquad\qquad\qquad \uplus \theta_{vE}) \\ \equiv Z(\theta'_i[X/Y])|_{var(p_i)} \equiv (Z\theta'_i)[X/Y] \\ \equiv Z(\theta'_1 \uplus \ldots \uplus \theta'_n \uplus \theta_{vE})[X/Y] \end{array}$$

- If $Z \in vExtra(R)$ then $Z((\theta'_1[X/Y])|_{var(p_1)} \uplus \ldots \uplus (\theta'_n[X/Y])|_{var(p_n)} \uplus \theta_{vE}) \equiv Z\theta_{vE}$. But $Z \in FV(r)$ which is part of the fresh instance, so $Z \neq X$, and $X \notin vran(\theta_{vE})$ as we saw above, hence $Z\theta_{vE} \equiv (Z\theta_{vE})[X/Y] \equiv Z(\theta_{vE} \uplus \ldots \theta'_n \uplus \theta_{vE})[X/Y]$.

So $\vdash_{CRWL_{let}} r(\theta'_1 \uplus \ldots \uplus \theta'_n \uplus \theta_{vE})[X/Y] \equiv r\theta \twoheadrightarrow t$ and we can apply the IH to get $\vdash_{CRWL_{let}} r(\theta'_1 \uplus \ldots \theta'_n \uplus \theta_{vE}) \twoheadrightarrow t'$ under the conditions stipulated. But then we can do:

$$\frac{e_i \twoheadrightarrow s_i \equiv p_i\theta'_i \quad r(\theta'_1 \uplus \ldots \theta'_n \uplus \theta_{vE}) \twoheadrightarrow t'}{e \equiv f(e_1, \ldots, e_n) \twoheadrightarrow t'}\ OR$$

for $i = 1..n$

Let Then it must happen $e \equiv let\ Z = e_1\ in\ e_2$ and the derivation is:

$$\frac{e_1[X/Y] \twoheadrightarrow t_1 \quad e_2[X/Y][Z/t_1] \twoheadrightarrow t}{e[X/Y] \equiv let\ Z = e_1[X/Y]\ in\ e_2[X/Y] \twoheadrightarrow t}\ Let$$

Then we can apply the IH over $\vdash_{CRWL_{let}} e_1[X/Y] \twoheadrightarrow t_1$ to get $\vdash_{CRWL_{let}} e_1 \twoheadrightarrow t'_1$ such that $t'_1[X/Y] \equiv t_1$, under the conditions stipulated. Furthermore by variable convention $Z \notin dom([X/Y]) \cup vran([X/Y])$, and so we can apply the substitution lemma to get $e_2[X/Y][Z/t_1] \equiv e_2[X/Y][Z/t'_1[X/Y]] \equiv e_2[Z/t'_1][X/Y]$. But then we have $e_2[Z/t'_1][X/Y] \equiv e_2[X/Y]\ [Z/t_1] \twoheadrightarrow t$ and we can apply the IH to get $\vdash_{CRWL_{let}} e_2[Z/t'_1] \twoheadrightarrow t'$ under the conditions stipulated, and we can do:

$$\frac{e_1 \twoheadrightarrow t'_1 \quad e_2[Z/t'_1] \twoheadrightarrow t'}{e \equiv let\ Z = e_1\ in\ e_2 \twoheadrightarrow t'}\ Let$$

Finally we are ready to prove lemma 2 with the help of the auxiliary lemmas above:

Proof (For lemma 2). Through this proof we will assume α-conversion when needed to fulfil the conditions of application of rules of $\rightarrow^{rt}$. We proceed by induction on the size of the $CRWL_{let}$ derivation for $\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t$, measured as the number of rules of $CRWL_{let}$ applied. Let us see which rule was applied at the root of that derivation:

Base cases This cases correspond to $e \twoheadrightarrow \bot$ by **B**, $X \twoheadrightarrow X$ by **RR**, for $X \in \mathcal{V}$, and $c \twoheadrightarrow c$ by **DC**, for $c \in CS^0$. In any of these cases $e \rightarrow^{rt\,0} e \equiv e'$ fulfils the conditions of the lemma, because then $\bot \sqsubseteq |e|$, $X \sqsubseteq X \equiv |X|$ and $c \sqsubseteq c \equiv |c|$.

Inductive steps

DC Then we have

$$\frac{e_1 \twoheadrightarrow t_1 \quad \ldots \quad e_n \twoheadrightarrow t_n}{c(e_1, \ldots, e_n) \twoheadrightarrow c(t_1, \ldots, t_n)}\ DC$$

Then by IH over each $e_i \twoheadrightarrow t_i$ then $e_i \rightarrow^{rt\,*} e'_i$ for some $e'_i \in LExp$ such that $t_i \sqsubseteq |e'_i|$. But then $c(e_1, \ldots, e_n) \rightarrow^{rt\,*} c(e'_1, \ldots, e'_n)$, so we are done as $\forall i, t_i \sqsubseteq |e'_i|$ implies $c(t_1, \ldots, t_n) \sqsubseteq c(|e'_1|, \ldots, |e'_n|) \equiv |c(e'_1, \ldots, e'_n)|$.

OR For the sake of clarity we will present the proof for the case of an application of $f \in FS^1$, the adaptation of this proof to zero or more than one argument is straightforward. Then we have:

$$\frac{e_1 \twoheadrightarrow p_1\theta \quad r\theta \twoheadrightarrow t}{f(e_1) \twoheadrightarrow t}\ OR$$

for some rule $R = (f(p_1) \to r) \in \mathcal{P}$ and $\theta \in CSusbt_\bot$. By IH over $e_1 \twoheadrightarrow p_1\theta$ then $e_1 \rightarrow^{rt\,*} e'_1$ for some $e'_1 \in LExp$ such that $p_1\theta \sqsubseteq |e'_1|$. But then:

$$\begin{array}{ll} f(e_1) \rightarrow^{rt\,*} f(e'_1) & \text{by IH} \\ \rightarrow^{rt\,*} f(let\ \overline{X_1 = a_1}\ in\ b_1) & \text{by the peeling lemma 14} \\ \rightarrow^{rt\,*} let\ \overline{X_1 = a_1}\ in\ f(b_1) & \text{by } (Flat_1^*) \end{array}$$

By the conditions of the peeling lemma we can decompose $\overline{a_1}$ as $\overline{a_1} = \overline{a_1^f} \uplus \overline{a_1^v}$, where $\overline{a_1^f}$ contains those $a \in \overline{a_1}$ which are function rooted and $\overline{a_1^v}$ contains those which are free variables (as (Flat_1) preserves the free variables these remain free in $let\ \overline{X_1 = a_1}\ in\ f(b_1)$). But as in the derivation of the peeling lemma (Fapp) was not applied then by lemma 2 the shell was preserved and so

$$\begin{array}{l} p_1(\theta|_{FV(p_1)}) \equiv p_1\theta \sqsubseteq |e'_1| \equiv |let\ \overline{X_1 = a_1}\ in\ b_1| \\ \equiv |b_1[\overline{X_1^f / \bot}, \overline{X_1^v / a_1^v}] \sqsubseteq |b_1[\overline{X_1^v / a_1^v}]| \end{array}$$

But, as $p_1 \in CTerm$ is lineal, $\theta|_{FV(p_1)} \in CSubst_\bot$ with $dom(\theta|_{FV(p_1)}) \subseteq FV(p_1)$, $b_1 \in Exp$ by the peeling lemma and so $b_1[\overline{X_1^v / a_1^v}] \in Exp$, and $p_1(\theta|_{FV(p_1)}) \sqsubseteq |b_1[\overline{X_1^v / a_1^v}]|$, then we can apply lemma 13 to get some $\sigma_1 \in Subst$ such that $dom(\sigma_1) = dom(\theta|_{FV(p_1)}) \subseteq FV(p_1)$, $p_1\sigma_1 \equiv b_1[\overline{X_1^v / a_1^v}]$ and $\theta|_{FV(p_1)} \sqsubseteq \sigma_1$. But then the conditions in lemma 15 are also fulfilled and so we can apply it to get some $\sigma'_1 \in Subst$ such that $dom(\sigma'_1) = dom(\sigma_1) \subseteq FV(p_1)$, $p_1\sigma'_1 \equiv b_1$ and $\sigma'_1[\overline{X_1^v / a_1^v}] = \sigma_1\ [FV(p_1)]$.

Without loss of generality we assume $dom(\theta) \subseteq FV(f(p_1) \to r)$, so $\theta = \theta|_{FV(p_1)} \uplus \theta|_{vExtra(R)}$. Now we can define $\theta' \in CSubst$ such that $\theta|_{vExtra(R)} \sqsubseteq \theta'$ and $dom(\theta') = vExtra(R)$, just replacing every $\bot$ introduced by $\theta|_{vExtra(R)}$ in its range with some constant or fresh variable. But then $\sigma_1 \uplus \theta'$ is correctly defined and besides $\theta \sqsubseteq \sigma_1 \uplus \theta'$, hence $r\theta \twoheadrightarrow t$ implies $r(\sigma_1 \uplus \theta') \twoheadrightarrow t$ with a derivation of the same size or smaller. Besides:

$$\begin{array}{ll} r(\sigma_1 \uplus \theta') \equiv r((\sigma_1)|_{FV(p_1)} \uplus \theta') & (1) \\ \equiv r((\sigma'_1[\overline{X_1^v / a_1^v}])|_{FV(p_1)} \uplus \theta') & (2) \\ \equiv r(\sigma'_1 \uplus \theta')[\overline{X_1^v / a_1^v}] & (3) \end{array}$$

(1) as $dom(\sigma_1) \subseteq FV(p_1)$;
(2) as $\sigma'_1[\overline{X_1^v/a_1^v}] = \sigma_1\ [FV(p_1)]$;
(3) Because given $Z \in FV(r) \subseteq FV(p_1) \uplus vExtra(R)$:
- If $Z \in FV(p_1)$ then $Z((\sigma'_1[\overline{X_1^v/a_1^v}])|_{FV(p_1)} \uplus \theta') \equiv Z(\sigma'_1[\overline{X_1^v/a_1^v}])|_{FV(p_1)} \equiv (Z\sigma'_1)[\overline{X_1^v/a_1^v}] \equiv Z(\sigma'_1 \uplus \theta')[\overline{X_1^v/a_1^v}]$.
- Otherwise $Z \in vExtra(R)$,
  but then $Z((\sigma'_1[\overline{X_1^v/a_1^v}])|_{FV(p_1)} \uplus \theta') \equiv Z\theta'$. But without loss of generality we assume that R is a fresh instance and so Z is fresh as it is part of R and $Z \notin \overline{X_1^v}$; besides $\overline{X_1^v} \cap vran(\theta') = \emptyset$ by lemma 5, as the variables in $\overline{X_1^v}$ either are bound in a subderivation of $\vdash_{CRWL_{let}} f(e_1) \twoheadrightarrow t$ or are fresh and introduced by $\rightarrow^{rt}$. Hence $Z\theta' \equiv Z\theta'[\overline{X_1^v/a_1^v}] \equiv Z(\sigma'_1 \uplus \theta')[\overline{X_1^v/a_1^v}]$.

So $\vdash_{CRWL_{let}} r(\sigma'_1 \uplus \theta')[\overline{X_1^v/a_1^v}] \equiv r(\sigma_1 \uplus \theta') \twoheadrightarrow t$ and $\overline{X_1^v} \cap var(t) = \emptyset$ by lemma 5, as the variables in $\overline{X_1^v}$ either are bound in a subderivation of $\vdash_{CRWL_{let}} f(e_1) \twoheadrightarrow t$ or are fresh and introduced by $\rightarrow^{rt}$. Then we can apply lemma 16 to get $\vdash_{CRWL_{let}} r(\sigma'_1 \uplus \theta') \twoheadrightarrow t'$ with a derivation of the same size, for some $t' \in CTerm_\perp$ such that $t'[\overline{X_1^v/a_1^v}] \equiv t$. Finally we can apply the IH to this derivation to get $r(\sigma'_1 \uplus \theta') \rightarrow^{rt\,*} e'$ for some $e' \in LExp$ such that $t' \sqsubseteq |e'|$, so we can do:

$$\begin{array}{ll} let\ \overline{X_1 = a_1}\ in\ f(b_1) \equiv let\ \overline{X_1 = a_1}\ in\ f(p_1\sigma'_1) & \\ \equiv let\ \overline{X_1 = a_1}\ in\ f(p_1(\sigma'_1 \uplus \theta')) \rightarrow^{rt} let\ \overline{X_1 = a_1}\ in\ r(\sigma'_1 \uplus \theta') & \text{by (Fapp)} \\ \rightarrow^{rt\,*} let\ \overline{X_1 = a_1}\ in\ e' & \text{by IH} \end{array}$$

Now, as $\overline{a_1^v} \subseteq \mathcal{V}$ then $t' \sqsubseteq |e'|$ implies $t'[\overline{X_1^v/a_1^v}] \sqsubseteq |e'|[\overline{X_1^v/a_1^v}]$ and so $t \equiv t'[\overline{X_1^v/a_1^v}] \sqsubseteq |e'|[\overline{X_1^v/a_1^v}]$. Besides $\overline{X_1^f} \cap var(t) = \emptyset$ for the same reasons that $\overline{X_1^v} \cap var(t) = \emptyset$, but then $t \sqsubseteq |e'|[\overline{X_1^v/a_1^v}]$ implies that any occurrence in $|e'|[\overline{X_1^v/a_1^v}]$ of some $X \in \overline{X_1^f}$ corresponds to an occurrence of $\perp$ in t, in the same position. Hence $t \sqsubseteq |e'|[\overline{X_1^v/a_1^v}]$ implies $t \sqsubseteq |e'|[\overline{X_1^v/a_1^v}, \overline{X_1^f/\perp}] \equiv |let\ \overline{X_1 = a_1}\ in\ e'|$.

Let Then we have

$$\frac{e_1 \twoheadrightarrow t_1 \quad e_2[X/t_1] \twoheadrightarrow t}{let\ X = e_1\ in\ e_2 \twoheadrightarrow t}\ Let$$

Then by IH over $e_1 \twoheadrightarrow t_1$ then $e_1 \rightarrow^{rt\,*} e'_1$ for some $e'_1 \in LExp$ such that $t_1 \sqsubseteq |e'_1|$, so we can do:

$$\begin{array}{l} let\ X = e_1\ in\ e_2 \rightarrow^{rt\,*} let\ X = e'_1\ in\ e_2 \\ \rightarrow^{rt\,*} let\ X = (let\ \overline{X_1 = a_1}\ in\ b_1)\ in\ e_2 \quad (1) \\ \rightarrow^{rt\,*} let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e_2 \quad (2) \end{array}$$

(1) by the peeling lemma 14; (2) by (Flat_2^*). By the conditions of the peeling lemma we can decompose $\overline{a_1}$ as $\overline{a_1} = \overline{a_1^f} \uplus \overline{a_1^v}$, where $\overline{a_1^f}$ contains those $a \in \overline{a_1}$ which are function rooted and $\overline{a_1^v}$ contains those which are free variables (as (Flat_2) preserves the free variables these remain free in $let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e_2$). But as in the derivation of the peeling lemma (Fapp) was not applied then by lemma 2 the shell was preserved and so

$$t_1 \sqsubseteq |e'_1| \equiv |let\ \overline{X_1 = a_1}\ in\ b_1| \equiv |b_1|[\overline{X_1^f/\perp}, \overline{X_1^v/a_1^v}] \sqsubseteq |b_1[\overline{X_1^v/a_1^v}]|$$

Now we have several possibilities, taking into account that $b_1 \in Exp$, by the conditions of the peeling lemma:

a) $b_1 \in CTerm$ such that every variable in $var(b_1)$ is bound in its context : Then as $t_1 \sqsubseteq |b_1[\overline{X_1^v/a_1^v}]|$ we have $t_1 \sqsubseteq b_1[\overline{X_1^v/a_1^v}]$ by lemma 13, so $[X/t_1] \sqsubseteq [X/b_1[\overline{X_1^v/a_1^v}]]$. Hence we have $\vdash_{CRWL_{let}} e_2[X/t_1] \twoheadrightarrow t$ implies $\vdash_{CRWL_{let}} e_2[X/b_1[\overline{X_1^v/a_1^v}]] \twoheadrightarrow t$ with a derivation of the same size or smaller. Besides, $\overline{X_1^v} \cap FV(e_2) = \emptyset$, by the conditions in (Flat$_2$), so $e_2[X/b_1[\overline{X_1^v/a_1^v}]] \equiv e_2[\overline{X_1^v/a_1^v}][X/b_1[\overline{X_1^v/a_1^v}]] \equiv e_2[X/b_1][\overline{X_1^v/a_1^v}]$ by the substitution lemma, as $X \notin (dom([\overline{X_1^v/a_1^v}]) \cup vran([\overline{X_1^v/a_1^v}]))$ by α-conversion. We can do this conversion because $let\ X = (let\ \overline{X_1 = a_1}\ in\ b_1)\ in\ e_2$ was an intermediate expression, in which we have $X \notin FV(\overline{X_1 = a_1}\ in\ b_1)$ because of the abstense of recursive *lets*. So $\vdash_{CRWL_{let}} e_2[X/b_1]\ [\overline{X_1^v/a_1^v}] \equiv e_2[X/b_1[\overline{X_1^v/a_1^v}]] \twoheadrightarrow t$, and then we can apply lemma 16 to get $\vdash_{CRWL_{let}} e_2[X/b_1] \twoheadrightarrow t'$ with a derivation of the same size, for some $t' \in CTerm_\perp$ such that $t'[\overline{X_1^v/a_1^v}] \equiv t$. Finally we can apply the IH to this derivation to get $e_2[X/b_1] \rightarrow^{rt\ *} e'$ for some $e' \in LExp$ such that $t' \sqsubseteq |e'|$, so we can do:

$$\begin{array}{ll} let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e_2 \rightarrow^{rt}\ let\ \overline{X_1 = a_1}\ in\ e_2[X/b_1] & \text{by (RBind)} \\ \rightarrow^{rt\ *} let\ \overline{X_1 = a_1}\ in\ e' & \text{by IH} \end{array}$$

Now, as $\overline{a_1^v} \subseteq \mathcal{V}$ then $t' \sqsubseteq |e'|$ implies $t'[\overline{X_1^v/a_1^v}] \sqsubseteq |e'|[\overline{X_1^v/a_1^v}]$ and so $t \equiv t'[\overline{X_1^v/a_1^v}] \sqsubseteq |e'|[\overline{X_1^v/a_1^v}]$. Besides we can prove that $\overline{X_1^f} \cap var(t) = \emptyset$ in the same way we did in the **OR** case, but then $t \sqsubseteq |e'|[\overline{X_1^v/a_1^v}]$ implies that any occurrence in $|e'|[\overline{X_1^v/a_1^v}]$ of some $X \in \overline{X_1^f}$ corresponds to an occurrence of $\perp$ in t, in the same position. Hence $t \sqsubseteq |e'|[\overline{X_1^v/a_1^v}]$ implies $t \sqsubseteq |e'|[\overline{X_1^v/a_1^v}, \overline{X_1^f/\perp}] \equiv |let\ \overline{X_1 = a_1}\ in\ e'|$.

b) $b_1 \notin CTerm$ or $b_1 \in CTerm$ but some variable in b_1 is not bound in its context.

i) If b_1 is function rooted then $t_1 \sqsubseteq |b_1[\overline{X_1^v/a_1^v}]| \equiv \perp$, hence $t_1 \equiv \perp$ and $\vdash_{CRWL_{let}} e_2[X/t_1] \twoheadrightarrow t$ implies $\vdash_{CRWL_{let}} e_2 \twoheadrightarrow t$ with a derivation of the same size or smaller, to which we can apply the IH to get $e_2 \rightarrow^{rt\ *} e'$ for some $e' \in LExp$ such that $t \sqsubseteq |e'|$, so we can do:

$$\begin{array}{l} let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e_2 \\ \rightarrow^{rt\ *} let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e' \ \text{ by IH} \end{array}$$

Now we can prove that $(\overline{X_1^f} \cup \overline{X_1^v} \cup \{X\}) \cap var(t) = \emptyset$ in the same way we did in the **OR** case, but then $t \sqsubseteq |e'|$ implies that any occurrence in $|e'|$ of some $Y \in \overline{X_1^f} \cup \overline{X_1^v} \cup \{X\}$ corresponds to an occurrence of $\perp$ in t, in the same position. Hence $t \sqsubseteq |e'|$ implies $t \sqsubseteq |e'|[\overline{X_1^v/\perp}, \overline{X_1^f/\perp}, X/\perp] \sqsubseteq |e'|[\overline{X_1^v/a_1^v}, \overline{X_1^f/\perp}, X/\perp] \equiv |let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ e'|$, as b_1 is function rooted.

ii) If $b_1 \equiv Y \in FV(let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e_2)$ then $Y \notin \overline{X_1}$ and $t_1 \sqsubseteq |b_1[\overline{X_1^v/a_1^v}]| \equiv |Y[\overline{X_1^v/a_1^v}]| \equiv |Y| \equiv Y$. So we can apply lemma 16 to get that $\vdash_{CRWL_{let}} e_2[X/Y] \equiv e_2[X/t_1] \twoheadrightarrow t$ implies $\vdash_{CRWL_{let}} e_2 \twoheadrightarrow t'$ with a derivation of the same size, for some $t' \in CTerm_\perp$ such that

$t'[X/Y] \equiv t$, to which we can apply the IH to get $e_2 \to^{rt\,*} e'$ for some $e' \in LExp$ such that $t' \sqsubseteq |e'|$, so we can do:

$$\begin{array}{l} let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e_2 \\ \to^{rt\,*} let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e' \ \text{ by IH} \end{array}$$

Now $t' \sqsubseteq |e'|$ implies $t'[X/Y] \sqsubseteq |e'|[X/Y]$ and so $t \equiv t'[X/Y] \sqsubseteq |e'|[X/Y]$. Besides we can prove that $(\overline{X_1^f} \cup \overline{X_1^v}) \cap var(t) = \emptyset$ in the same way we did in the **OR** case, but then $t \sqsubseteq |e'|[X/Y]$ implies that any occurrence in $|e'|[X/Y]$ of some $Z \in \overline{X_1^f} \cup \overline{X_1^v}$ corresponds to an occurrence of $\bot$ in t, in the same position. Hence $t \sqsubseteq |e'|[X/Y]$ implies $t \sqsubseteq |e'|[\overline{X_1^v/\bot}, \overline{X_1^f/\bot}, X/Y] \sqsubseteq |e'|[\overline{X_1^v/a_1^v}, \overline{X_1^f/\bot}, X/Y] \equiv |let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ e'|$.

iii) $b_1 \equiv \mathcal{C}[s_1, \ldots, s_n]$ for $\mathcal{C} \neq []$ a many hole c-context and $s_1, \ldots, s_n \in Exp$ the maximal (in the order of positions) subexpressions of b_1 which are function rooted or variables free in their contexts. Those free s_i are also not bound in their contexts, and so we can perform several (LetIn_1) or (LetIn_2) steps:

$$\begin{array}{l} let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e_2 \\ \equiv let\ \overline{X_1 = a_1}\ in\ let\ X = \mathcal{C}[s_1, \ldots, s_n]\ in\ e_2 \\ \to^{rt\,*} let\ \overline{X_1 = a_1}\ in\ let\ \overline{Y = s}\ in\ let\ X = \mathcal{C}[\overline{Y}]\ in\ e_2 \\ \to^{rt} let\ \overline{X_1 = a_1}\ in\ let\ \overline{Y = s}\ in\ e_2[X/\mathcal{C}[\overline{Y}]] \end{array}$$

the first step by $((\text{LetIn}_1 \mid \text{LetIn}_2)^*)$ and the second by (RBind). Now we can decompose $\overline{s}$ as $\overline{s} = \overline{s^f} \uplus \overline{s^v}$, where $\overline{s^f}$ contains those $s_i \in \overline{s}$ which are function rooted and $\overline{s^v}$ contains those which are free variables. Then $t_1 \sqsubseteq |\overline{b_1[X_1^v/a_1^v]}| \equiv |(\mathcal{C}[\overline{s}])[\overline{X_1^v/a_1^v}]| \sqsubseteq |(\mathcal{C}[\overline{s^v}, \overline{Y^f}])[\overline{X_1^v/a_1^v}]|$, by lemma 3, as $|\overline{s^f}| = \overline{\bot}$ because those are function rooted. But then $t_1 \sqsubseteq (\mathcal{C}[\overline{s^v}, \overline{Y^f}])[\overline{X_1^v/a_1^v}]$ by lemma 13, hence $\vdash_{CRWL_{let}} e_2[X/t_1] \twoheadrightarrow t$ implies $\vdash_{CRWL_{let}} e_2[X/(\mathcal{C}[\overline{s^v}, \overline{Y^f}])[\overline{X_1^v/a_1^v}]] \twoheadrightarrow t$ with a derivation of the same size or smaller.
Besides, $\overline{X_1^v} \cap FV(e_2) = \emptyset$, by the conditions in ($\text{Flat}_2$), so

$$\begin{array}{l} e_2[X/(\mathcal{C}[\overline{s^v}, \overline{Y^f}])[\overline{X_1^v/a_1^v}]] \equiv e_2[\overline{X_1^v/a_1^v}][X/(\mathcal{C}[\overline{s^v}, \overline{Y^f}])[\overline{X_1^v/a_1^v}]] \\ \equiv e_2[X/\mathcal{C}[\overline{s^v}, \overline{Y^f}]][\overline{X_1^v/a_1^v}] \end{array}$$

by the substitution lemma, as $X \notin (dom(\overline{[X_1^v/a_1^v]}) \cup vran(\overline{[X_1^v/a_1^v]}))$ by α-conversion. So we can apply lemma 16 to get that $\vdash_{CRWL_{let}} e_2[X/\mathcal{C}[\overline{s^v}, \overline{Y^f}]][\overline{X_1^v/a_1^v}] \equiv e_2[X/(\mathcal{C}[\overline{s^v}, \overline{Y^f}])[\overline{X_1^v/a_1^v}]] \twoheadrightarrow t$ implies that $\vdash_{CRWL_{let}} e_2[X/\mathcal{C}[\overline{s^v}, \overline{Y^f}]] \twoheadrightarrow t'$ with a derivation of the same size, for some $t' \in CTerm_\bot$ such that $t'[\overline{X_1^v/a_1^v}] \equiv t$.
Furthermore, as $\overline{Y}$ are fresh and linear then

$$\begin{array}{l} e_2[X/\mathcal{C}[\overline{s^v}, \overline{Y^f}]] \equiv e_2[X/(\mathcal{C}[\overline{Y}])[\overline{Y^v/s^v}]] \\ \equiv e_2[\overline{Y^v/s^v}][X/(\mathcal{C}[\overline{Y}])[\overline{Y^v/s^v}]] \\ \equiv e_2[X/\mathcal{C}[\overline{Y}]][\overline{Y^v/s^v}] \end{array}$$

because $\overline{Y} \cap FV(e_2) = \emptyset$ by the freshness of $\overline{Y}$, and by the substitution lemma, as $X \notin (dom([\overline{Y^v/s^v}]) \cup vran([\overline{Y^v/s^v}]))$. So we can apply lemma 16 to get that $\vdash_{CRWL_{let}} e_2[X/\mathcal{C}[\overline{Y}]][\overline{Y^v/s^v}] \equiv e_2[X/\mathcal{C}[\overline{s^v}, \overline{Y^f}]] \twoheadrightarrow t'$ implies $\vdash_{CRWL_{let}} e_2[X/\mathcal{C}[\overline{Y}]] \twoheadrightarrow t''$ with a derivation of the same size, for some $t'' \in CTerm_\bot$ such that $t''[\overline{Y^v/s^v}] \equiv t'$. We can apply the IH to that derivation to get:

$$\begin{array}{l} let\ \overline{X_1 = a_1}\ in\ let\ \overline{Y = s}\ in\ e_2[X/\mathcal{C}[\overline{Y}]] \\ \rightarrow^{rt\,*} let\ \overline{X_1 = a_1}\ in\ let\ \overline{Y = s}\ in\ e' \end{array}$$

Now, as $\overline{a_1^v} \cup \overline{s^v} \subseteq \mathcal{V}$ and $t'' \sqsubseteq |e'|$ by IH then $t''[\overline{Y^v/s^v}][\overline{X_1^v/a_1^v}] \sqsubseteq |e'|[\overline{Y^v/s^v}][\overline{X_1^v/a_1^v}]$ and so

$$t \equiv t'[\overline{X_1^v/a_1^v}] \equiv t''[\overline{Y^v/s^v}][\overline{X_1^v/a_1^v}] \sqsubseteq |e'|[\overline{Y^v/s^v}][\overline{X_1^v/a_1^v}]$$

Besides we can prove that $(\overline{X_1^f} \cup \overline{Y^f}) \cap var(t) = \emptyset$ in the same way we did in the **OR** case, but then $t \sqsubseteq |e'|[\overline{Y^v/s^v}][\overline{X_1^v/a_1^v}]$ implies that any occurrence in $|e'|[\overline{Y^v/s^v}][\overline{X_1^v/a_1^v}]$ of some $Z \in \overline{X_1^f} \cup \overline{Y^f}$ corresponds to an occurrence of $\bot$ in t, in the same position. Hence $t \sqsubseteq |e'|[\overline{Y^v/s^v}][\overline{X_1^v/a_1^v}]$ implies $t \sqsubseteq |e'|[\overline{Y^v/s^v}, \overline{Y^f/\ \bot}, \overline{X_1^v/a_1^v}, \overline{X_1^f/\ \bot}] \equiv |let\ \overline{X_1 = a_1}\ in\ let\ \overline{Y = s}\ in\ e'|$.

Proof (For Theorem 6).
a) Let $\mathcal{P}$ be a program, $e \in LExp$, $t \in CTerm$ and assume $\mathcal{P} \vdash e \twoheadrightarrow t$, which implies $\tau(\mathcal{P}) \vdash e \twoheadrightarrow t$. Lemma 2 ensures that $e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} e'$ for some $e' \in LExp$ such that $t \sqsubseteq |e'|$. Since t is total, $t = |e'|$, and $|e'|$ does not contain $\bot$ and therefore no function application. Using this fact to interpret the conlusion of the peeling lemma 14 applied to e', we obtain $e' \rightarrow^{rt\,*}_{\tau(\mathcal{P})} let\ \overline{Y = a}\ in\ b$ where all a must free variables and b must be a c-term, say t'. But then we have $e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} e' \rightarrow^{rt\,*}_{\tau(\mathcal{P})} let\ \overline{Y = a}\ in\ t'$. All bindings $Y = a$ corresponding to Y's not occurring in t' can disappear by **(Elim)**, and therefore we obtain $e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} let\ \overline{Y = a}\ in\ t'$ where all remaining Y are variables occurring in t'. Since the reductions made by the peeling lemma and the **(Elim)** rule do not change shells, we have $t = |e'| = |let\ \overline{Y = a}\ in\ t'| = t'[\overline{Y/a}]$, as desired. Notice that since all remaining Y occurred in t', all the a occur (free) in $t'[\overline{Y/a}]$, and therefore in t.
b) Notice simply that since t is ground, the set of bindings in the expression $let\ \overline{Y = a}\ in\ t'$ given by *a)* must be empty, and morevover $t' = t$.

Proof (For Theorem 7).
Let $\mathcal{P}$ be a program, $e \in LExp$, $t \in CTerm_\bot$.

a) The left to right implication is Lemma 2. For $\Leftarrow$, assume $e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} e'$, for some $|e'| \sqsupseteq t$. By Theorem 4, $[\![e']\!]^{\tau(\mathcal{P})} \subseteq [\![e]\!]^{\tau(\mathcal{P})}$. Now, since $|e'| \in [\![e']\!]^{\tau(\mathcal{P})}$, we have $|e'| \in [\![e]\!]^{\tau(\mathcal{P})}$. As $|e'| \sqsupseteq t$, a basic property of CRWL-semantics ensures that $t \in [\![e]\!]^{\tau(\mathcal{P})}$, which exactly means that $\mathcal{P} \vdash e \twoheadrightarrow t$.

b) Assume t is total. We have the equivalences $\mathcal{P} \vdash e \twoheadrightarrow t \ \Leftrightarrow\ e \rightarrow^{ct\,*}_{\mathcal{P}} t$ and $\mathcal{P} \vdash e \twoheadrightarrow t \ \Leftrightarrow\ \tau(\mathcal{P}) \vdash e \twoheadrightarrow t$ by Theor. 7 of [20] and Theor. 3 respectively. It remains to prove that $\tau(\mathcal{P}) \vdash e \twoheadrightarrow t \ \Leftrightarrow\ e \rightarrow^{rt\,*}_{\tau(\mathcal{P})}\ let\ \overline{Y = X}\ in\ t'$ for some $t' \in CTerm$ with $t'[\overline{Y/X}] \equiv t$

and $\overline{X} \subseteq FV(t)$. The implication $\Rightarrow$ is part *a)* of Theor. 6. For $\Leftarrow$ we reason as follows: assume $e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} \ let\ \overline{Y = X}\ in\ t'$ for some $t' \in CTerm$ with $t'[\overline{Y/X}] \equiv t$ and $\overline{X} \subseteq FV(t)$. Since $e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} \ let\ \overline{Y = X}\ in\ t'$, part *a)* of Theor. 4 ensures that $[\![let\ \overline{Y = X}\ in\ t']\!]^{\tau(\mathcal{P})} \subseteq [\![e]\!]^{\tau(\mathcal{P})}$. But it it easy to prove in the $CRWL_{let}$ framework that $[\![let\ \overline{Y = X}\ in\ t']\!]^{\tau(\mathcal{P})} = [\![t'[Y/X]]\!]^{\tau(\mathcal{P})}$. As $t'[\overline{Y/X}] \equiv t$, we have $[\![t]\!]^{\tau(\mathcal{P})} \subseteq [\![e]\!]^{\tau(\mathcal{P})}$. Finally, $t \in [\![t]\!]$ implies $t \in [\![e]\!]^{\tau(\mathcal{P})}$, which precisely means $\tau(\mathcal{P}) \vdash e \twoheadrightarrow t$, as desired.

c) It follows directly from *b)*, taking into account that the set $\overline{X}$ must be empty since t is ground and $\overline{X} \subseteq FV(t) = \emptyset$.