A Finite Simulation Method in a Non-Deterministic Call-by-Need Calculus with letrec, constructors and case

Manfred Schmidt-Schauss[1] and Elena Machkasova[2]

[1] Fachbereich Informatik und Mathematik,
Institut für Informatik, Johann Wolfgang Goethe-Universität,
Postfach 11 19 32, D-60054 Frankfurt, Germany,
{schauss}@ki.informatik.uni-frankfurt.de
[2] Division of Science and Mathematics,
University of Minnesota, Morris
Morris, MN 56267-2134, U.S.A
{elenam}@morris.umn.edu

Technical Report Frank-32

12. February 2008

Abstract. The paper proposes a variation of simulation for checking and proving contextual equivalence in a non-deterministic call-by-need lambda-calculus with constructors, case, seq, and a letrec with cyclic dependencies. It also proposes a novel method to prove its correctness. The calculus' semantics is based on a small-step rewrite semantics and on may-convergence. The cyclic nature of letrec bindings, as well as non-determinism, makes known approaches to prove that simulation implies contextual equivalence, such as Howe's proof technique, inapplicable in this setting. The basic technique for the simulation as well as the correctness proof is called pre-evaluation, which computes a set of answers for every closed expression. If simulation succeeds in finite computation depth, then it is guaranteed to show contextual preorder of expressions.

1 Introduction

The construction of compilers and the compilation of programs in higher level, expressive programming languages is an important process in computer science that is a highly sophisticated engineering task. Unfortunately there remains a gap between theory and practice. Usually compilers incorporating lots of complicated transformations and optimizations are built with only a partial knowledge about correctness issues. This gap increases with the number of features, in particular if higher-order functions, concurrency, store, and system- or user-interaction is permitted. Unfortunately, there is also no common agreement on a standard notion of correctness, in particular for languages with concurrency constructs.

We approach these issues for non-strict functional programming languages with concurrency by studying a call-by-need lambda-calculus with data structures and non-determinism that in addition has letrec allowing cyclic binding dependencies. This is rather close to Haskell [Pey03] and also concurrent Haskell [PGF96], but is also applicable to other lazy non-deterministic languages [Han96]. Our language L comes with a rewrite semantics (a small-step semantics) that is more appropriate to investigate non-determinism than a big-step semantics, since this presents interleaving reductions and atomic reductions explicitly. On top of the operational semantics we define as equivalence an observational semantics (also called contextual semantics) that can be seen as a maximal equivalence that is also a congruence with respect to the observation of successful termination. The congruence is maximal, since all expressions are identified that cannot be distinguished by observations.

This follows an approach pioneered in [Plo75] of considering two small-step relations in a calculus: a normal order reduction which represents evaluation of a term by some evaluation engine, such as an interpreter, and transformation steps performed by a compiler to optimize a program. The latter steps include the same kind of reductions as the normal order reduction, only performed in a context different from the one chosen by the interpreter. These steps may also be transformations different from any normal order steps. The goal is to prove contextual equivalence of the original and the transformed expressions, i.e. that any transformation step performed anywhere in a term does not change the term's terminational behavior.

Unfortunately the approach in [Plo75] cannot be applied to systems with cyclic dependencies (such as `letrec`) since the approach requires confluence of transformations which fails in such systems (see [AK96]). Some alternative approaches include restrictions on cyclic substitution [AK97] or considering terms up to infinite unwindings of cycles [AB02]. However, the former approach is undesirable for our work since it introduces extra restrictions and the latter one is quite complex even for the original system and it is unclear whether it is adaptable to non-deterministic call-by-need calculi.

Investigations of correctness (also called meaning-preservation) for a call-by-value system of mutually recursive components with applications to modules and linking were undertaken in [MT00,Mac02] where a proof method based on diagrams called *lift* and *project* was introduced. The diagram approach was later extended and generalized in [WPK03] in an abstract setting. Another approach based on multihole contexts was used for a call-by-name system of mutually recursive components in [Mac07]. However, the diagram-based and context-based approaches above require that all normal-order reductions preserve behavior of a term, which is not the case for (choice)-reduction. Contextual equivalence for non-deterministic call-by-need calculi was also investigated in [KSS98,MSC99,SSS07,SSSS08] using the method of forking and commuting diagrams, and in [MSC99] using abstract machine-reductions.

An important tool to prove contextual equivalence of concrete expressions is simulation-based (also called applicative simulation) since it allows to show contextual equivalence of expressions s, t based only on the analysis of the reductions of s, t, in contrast to the definition, which requires checking reduction in infinitely many contexts. This method was also used for other lambda calculi, see e.g [Abr90,How89,Gor99]. Application of simulation to a non-deterministic call-by-need calculus was done in [Man04], where Howe's [How89,How96] proof technique is extended to call-by-need by using an intermediate approximation calculus. This was generalized to a calculus also including constructors in [SSM07]. Unfortunately, these proof methods based on the approach of Howe appear not to be adaptable to call-by-need non-deterministic calculi with letrec, since the cyclic dependencies cannot be treated using the proof method.

In this paper we propose a method of pre-evaluation of closed expressions to answers as a substitute for simulation in these calculi, and then to compare expressions based on the set of answers. In contrast to the method of Howe, our method allows only simulations of finite depth, whereas common simulation is defined also for infinite depth. An answer is either an abstraction or an expression built from constructors, Ω, and abstractions, like partial lists. Pre-evaluation means to look at the set of answers that can be reached from a closed expression t by applying reduction rules from the calculus, where also certain non-normal-order reductions are permitted, and also a rule that stops further reduction by approximating sub-expressions with an Ω. We present in an informal way an illustrating example that will be made more precise later. Consider the two (non-convertible) expressions s, t where $s = \texttt{repeat True}$, which will evaluate to a nonending list that only contains the element `True`, where t is the recursively defined expression $t = \texttt{choice} \perp (\texttt{Cons True}\ t)$. The latter will evaluate, depending on the choices to $\perp$, $(\texttt{Cons True} \perp)$, $(\texttt{Cons True} (\texttt{Cons True} \perp))$, ... Since the observational equivalence is a maximal equivalence, it is not possible to distinguish these two expressions using contexts and may-convergence. Our simulation method permits to compute the equivalence only on the basis of the (approximative) answers that can be derived from each one, and also shows that they are contextually equivalent.

The method to prove this result for an extended call-by-need lambda calculus L with letrec requires several steps. The first one is to investigate the correctness of several reductions and transformations in L. Note that the normal-order reduction in the language L treats chains of variable-variable bindings as transparent for several reductions, which is indispensable, since otherwise the correctness proofs are not possible. A context lemma and standardization of reductions are proved. We also have to analyze length-properties of normal-order reduction sequences. The second step is a transfer from L to the calculus L_S, which is a simpler calculus with the same contextual equivalence, but with simpler reduction rules. In particular, variable-variable bindings are now opaque and have to be treated locally. The calculus L_S employs an eager copy of abstractions and flat values. The third step is to define the computation of answers from a closed expression, and to prove criteria for contextual equivalence on the basis of the answer sets. We also provide a method to analyze contextual equivalence and preorder of answers, which is necessary for comparing the answer-sets. Here we

solved a particular hard case for these kind of calculi: we can compare abstractions based on applying them to all closed answer or to Ω.
As an application of this technique, we show that `choice`, seen as a binary infix-operator, has useful algebraic properties, such as idempotency, commutativity and associativity, for all expressions, including open ones.

2 The Calculus L

In this part we introduce the calculus L, i.e. its syntax, the operational semantics and the program equivalence based on contextual equivalence. In Subsection 2.1 we introduce the syntax of L, followed by Subsection 2.2 where we define the normal order reduction for L. Based on the notion of termination we introduce contextual equivalence in Subsection 2.2.2. The calculus is the same as the one considered in [SSSS04] and extension by `choice` of the one in [SSSS08].

2.1 Syntax and Reductions of the Functional Core Language L

We define the calculus L consisting of a language $\mathcal{L}(\mathrm{L})$ and its reduction rules, presented in this section, and the normal order reduction strategy and contextual equivalence, presented in Section 2.2. If no confusion arises, we also speak of the language L.
In our language, L, expressions have the following syntax: there are finitely many constants, called constructors. The set of constructors is partitioned into (non-empty) types. For every type T there are finitely many constructors, say $\#(T)$. We denote the constructors as $c_{T,i}, i = 1, \ldots, \#(T)$. Every constructor has an arity $\mathrm{ar}(c_{T,i}) \geq 0$.
The syntax for expressions E, case alternatives Alt and patterns Pat is as follows:

$$\begin{aligned}
E &::= V \mid (c\ E_1 \ldots E_{\mathrm{ar}(c)}) \mid (\mathtt{seq}\ E_1\ E_2) \mid (\mathtt{case}_T\ E\ Alt_1 \ldots Alt_{\#(T)}) \mid (E_1\ E_2) \\
&\quad\ (\mathtt{choice}\ E_1\ E_2) \mid (\lambda\ V.E) \mid (\mathtt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \mathtt{in}\ E) \\
Alt &::= (Pat\ \rightarrow\ E) \\
Pat &::= (c\ V_1 \ldots V_{\mathrm{ar}(c)})
\end{aligned}$$

where E, E_i are expressions, V, V_i are variables and where c denotes a constructor. Within each individual pattern, variables are not repeated. In a `case`-expression of the form $(\mathtt{case}_T \ldots)$, for every constructor $c_{T,i}, i = 1, \ldots, \#(T)$ of type T, there is exactly one alternative with a pattern of the form $(c_{T,i}\ y_1 \ldots y_n)$ where $n = \mathrm{ar}(c_{T,i})$. We assign the names *application*, *abstraction*, *constructor application*, `seq`-*expression*, `case`-*expression*, or `letrec`-*expression* to the expressions $(E_1\ E_2)$, $(\lambda V.E)$, $(c\ E_1 \ldots E_{\mathrm{ar}(c)})$, $(\mathtt{seq}\ E_1\ E_2)$, $(\mathtt{case}_T\ E\ Alt_1 \ldots Alt_{\#(T)})$, $(\mathtt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \mathtt{in}\ E)$, respectively. We use the notation $\lambda V_1, V_2, \ldots, V_n.E$ as an abbreviation for $\lambda V_1.\lambda V_2. \ldots \lambda V_n.E$.
The constructs `case`, `seq`, `choice` and the constructors $c_{T,i}$ can only occur in special syntactic constructions. Thus expressions where `case`, `seq`, `choice` or a constructor is applied to the wrong number of arguments are not allowed.

The structure `letrec` obeys the following conditions: The variables V_i in the bindings are all distinct. We also assume that the bindings in `letrec` are commutative, i.e. `letrec`s with bindings interchanged are considered to be syntactically equivalent. `letrec` is recursive: I.e., the scope of x_j in (`letrec` $x_1 = E_1, \ldots, x_j = E_j, \ldots$ `in` E) is E and all expressions E_i. This fixes the notions of closed, open expressions and α-renamings. Free and bound variables in expressions are defined using the usual conventions. Variable binding primitives are λ, `letrec`, patterns, and the scope of variables bound in a `letrec` are all the expressions occurring in it. The set of free variables in an expression t is denoted as $FV(t)$. For simplicity we use the distinct variable convention: I.e., all bound variables in expressions are assumed to be distinct, and free variables are distinct from bound variables. The reduction rules are assumed to implicitly rename bound variables in the result by α-renaming if necessary to obey this convention. Note that this is only necessary for the copy rule (cp) (see figure 1). We follow the convention by omitting parentheses in nested applications: $(s_1 \ldots s_n)$ denotes $(\ldots(s_1\ s_2) \ldots s_n)$ provided s_1 is an expression. The set of closed L-expressions is denoted as L^0.
To abbreviate the notation, we will sometimes use ($\texttt{case}_T$ E *alts*) instead of ($\texttt{case}_T$ E $alt_1 \ldots alt_{\#(T)}$). Sometimes we abbreviate the notation of `letrec`-expression (`letrec` $x_1 = E_1, \ldots, x_n = E_n$ `in` E), as (`letrec` *Env* `in` E), where $Env \equiv \{x_1 = E_1, \ldots, x_n = E_n\}$. This will also be used freely for parts of the bindings. The notation $\{x_{g(i)} = s_{h(i)}\}_{i=m}^{n}$ is used for the chain $x_{g(m)} = s_{h(m)}, x_{g(m+1)} = s_{h(m+1)}, \ldots, x_{g(n)} = s_{h(n)}$ of bindings where $g, h : \mathbb{N} \to \mathbb{N}$, e.g., $\{x_i = s_{i-1}\}_{i=m}^{n}$ means the bindings $x_m = s_{m-1}, x_{m+1} = s_m, \ldots x_n = s_{n-1}$. We assume that `letrec`-expressions have at least one binding. The set $\{x_1, \ldots, x_n\}$ of variables that are bound by the `letrec`-environment $Env = \{x_1 = s_1, \ldots, x_n = s_n\}$ is denoted as $LV(Env)$. In examples we will use `:` as an infix binary list-constructor, and `Nil` as the constant constructor for lists. We will write $(c_i\ \overrightarrow{z})$ as shorthand for the constructor application $(c_i\ z_1\ \ldots\ z_{\mathrm{ar}(c_i)})$. In the following we define different context classes and contexts. To visually distinguish context classes from individual contexts, we use different text styles.

Definition 2.1. *The class $\mathcal{C}$ of all* contexts *is defined as the set of expressions C from* L*, where the symbol $[\cdot]$, the* hole*, is a predefined context, treated as an atomic expression, such that $[\cdot]$ occurs exactly once in C.*
Given a term t and a context C, we will write $C[t]$ for the expression constructed from C by plugging t into the hole, i.e, by replacing $[\cdot]$ in C by t, where this replacement is meant syntactically, i.e., a variable capture is permitted.

Definition 2.2. *A* value *is either an abstraction, or a constructor application. We denote values by the letters v, w.*

The reduction rules in Definition 2.3, i.e. in figures 1 and 2 are defined more liberally than necessary for the normal order reduction, in order to permit an easy use as transformations.

Definition 2.3 (Reduction Rules of the Calculus L**).** *The (base) reduction rules for the calculus and language* L *are defined in figures 1 and 2, where the* *labels S, V, T are to be ignored in this subsection, but will be used in subsection 2.2. The reduction rules can be applied in any context. The union of (llet-in) and (llet-e) is called (llet), the union of the rules (choice-l) and (choice-r) is called (choice), the union of (case-c), (case-in), (case-e) is called (case), the union of (seq-c), (seq-in), (seq-e) is called (seq), the union of (cp-in) and (cp-e) is called (cp), and the union of (llet), (lcase), (lapp), (lseq) is called (lll).*

Reductions (and transformations) are denoted using an arrow with super and/or subscripts: e.g. $\xrightarrow{llet}$. To explicitly state the context in which a particular reduction is executed we annotate the reduction arrow with the context in which the reduction takes place. If no confusion arises, we omit the context at the arrow.

The redex *of a reduction is the term as given on the left side of a reduction rule. We will also speak of the* inner redex*, which is the focused* `case`*-expression for (case)-reductions, the focused* `seq`*-expression for (seq)-reductions, or the variable position which is replaced by a (cp), depending on the applied rule. Otherwise it is the same as the redex.*

Transitive closure of reductions is denoted by a $+$, reflexive transitive closure by a $*$. E.g. $\xrightarrow{*}$ is the reflexive, transitive closure of $\rightarrow$. We also use ? to denote one or zero occurrences of a step. If necessary, we attach more information to the arrow.

Note that the reduction rules generate only syntactically correct expressions, since reductions, transformations and contexts are appropriately defined.

2.2 Normal Order Reduction and Contextual Equivalence

We define and explain the final components of the calculus L.

2.2.1 Normal Order Reduction The normal order reduction strategy of the calculus L is a call-by-need strategy, which is a call-by-name strategy adapted to sharing. The following labeling algorithm will detect the position to which a reduction rule will be applied according to normal order. It uses the labels: S, T, V, W, where T means reduction of the top term, S means reduction of a subterm, and V, W mark already visited subexpressions, where W at a variable indicates that the variable must not be replaced by a (cp)-reduction. Note that the labeling algorithm does not look into S-labeled `letrec`-expressions. For a term s the labeling algorithm starts with s^T, where no other subexpression in s is labeled and proceeds until no more labeling is possible or until a fail occurs.

(lbeta)	$((\lambda x.s)^S\ r) \to (\mathtt{letrec}\ x = r\ \mathtt{in}\ s)$
(cp-in)	$(\mathtt{letrec}\ x_1 = v^S, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}\ \ \mathtt{in}\ C[x_m^V])$
	$\to (\mathtt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}\ \ \mathtt{in}\ C[v])$
	where v is an abstraction
(cp-e)	$(\mathtt{letrec}\ x_1 = v^S, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}, y = C[x_m^V]\ \mathtt{in}\ r)$
	$\to (\mathtt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}, y = C[v]\ \mathtt{in}\ r)$
	where v is an abstraction
(llet-in)	$(\mathtt{letrec}\ \mathit{Env}_1\ \mathtt{in}\ (\mathtt{letrec}\ \mathit{Env}_2\ \mathtt{in}\ r)^S)$
	$\to (\mathtt{letrec}\ \mathit{Env}_1, \mathit{Env}_2\ \mathtt{in}\ r)$
(llet-e)	$(\mathtt{letrec}\ \mathit{Env}_1, x = (\mathtt{letrec}\ \mathit{Env}_2\ \mathtt{in}\ s_x)^S\ \mathtt{in}\ r)$
	$\to (\mathtt{letrec}\ \mathit{Env}_1, \mathit{Env}_2, x = s_x\ \mathtt{in}\ r)$
(lapp)	$((\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ t)^S\ s) \to (\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ (t\ s))$
(lcase)	$(\mathtt{case}_T\ (\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ t)^S\ \mathit{alts}) \to (\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ (\mathtt{case}_T\ t\ \mathit{alts}))$
(seq-c)	$(\mathtt{seq}\ v^S\ t) \to t$ if v is a value
(seq-in)	$(\mathtt{letrec}\ x_1 = v^S, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}\ \mathtt{in}\ C[(\mathtt{seq}\ x_m^V\ t)])$
	$\to (\mathtt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}\ \mathtt{in}\ C[t])$
	if v is a value
(seq-e)	$(\mathtt{letrec}\ x_1 = v^S, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}, y = C[(\mathtt{seq}\ x_m^V\ t)]\ \mathtt{in}\ r)$
	$\to (\mathtt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}, y = C[t]\ \mathtt{in}\ r)$
	if v is a value
(lseq)	$(\mathtt{seq}\ (\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ s)^S\ t) \to (\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ (\mathtt{seq}\ s\ t))$
(choice-l)	$(\mathtt{choice}\ s\ t)^{S \vee T} \to s$
(choice-r)	$(\mathtt{choice}\ s\ t)^{S \vee T} \to t$

Fig. 1. Reduction rules, part a

The rules of the labeling algorithm are as follows, where the rules can be applied in any context.

$(\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ t)^T$	$\to (\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ t^S)^V$
$(s\ t)^{S \vee T}$	$\to (s^S\ t)^V$
$(\mathtt{seq}\ s\ t)^{S \vee T}$	$\to (\mathtt{seq}\ s^S\ t)^V$
$(\mathtt{case}_T\ s\ \mathit{alts})^{S \vee T}$	$\to (\mathtt{case}_T\ s^S\ \mathit{alts})^V$
$(\mathtt{letrec}\ x = s, \mathit{Env}\ \mathtt{in}\ C[x^S])$	$\to (\mathtt{letrec}\ x = s^S, \mathit{Env}\ \mathtt{in}\ C[x^V])$
$(\mathtt{letrec}\ x = s, y = C[x^S], \mathit{Env}\ \mathtt{in}\ t)$	$\to (\mathtt{letrec}\ x = s^S, y = C[x^V], \mathit{Env}\ \mathtt{in}\ t)$
	if $C[x] \neq x$
$(\mathtt{letrec}\ x = s, y = x^S, \mathit{Env}\ \mathtt{in}\ t)$	$\to (\mathtt{letrec}\ x = s^S, y = x^W, \mathit{Env}\ \mathtt{in}\ t)$

The notation $S \vee T$ stands for S or T. If a rule tries to label a subexpression already labeled V or W, then a loop has been detected and the algorithm stops with fail. Otherwise, if the labeling algorithm terminates, since it is no longer possible to apply a rule, then we say the termination is successful, and a potential normal order redex is found, which can only be the direct superterm of the S-marked subexpression. It is possible that there is no normal order reduction: in this case either the evaluation is already finished, or it is a dynamically detected error (like a type-error), or it is prevented by the loop-check above.

(case-c) $(\texttt{case}_T\ (c_i\ \overrightarrow{t}\,)^S\ \ldots((c_i\ \overrightarrow{y}\,) \to t)\ldots) \to (\texttt{letrec}\ \{y_i = t_i\}_{i=1}^n\ \texttt{in}\ t)$
where $n = \text{ar}(c_i) \geq 1$

(case-c) $(\texttt{case}_T\ c_i^S\ \ldots\ (c_i \to t)\ldots) \to t$ if $\text{ar}(c_i) = 0$

(case-in) $\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{t}\,)^S, \{x_i = x_{i-1}\}_{i=2}^m, Env$
$\texttt{in}\ C[\texttt{case}_T\ x_m^V\ \ldots((c_i\ \overrightarrow{z}\,)\ldots \to t)\ldots]$
$\to \texttt{letrec}\ x_1 = (c_i\ \overrightarrow{y}\,), \{y_i = t_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=2}^m, Env$
$\texttt{in}\ C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^n\ \texttt{in}\ t)]$
where $n = \text{ar}(c_i) \geq 1$ and y_i are fresh variables

(case-in) $\texttt{letrec}\ x_1 = c_i^S, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ C[\texttt{case}_T\ x_m^V\ \ldots\ (c_i \to t)\ldots]$
$\to \texttt{letrec}\ x_1 = c_i, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ \ C[t]$
if $\text{ar}(c_i) = 0$

(case-e) $\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{t}\,)^S, \{x_i = x_{i-1}\}_{i=2}^m,$
$u = C[\texttt{case}_T\ x_m^V\ \ldots((c_i\ \overrightarrow{z}\,) \to r_1)\ldots], Env$
$\texttt{in}\ r_2$
$\to \texttt{letrec}\ x_1 = (c_i\ \overrightarrow{y}\,), \{y_i = t_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=2}^m,$
$u = C[(\texttt{letrec}\ z_1 = y_1, \ldots, z_n = y_n\ \texttt{in}\ r_1)], Env$
$\texttt{in}\ r_2$
where $n = \text{ar}(c_i) \geq 1$ and y_i are fresh variables

(case-e) $\texttt{letrec}\ x_1 = c_i^S, \{x_i = x_{i-1}\}_{i=2}^m, u = C[\texttt{case}_T\ x_m^V\ \ldots\ (c_i \to r_1)\ldots], Env$
$\texttt{in}\ r_2$
$\to \texttt{letrec}\ x_1 = c_i, \{x_i = x_{i-1}\}_{i=2}^m \ldots, u = C[r_1], Env\ \texttt{in}\ r_2$
if $\text{ar}(c_i) = 0$

Fig. 2. Reduction rules, part b

We define reduction contexts and weak reduction contexts:

Definition 2.4. *A reduction context R is any context, such that its hole will be labeled with S or T by the labeling algorithm. A* weak reduction context, *R^-, is a reduction context, where the hole is not within a* `letrec`*-expression.*
A maximal reduction context *of an expression s is a reduction context R with $R[s'] = s$, such that the labeling algorithm applied to s will label the subexpression s' with S or T and will then stop with success.*

For example the maximal reduction context of $(\texttt{letrec}\ x_2 = \lambda x.x, x_1 = x_2\ x_1\ \texttt{in}\ x_1)$ is $(\texttt{letrec}\ x_2 = [\cdot], x_1 = x_2\ x_1\ \texttt{in}\ x_1)$, in contrast to the non-maximal reduction context $(\texttt{letrec}\ x_2 = \lambda x.x, x_1 = x_2\ x_1\ \texttt{in}\ [\cdot])$.

Definition 2.5 (Normal Order Reduction of L**).** *Let t be an expression. Then a single normal order reduction step $\xrightarrow{no}$ is defined by first applying the labeling algorithm to t, and if the labeling algorithm terminates successfully, then one of the rules in figures 1 and 2 has to be applied, if possible, where the labels S, V, T must match the labels in the expression t.*
The normal order redex *is defined as the subexpression as given on the left side of a reduction rule. This includes the* `letrec`*-expression that is mentioned in the reduction rules, for example in (cp-e).*

The inner normal order redex *is the following subterm in t: it is the* `case`*-expression that is replaced by a letrec-expression for (case)-reductions, the* `seq`*-expression that is replaced by a value for (seq)-reductions, or the V-labeled variable position which is replaced by a value for (cp)-reductions. Otherwise it is the same as the normal order redex.*

The normal order reduction implies that `seq` behaves like a function strict in its first argument, and that the `case`-construct is strict in its first argument, i.e. these rules can only be applied if the corresponding argument is a value or if the argument is a variable bound to a value.
We are interested in normal order reduction sequences, i.e. $\xrightarrow{no,*}$-reductions, and mainly those that end with a generalized value, also called weak head normal form.

Definition 2.6. *A* weak head normal form (WHNF) *is one of the cases:*

1. *A value v.*
2. *A term of the form* (`letrec` *Env* `in` v)*, where v is a value.*
3. *A term of the form* (`letrec` $x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^{m}, Env$ `in` x_m)*.*

If the value v in the WHNF t is an abstraction, we call t a functional WHNF (FWHNF)*, otherwise, if v is a constructor application, we call t a* constructor WHNF (CWHNF)*.*

Lemma 2.7. *For every term t: if t has a normal order redex, then the normal order redex, the inner normal order redex are unique, and for all rule-applications, with the exception of (choice), the normal order reduction is unique.*

Definition 2.8. *A normal order reduction sequence is called a* (normal-order) evaluation *if the last term is a WHNF. Otherwise, i.e. if the normal order reduction sequence is non-terminating, or if the last term is not a WHNF, but has no normal order reduction, then we say that it is a* failing *normal order reduction sequence.*
For a term t, we write $t\downarrow$ iff there is an evaluation starting from t. We call this the evaluation of t *and denote it as $nor(t)$. If $t\downarrow$, we also say that t is* converging *(or* terminating*). Otherwise, if there is no evaluation of t, we write $t\Uparrow$. A specific representative of the non-converging expressions is Ω, which can be defined as*

$$\Omega := (\lambda z.(z\ z))\ (\lambda x.(x\ x)).$$

Note that there are useful open terms t that might not have an evaluation, e.g. x is such a term. Note also that there are (closed) terms t that are neither WHNFs nor have a normal order redex. For example ($\texttt{case}_T$ $(\lambda x.x)$ *alts*) or `((cons 1 2) 3)`, where `cons` is a constructor of arity 2. These terms are bot-terms and could be considered as violating type conditions. Consider the closed "cyclic term" (`letrec` $x = x$ `in` x). A reduction context for this term is (`letrec` $x = [\cdot]$ `in` x). Obviously, there is no normal order reduction defined for this term, hence also no evaluation of t.

As an example, we show the first normal order reduction steps of an evaluation of $\Omega = (\lambda z.(z\ z))\ (\lambda x.(x\ x))$:

$(\lambda z.(z\ z))\ (\lambda x.(x\ x)) \xrightarrow{no,lbeta} (\texttt{letrec}\ z = \lambda x.(x\ x)\ \texttt{in}\ (z\ z)) \xrightarrow{no,cp} (\texttt{letrec}\ z = \lambda x.(x\ x)\ \texttt{in}\ ((\lambda x'.(x'\ x'))\ z)) \xrightarrow{no,lbeta} (\texttt{letrec}\ z = \lambda x.(x\ x)\ \texttt{in}\ (\texttt{letrec}\ x_1 = z\ \texttt{in}\ (x_1\ x_1))) \xrightarrow{no,llet} (\texttt{letrec}\ z = \lambda x.(x\ x), x_1 = z\ \texttt{in}\ (x_1\ x_1)) \longrightarrow \ldots$.

2.2.2 Contextual Equivalence The semantic foundation of our calculus L is the equality of expressions defined by contextual equivalence. We define contextual equivalence w.r.t. evaluations.

Definition 2.9 (contextual preorder and equivalence). *Let s, t be terms. Then:*

$$\begin{array}{l} s \leq_c t \textit{ iff } \forall C[\cdot]:\quad C[s]{\downarrow} \Rightarrow C[t]{\downarrow} \\ s \sim_c t \textit{ iff } s \leq_c t \wedge t \leq_c s \end{array}$$

Note that we permit contexts $C[]$ such that $C[s]$ may be an open term.
By standard arguments, we see that $\leq_c$ is a precongruence and that $\sim_c$ is a congruence, where a *precongruence* $\leq_c$ is a preorder on expressions, such that $s \leq_c t \Rightarrow C[s] \leq_c C[t]$ for all contexts C, and a *congruence* is a precongruence that is also an equivalence relation.

3 Contextual Equivalence of L

This part provides methods and tools used in the correctness proof of transformations the pre-evaluation and the finite simulation method. In Subsection 3.1 and Section 4 we develop the proof methods using a context lemma and overlap diagrams in order to show that $s \sim_c t$ for all reduction rules $s \to t$ except for (choice). A set of extra transformation rules is defined and investigated in Section 4.
We introduce extra transformation rules below and two kinds of surface contexts.

Definition 3.1 (Surface Context Classes). *A* surface context *is a context where the hole is not contained in an abstraction. The class of surface contexts is denoted as $\mathcal{S}$. An* application surface context *is a surface context where the hole is neither contained in an abstraction nor in an alternative of a case-expression, denoted as $\mathcal{AS}$. A* weak application surface context *is an application surface context where the hole is in addition not contained in a* `letrec`*-expression, denoted as $\mathcal{W}$.*
For a context C its main depth *is defined as the depth of its hole. With $C_{(i)}$ we denote a context of main depth i.*

Note that every reduction context is also a surface context and an application surface context, and that a weak reduction context is also a weak application surface context.

3.1 Context Lemma

The Context Lemma restricts the criterion for contextual equivalence to reduction contexts. This restriction is of great value in proving the conservation of contextual equivalence by certain reductions.

Lemma 3.2 (Context Lemma). *Let s, t be terms. If for all reduction contexts R: $(R[s]{\downarrow} \Rightarrow R[t]{\downarrow})$, then $\forall C : (C[s]{\downarrow} \Rightarrow C[t]{\downarrow})$; i.e. $s \leq_c t$.*

The proof of the context lemma 3.2 follows the structure of a similar proof in [SSS06]. It uses multihole contexts (multicontexts) and derives the context lemma from a stronger statement (see Lemma 3.4). The presence of (choice) reduction in the calculus does not change the proof.

Definition 3.3 (Multicontext). *A multicontext is the result of replacing n subterms (where $n \geq 0$) in a well-formed term by a special symbol $\cdot$ called a hole. Multicontexts are denoted as $C[\cdot_1, \ldots, \cdot_n]$. An n-hole context $C[\cdot_1, \ldots, \cdot_n]$ can be filled with n terms $s_1, \ldots, s_n$, resulting in a term $C[s_1, \ldots, s_n]$. Free variables of the terms s_i may be captured in the process. We may also fill only some holes in a multicontext, leaving the rest as holes. The result of filling an n-hole multicontext with $m \leq n$ terms is a multicontext with $n - m$ holes.*

A regular one-hole context is a partial case of a multicontext. Note that, unlike terms, contexts are not subject to α-renaming.
Following the approach in [SSS06], we prove the claim that is more general than the desired context lemma 3.2:

Lemma 3.4. *For all expressions $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$, where $n \geq 0$, the following holds: If for all i s.t. $1 \leq i \leq n$ and for all reduction contexts R: if $(R[s_i]{\downarrow} \Rightarrow R[t_i]{\downarrow})$ then $C[s_1, \ldots, s_n]{\downarrow} \Rightarrow C[t_1, \ldots, t_n]{\downarrow}$ for all n-hole multicontexts C.*

Proof. The proof is completely analogous to that in [SSS06]. □

The context lemma (Lemma 3.2) is a partial case of the above lemma.
The context lemma has as consequence the following useful corollary:

Corollary 3.5. *Let s, t be terms. If for all surface contexts $\mathcal{S}$: $(\mathcal{S}[s]{\downarrow} \Rightarrow \mathcal{S}[t]{\downarrow})$, then $\forall C : (C[s]{\downarrow} \Rightarrow C[t]{\downarrow})$; i.e. $s \leq_c t$.*

4 Correctness of Transformations

We say that a transformation $\rightsquigarrow$ on terms is *correct*, if $s \rightsquigarrow t$ implies $s \sim_c t$ for all terms s, t. In the following we will use the base reductions also as transformations (ignoring the labels S, V, T). The goal is to prove the following correctness claim for all transformations.

> All the reductions (viewed as transformations) in the base calculus L with the exception of (choice) maintain contextual equivalence. I.e. whenever $t \xrightarrow{a} t'$, with $a \in \{\text{cp, lll, case, seq, lbeta}\}$, then $t \sim_c t'$.

Note that the correctness proof for $a \in \{\text{cp, lll, seq, lbeta}\}$ can be done in a standard way (see e.g. the appendix of [SSSS08]) using the context lemma and complete sets of diagrams. However, the correctness proof for (case) requires further tools, in particular the extra transformations defined below in the next subsection.

4.1 Extra Transformations

In addition to transformations based on the L-calculus rules applied in a non-reduction context, we define several other transformations given in Figure 3. Several transformations have two or more forms. For instance, the variable elimination rule has forms (ve1) and (ve2). We use the rule abbreviation without a number (such as (ve)) to refer to the union of all forms of that transformation so (ve) stands for a transformation that is either (ve1) or (ve2).

We use (abs1) and (ve) as a part of the correctness proof of the calculus-based transformations. Additionally we use (cpcx), (abs2), and (ve) for showing that the calculus L is equivalent to a simpler calculus L_S. The transformation (ucp) is used in Section 7 and (gc) is needed for showing correctness of (ucp).

(ve1)	$\texttt{letrec}\ x = y, x_1 = t_1, \ldots, x_n = t_n\ \texttt{in}\ r \rightarrow \texttt{letrec}\ x_1 = t'_1, \ldots, x_n = t'_n\ \texttt{in}\ r'$
	where $t'_i = t_i[y/x], r' = r[y/x], n \geq 1$ and if $x \neq y$
(ve2)	$\texttt{letrec}\ x = y\ \texttt{in}\ s \rightarrow s[y/x]$ if $x \neq y$
(abs1)	$(\texttt{letrec}\ x = c\ \overrightarrow{t}, \mathit{Env}\ \ \texttt{in}\ s) \rightarrow (\texttt{letrec}\ x = c\ \overrightarrow{x}, \{x_i = t_i\}_{i=1}^{\mathrm{ar}(c)}, \mathit{Env}\ \ \texttt{in}\ s)$
	where $\mathrm{ar}(c) \geq 1$ and for $1 \leq i \leq \mathrm{ar}(c)$:$x_i$ is fresh
(abs2)	$(c\, t_1 \ldots t_n) \rightarrow \texttt{letrec}\ x_1 = t_1, \ldots, x_n = t_n\ \texttt{in}\ (c\, x_1 \ldots x_n)$
	where at least one of t_i is not a variable and $n \geq 1$
(cpcx-in)	$(\texttt{letrec}\ x = c\ \overrightarrow{t}, \mathit{Env}\ \texttt{in}\ C[x])$
	$\rightarrow (\texttt{letrec}\ x = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^{\mathrm{ar}(c)}, \mathit{Env}\ \texttt{in}\ C[c\ \overrightarrow{y}])$
(cpcx-e)	$(\texttt{letrec}\ x = c\ \overrightarrow{t}, z = C[x], \mathit{Env}\ \texttt{in}\ t)$
	$\rightarrow (\texttt{letrec}\ x = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^{\mathrm{ar}(c)}, z = C[c\ \overrightarrow{y}], \mathit{Env}\ \texttt{in}\ t)$
(gc1)	$(\texttt{letrec}\ \{x_i = s_i\}_{i=1}^{n}, \mathit{Env}\ \texttt{in}\ t) \rightarrow (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ t)$
	if for all $i : x_i$ does not occur in Env nor in t
(gc2)	$(\texttt{letrec}\ \{x_i = s_i\}_{i=1}^{n}\ \texttt{in}\ t) \rightarrow t$
	if for all $i : x_i$ does not occur in t
(ucp1)	$(\texttt{letrec}\ \mathit{Env}, x = t\ \texttt{in}\ S[x]) \rightarrow (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ S[t])$
(ucp2)	$(\texttt{letrec}\ \mathit{Env}, x = t, y = S[x]\ \texttt{in}\ r) \rightarrow (\texttt{letrec}\ \mathit{Env}, y = S[t]\ \texttt{in}\ r)$
(ucp3)	$(\texttt{letrec}\ x = t\ \texttt{in}\ S[x]) \rightarrow S[t]$
	where in the (ucp)-rules, x has at most one occurrence in $S[x]$ and no occurrence in Env, t, r; and S is a surface context

Fig. 3. Transformations in L calculus

4.2 Correctness of (lbeta), (lapp), (lcase), (lseq), (seq-c), (case-c)

Proposition 4.1. *The following rules produce correct transformations in any context: (lbeta), (lapp), (lcase), (lseq), (seq-c), and (case-c).*

Proof. The argument is completely analogous to that in [SSS06,SSSS08]: if any of these reductions appears in a reduction context, then it must be a normal order reduction. Since the normal-order reduction is unique for these reductions, the context lemma 3.2 is applicable and implies correctness. □

The missing reductions are (llet), (seq-in), (seq-e), (case-c), (case-e), and (cp). Their correctness is proven in Section 4.3 using overlap diagrams.

4.3 The Correctness Proof Method Using Diagrams.

Correctness proofs in this section are based on overlap diagrams. Overlap diagrams summarize cases when a normal-order step and a transformation step originate in the same term. These diagrams are used to transform an evaluation of a non-transformed term into an evaluation of this term after a given transformation, and vice-versa.
There are two basic kinds of overlap diagrams: *commuting* and *forking* diagrams. In both kinds of diagrams a normal order step (below denoted by $\xrightarrow{no,a}$) and a transformation step or a non-normal order reduction in a surface context (below denoted by $\xrightarrow{b}$) are given and both diagrams show how the given pair can be replaced by another sequence connecting the same given terms which may have non-normal-order steps in forward or backward direction, relative to the direction of the given normal order step.
The two kinds of diagrams differ by directions of the given steps as follows. In a commuting diagram a given pair of reduction steps has the form $s_1 \xrightarrow{b} s_2 \xrightarrow{no,a} s_3$ where $\xrightarrow{b}$ is not normal-order and the diagram is completed by a stating that there exists a sequence of steps from s_1 to s_3. In a forking diagram a given pair of reductions is of the form $s_1 \xleftarrow{b} s_2 \xrightarrow{no,a} s_3$ where $\xrightarrow{b}$ is not normal-order and the diagram is completed by stating the existence of a sequence from s_1 to s_3.
Note that by the context lemma 3.2 it is sufficient to show that any transformation in a reduction context preserves convergence since this would imply that it preserves convergence in any context. However, it is more convenient to prove a more general statement by showing that a given transformation in a *surface context* preserves convergence (recall that a reduction context is a partial case of a surface context; see Corollary 3.5). Hence we assume that the given non-normal-order step or transformation is in a surface context. All the resulting non-normal-order steps are also in a surface context unless stated otherwise. We omit the S mark for a reduction in a surface context since it is always implied.
In a graphical representation of overlap diagrams we often combine commuting and forking diagrams since they both can be read off of the same diagram. If we show commuting and forking diagrams separately then we use a dashed line to show steps whose existence is claimed by the diagram.

As an example illustrating all the notations and conventions, consider the following overlap diagram:

$$\begin{tikzcd}
\cdot \arrow[rrr, "case{-}in"] \arrow[d, "no{,}case"'] & & & \cdot \arrow[d, "no{,}case"] \\
\cdot \arrow[r, "case{-}in"] & \cdot \arrow[r, "ve1{,}*"] & \cdot & \cdot \arrow[l, "ve1{,}*"']
\end{tikzcd}$$

The diagram shows interactions between a normal order (case) reduction, denoted *no,case*, and a transformation, i.e. a non-normal-order, denoted *case-in*, step. The diagram combines a commuting and a forking diagram, representing the following two claims:

– Commuting diagram: in some cases given the reduction $s_1 \xrightarrow{S,case-in} s_2 \xrightarrow{no,case} s_3$ there exist terms t_1, t_2, t_3 such that $s_1 \xrightarrow{no,case} t_1 \xrightarrow{S,case-in} t_2 \xleftarrow{S,ve1,*} t_3 \xrightarrow{S,ve1,*} s_3$, where $*$ denotes 0 or more occurrences of a step.
– Forking diagram: in some cases given the reduction $s_1 \xleftarrow{S,case-in} s_2 \xrightarrow{no,case}$ there exist terms t_1, t_2, t_3 such that $s_1 \xrightarrow{no,case} t_1 \xrightarrow{S,ve1,*} t_2 \xleftarrow{S,ve1,*} t_3 \xleftarrow{S,case-in} s_3$.

Note that each of the diagrams takes place only for some, but not for all cases of the given reduction pair. A *complete* set of diagrams is a set that includes all possible interaction cases of a given transformation step and a normal order step. The complete set of diagrams for a (case-in) transformation in a surface context is given in Subsection 4.9.
In addition to overlap diagrams correctness proofs use the lemma stated below.

Lemma 4.2. *If $s \xrightarrow{S,a} t$, where S denotes a surface context and a is any reduction step or a transformation step, then for any reduction context R there exists a surface context S' s.t. $R[s] \xrightarrow{S',a} R[t]$.*

Proof. Since S context is any context that does not reach under a λ and R does not reach under a λ either, $S' = R[S]$ is a surface context. $\square$

In several proofs we use the following measure:

Definition 4.3. *Ordered sequence measure M is an ordered set of sequences of non-negative integers defined as follows: the elements of the set are finite sequences of non-negative integers $s_1, \dots, s_n$ s.t. for all $i < n$ $s_i \geq s_{i+1}$, $n \geq 0$. The sequences are ordered lexicographically.*

The lexicographic ordering works as follows: $(1,1) < (2,1)$, $(1,1) < (2)$, $(1,1) < (1,1,1)$. The empty sequence is less than any other sequence. This measure is equivalent to the *multiset order* (for a total base ordering) defined in [BN98]. However, this representation of this measure is more intuitive for our proofs than the multiset order.

Lemma 4.4. *For the measure M defined in Definition 4.3 the following is true:*

– *The lexicographic ordering is a total ordering on M.*
– *Given a sequence s, there is no infinite descending chain $s > s_1 > s_2 > \dots$*

Proof. Part 1 is trivial since a lexicographic ordering of strings whose elements belong to an ordered set is a total order. The second part follows from well-foundedness of a multiset ordering with well-founded base ordering [BN98]. □

4.4 Correctness of (cp)

Following the approach in [SSS06,SSSS08], we distinguish between the following versions of (cp):

– (cpS) - the variable occurrence that gets replaced is in a surface context.
– (cpd) - the variable occurrence that gets replaced is not in a surface context.

Note that the distinction is based on the target variable occurrence, not on the context in which the (cp) redex appears. For instance, in the following example the (cpd) redex appears in a surface context:

$$\texttt{letrec}\ x = v\ \texttt{in}\ \lambda y.x$$

For (cpS) we have the following complete sets of diagrams.

$$\begin{tikzcd}
\cdot \arrow[r, "cpS"] \arrow[d, "no{,}a"'] & \cdot \arrow[d, "no{,}a"] \\
\cdot \arrow[r, "cpS"] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "cpS"] \arrow[d, "no{,}a"'] & \cdot \arrow[dl, "no{,}a"] \\
\cdot &
\end{tikzcd}$$

Here (cpS) happens in a surface context and a is any normal order reduction. In the first diagram the resulting (cpS) step may also be a normal order reduction. The diagrams for (cpd) are as follows:

$$\begin{tikzcd}
\cdot \arrow[r, "cpd"] \arrow[d, "no{,}a"'] & \cdot \arrow[d, "no{,}a"] \\
\cdot \arrow[r, "cpd"] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "cpd"] \arrow[d, "no{,}a"'] & \cdot \arrow[dl, "no{,}a"] \\
\cdot &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "cpd"] \arrow[d, "no{,}lbeta"'] & \cdot \arrow[d, "no{,}lbeta"] \\
\cdot \arrow[r, "cpS"] & \cdot
\end{tikzcd}$$

$$\begin{tikzcd}
\cdot \arrow[rr, "cpd"] \arrow[d, "no{,}cp"'] & & \cdot \arrow[d, "no{,}cp"] \\
\cdot \arrow[r, "cpd"] & \cdot \arrow[r, "cpd"] & \cdot
\end{tikzcd}$$

The following is easy to show by cases of WHNF.

Lemma 4.5. *If $s \xrightarrow{cp} t$ is not a normal order step then s is a WHNF if and only if t is a WHNF.*

Proposition 4.6. *(cp) is correct, i.e. if $s \xrightarrow{cp} t$ then $s \sim_c t$.*

Proof. The proof is analogous to that in [SSS06]. The added (choice) reduction does not change the diagrams. □

4.5 Correctness of (lll)

Diagrams for (lll) rules are as follows:

$$\begin{tikzcd}
\cdot \arrow[r, "lll"] \arrow[d, "{no,a}"'] & \cdot \arrow[d, "{no,a}"] \\
\cdot \arrow[r, "lll"] & \cdot
\end{tikzcd}
\quad
\begin{tikzcd}
\cdot \arrow[r, "lll"] \arrow[d, "{no,a}"'] & \cdot \arrow[dl, "{no,a}"] \\
\cdot &
\end{tikzcd}
\quad
\begin{tikzcd}
\cdot \arrow[r, "lll"] \arrow[d, "{no,lll,+}"'] & \cdot \arrow[d, "{no,lll,+}"] \\
\cdot \arrow[r, "lll"] & \cdot
\end{tikzcd}
\quad
\begin{tikzcd}
\cdot \arrow[r, "lll"] \arrow[d, "{no,lll,+}"'] & \cdot \arrow[dl, "{no,lll,+}"] \\
\cdot &
\end{tikzcd}$$

Proposition 4.7. *(lll) is correct, i.e. if $s \xrightarrow{lll} t$ then $s \sim_c t$.*

Proof. The proof is analogous to that in [SSS06]. The added (choice) reduction does not change the diagrams. □

Lemma 4.8. *(lll) transformation is terminating, i.e. there is no infinite transformation sequence that consists only of (lll) steps.*

Proof. The proof is analogous to that in [SSS06]. □

4.6 Correctness of (seq)

The diagrams for (seq) are very similar to those of (cp). Note that according to the convention in Section 4.3 all (seq) reductions take place in a surface context.

$$\begin{tikzcd}
\cdot \arrow[r, "seq"] \arrow[d, "{no,a}"'] & \cdot \arrow[d, "{no,a}"] \\
\cdot \arrow[r, "seq"] & \cdot
\end{tikzcd}
\quad
\begin{tikzcd}
\cdot \arrow[r, "seq"] \arrow[d, "{no,a}"'] & \cdot \arrow[dl, "{no,a}"] \\
\cdot &
\end{tikzcd}
\quad
\begin{tikzcd}
\cdot \arrow[rr, "seq"] \arrow[d, "{no,cp}"'] & & \cdot \arrow[d, "{no,cp}"] \\
\cdot \arrow[r, "seq"] & \cdot \arrow[r, "seq"] & \cdot
\end{tikzcd}$$

In the first diagram the resulting (seq) step may or may not be a normal order reduction. The last diagram takes place when the (seq) redex gets duplicated by a (cp) reduction, as shown in the following example:

$$\begin{array}{ll}
\texttt{letrec}\ x = \lambda z.(\texttt{seq}\ y\ t), y = v\ \texttt{in}\ x & \xrightarrow{seq-e} \\
\texttt{letrec}\ x = \lambda z.t, y = v\ \texttt{in}\ x & \xrightarrow{no,cp} \\
\texttt{letrec}\ x = \lambda z.t, y = v\ \texttt{in}\ \lambda z.t &
\end{array}$$

$$\begin{array}{ll}
\texttt{letrec}\ x = \lambda z.(\texttt{seq}\ y\ t), y = v\ \texttt{in}\ x & \xrightarrow{no,cp} \\
\texttt{letrec}\ x = \lambda z.(\texttt{seq}\ y\ t), y = v\ \texttt{in}\ \lambda z.(\texttt{seq}\ y\ t) & \xrightarrow{seq-e} \\
\texttt{letrec}\ x = \lambda z.t, y = v\ \texttt{in}\ \lambda z.(\texttt{seq}\ y\ t) & \xrightarrow{seq-in} \\
\texttt{letrec}\ x = \lambda z.t, y = v\ \texttt{in}\ \lambda z.t &
\end{array}$$

Lemma 4.9. *If $s \xrightarrow{seq,*} t$ and none of the reductions are normal order then s is a WHNF if and only if t is a WHNF.*

Proof. By a simple case analysis on definition of WHNF. □

Proposition 4.10. *(seq) is correct, i.e. if $s \xrightarrow{seq} t$ then $s \sim_c t$.*

Proof. The following diagram follows from the commuting and forking diagrams by induction on the number of (seq).

$$\begin{tikzcd}
\cdot \arrow[rr, "seq{,}*"] \arrow[d, "no{,}a"'] & & \cdot \arrow[d, "no{,}a"] \\
\cdot \arrow[r, "no{,}seq{,}*"] & \cdot \arrow[r, "seq{,}*"] & \cdot
\end{tikzcd}$$

We use (I,seq) to denote a non-normal-order reduction. A reduction $\xrightarrow{seq,S,*}$ is a mixture of (no,seq) and (I,S,seq)-reductions. A pair $\xrightarrow{I,S,seq} . \xrightarrow{no,seq}$ can be switched using the complete set of commuting diagrams for (seq) to either $\xrightarrow{no,seq}$, $\xrightarrow{no,seq} . \xrightarrow{I,S,seq}$, or $\xrightarrow{no,seq} . \xrightarrow{no,seq}$. Now an induction shows that the switching will produce a sequence $\xrightarrow{no,seq,*} . \xrightarrow{I,S,seq,*}$. The corresponding commuting and forking properties are as follows, assuming that the (seq) sequence not marked as a normal order sequence does not have any normal order steps:

- **Commuting property:** If $s_1 \xrightarrow{seq,*} s_2 \xrightarrow{no,a} s_3$ then there exist t_1, t_2 s.t. $s_1 \xrightarrow{no,a} t_1 \xrightarrow{no,seq,*} t_2 \xrightarrow{seq,*} s_3$.
- **Forking property:** If $s_1 \xleftarrow{seq,*} s_2 \xrightarrow{no,a} s_3$ then there exist t_1, t_2 s.t. $s_1 \xrightarrow{no,a} t_1 \xleftarrow{seq,*} t_2 \xleftarrow{no,seq,*} s_3$.

Applying induction on the number of given normal order steps to the commuting diagram above, we conclude that if $s \xrightarrow{seq} t$ in a surface context and for a reduction context R we have $R[t] \xrightarrow{no,*} t'$, where t' is a WHNF, then there exist s', s'' s.t. $R[s] \xrightarrow{no,*} s' \xrightarrow{no,seq,*} s'' \xrightarrow{seq,*} t'$ where none of the (seq) steps in the sequence $s'' \xrightarrow{seq,*} t'$ are normal order steps. By Lemma 4.9 s'' is a WHNF.
Similarly we can use the forking diagram to prove the other direction of the lemma. Suppose $s \xrightarrow{seq} t$ in a surface context and for a reduction context R we have $R[s] \xrightarrow{no,*} s'$, where s' is a WHNF. Then by the forking diagram above there exist t', t'' s.t. $s' \xrightarrow{no,seq,*} t' \xrightarrow{seq,*} t''$. Since s' is a WHNF, the subsequence $s' \xrightarrow{no,seq,*} t'$ is empty and $s' \xrightarrow{seq,*} t''$, where all (seq) steps are non-normal-order. Then by Lemma 4.9 is a WHNF.
We have proven that if $s \xrightarrow{seq} t$ then for any reduction context R $R[s]{\downarrow}$ if and only if $R[t]{\downarrow}$. The claim of the lemma follows from the Context Lemma 3.2. □

4.7 Correctness of (ve)

The *variable elimination* transformation (ve) is used in the correctness proof of (case). It has two versions, (ve1) and (ve2), defined in Figure 3.

The forking diagrams for (ve) are as follows:

$$\begin{tikzcd}
\cdot \arrow[r, "ve"] \arrow[d, "no{,}a"'] & \cdot \arrow[d, dashed, "no{,}a"] \\
\cdot \arrow[r, dashed, "ve"] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "ve"] \arrow[d, "no{,}a_1"'] & \cdot \arrow[dl, dashed, "no{,}a_1"] \\
\cdot &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "ve2"] \arrow[d, "no{,}lll"'] & \cdot \\
\cdot \arrow[ur, dashed, "ve"'] &
\end{tikzcd}$$

Here a is any normal order reduction and $a_1 \in \{$seq, choice, case$\}$. Note that if the given (ve) step is a (ve1) then the resulting step may not be a (ve2).

In the following example for the first diagram the second (ve) step happens in a different `letrec` than the first one.

$$\begin{array}{lr}
\texttt{letrec}\ y = v\ \texttt{in}\ (\texttt{letrec}\ x = y, x_1 = t_1, \ldots, x_n = t_n\ \texttt{in}\ (\texttt{seq}\ x\ s)) & \xrightarrow{ve1} \\
\texttt{letrec}\ y = v\ \texttt{in}\ (\texttt{letrec}\ x_1 = t_1[y/x], \ldots, x_n = t_n[y/x]\ \texttt{in}\ (\texttt{seq}\ x\ s[y/x])) & \xrightarrow{no,llet-in} \\
\texttt{letrec}\ y = v, x_1 = t_1[y/x], \ldots, x_n = t_n[y/x]\ \texttt{in}\ (\texttt{seq}\ x\ s[y/x]) &
\end{array}$$

$$\begin{array}{lr}
\texttt{letrec}\ y = v\ \texttt{in}\ (\texttt{letrec}\ x = y, Env\ \texttt{in}\ (\texttt{seq}\ x\ s)) & \xrightarrow{no,llet-in} \\
\texttt{letrec}\ y = v, x = y, Env\ \texttt{in}\ (\texttt{seq}\ x\ s) & \xrightarrow{ve1} \\
\texttt{letrec}\ y = v, x_1 = t_1[y/x], \ldots, x_n = t_n[y/x]\ \texttt{in}\ (\texttt{seq}\ x\ s[y/x]) &
\end{array}$$

In this case the inner `letrec` will be marked with S by the unwinding algorithm. Since the unwinding does not descend into S-marked `letrec`, there is no possibility for a normal-order (cp), (seq), or (case) reduction of the outer `letrec` with y as a target. For instance, in the above example the normal order reduction is (llet-in), and not (cp) or (seq). Thus the binding $x = y$ in s will be lifted to the `letrec` where y is bound before a reduction under the inner `letrec` takes place.

An example for the last diagram is when a (ve2) step removes a `letrec`. Such examples can be constructed for any (lll) step, we show it for (lapp):

$$\begin{array}{l}
(\texttt{letrec}\ x = y\ \texttt{in}\ t)\ s \ \xrightarrow{ve2}\ (t'\ s') \\
(\texttt{letrec}\ x = y\ \texttt{in}\ t)\ s \ \xrightarrow{no,lapp}\ \texttt{letrec}\ x = y\ \texttt{in}\ (t\ s) \ \xrightarrow{ve2}\ (t'\ s')
\end{array}$$

where $t' = t[y/x], s' = s[y/x]$.

Another example for the last diagram shows that (ve2) can be transformed into (ve1):

$$\begin{array}{lr}
\texttt{letrec}\ x = (\texttt{letrec}\ y = z\ \texttt{in}\ s)\ \texttt{in}\ x & \xrightarrow{ve2} \\
\texttt{letrec}\ x = s[z/y]\ \texttt{in}\ x &
\end{array}$$

$$\begin{array}{lr}
\texttt{letrec}\ x = (\texttt{letrec}\ y = z\ \texttt{in}\ s)\ \texttt{in}\ x & \xrightarrow{no,llet-e} \\
\texttt{letrec}\ x = s, y = z\ \texttt{in}\ x & \xrightarrow{ve1} \\
\texttt{letrec}\ x = s[z/y]\ \texttt{in}\ x &
\end{array}$$

Lemma 4.11. *Commuting diagrams for (ve) are as follows:*

$$\begin{tikzcd}
\cdot \arrow[r, "ve"] \arrow[d, dashed, "no{,}a"'] & \cdot \arrow[d, "no{,}a"] \\
\cdot \arrow[r, dashed, "ve"] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "ve"] \arrow[d, dashed, "no{,}a"'] & \cdot \arrow[dl, "no{,}a"] \\
\cdot &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "ve2"] \arrow[d, dashed, "no{,}lll"'] & \cdot \arrow[dd, "no{,}a"] \\
\cdot \arrow[d, dashed, "no{,}a"'] & \\
\cdot \arrow[r, dashed, "ve"'] & \cdot
\end{tikzcd}$$

Proof. The first two diagrams are the same as the forking diagrams. The last commuting diagram is a combination of the reverse forms of the first and the last forking diagrams: the last diagram turns a given sequence $\xrightarrow{ve2}\xrightarrow{no,a}$ into $\xrightarrow{no,lll}\xrightarrow{ve}\xrightarrow{no,a}$, and then the first diagram converts the latter sequence into $\xrightarrow{no,lll}\xrightarrow{no,a}\xrightarrow{ve}$. Note that the last diagram may be applied only once since the (lll) step is applied to the `letrec` used in the (ve) step, and there is only one such `letrec` (see the example below). Also note that the reverse form of the second forking diagram cannot be combined with the last one since the second diagram only happens when the (ve) step is inside an argument of a `seq`, `choice`, or `case`, but this means that the `letrec` eliminated by (ve2) is not in a normal order reduction context. □

As an illustration of the last commuting diagram in the above lemma consider:

$$\begin{array}{lr}
\texttt{letrec}\ x = (\texttt{letrec}\ y = z\ \texttt{in}\ v)\ \texttt{in}\ x & \xrightarrow{ve2} \\
\texttt{letrec}\ x = v[z/y]\ \texttt{in}\ x & \xrightarrow{no,cp} \\
\texttt{letrec}\ x = v[z/y]\ \texttt{in}\ v[z/y] &
\end{array}$$

$$\begin{array}{lr}
\texttt{letrec}\ x = (\texttt{letrec}\ y = z\ \texttt{in}\ v)\ \texttt{in}\ x & \xrightarrow{no,lll} \\
\texttt{letrec}\ x = v, y = z\ \texttt{in}\ x & \xrightarrow{no,cp} \\
\texttt{letrec}\ x = v, y = z\ \texttt{in}\ v & \xrightarrow{ve} \\
\texttt{letrec}\ x = v[z/y]\ \texttt{in}\ v[z/y] &
\end{array}$$

Lemma 4.12.
- *If $s \xrightarrow{ve1} t$ then s is a WHNF if and only if t is a WHNF.*
- *If $s \xrightarrow{ve2} t$ then: if s is a WHNF then t is a WHNF; if t is a WHNF then either s is a WHNF or there is a WHNF s' s.t. $s \xrightarrow{no,lll} s' \xrightarrow{ve} t$.*

Proof.

- (ve1) does not change the shape of a term. In the last case of WHNF if the last binding in the chain is removed by (ve1) then the variable in the body of the `letrec` will be renamed accordingly.
- The following are non-trivial cases for (ve2) (we omit some that are very similar to the ones given below). If Env stands for $x_1 = t_1, \ldots, x_n = t_n$ then $Env[y/x]$ denotes $x_1 = t_1[y/x], \ldots, x_n = t_n[y/x]$. Similarly we use $\overrightarrow{t}[y/x]$ to denote the result of replacing x by y in each of t_i in $\overrightarrow{t}$. For simplicity we

do not show variable chains when showing the cases below.

$\texttt{letrec}\ x = y\ \texttt{in}\ (\texttt{letrec}\ Env\ \texttt{in}\ v)\ \xrightarrow{ve2}\ \texttt{letrec}\ Env[y/x]\ \texttt{in}\ v[y/x]$
$\texttt{letrec}\ x = y\ \texttt{in}\ (\texttt{letrec}\ Env\ \texttt{in}\ v)\ \xrightarrow{no,lll}\ \texttt{letrec}\ x = y, Env\ \texttt{in}\ v\ \xrightarrow{ve2}$
$\texttt{letrec}\ Env[y/x]\ \texttt{in}\ v[y/x]$

$\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{letrec}\ x = y\ \texttt{in}\ v)\ \xrightarrow{ve2}\ \texttt{letrec}\ Env\ \texttt{in}\ v[y/x]$
$\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{letrec}\ x = y\ \texttt{in}\ v)\ \xrightarrow{no,lll}\ \texttt{letrec}\ Env, x = y\ \texttt{in}\ v\ \xrightarrow{ve}$
$\texttt{letrec}\ Env\ \texttt{in}\ v[y/x]$

$\texttt{letrec}\ x = y\ \texttt{in}\ (\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), Env\ \texttt{in}\ x_1)\ \xrightarrow{ve2}$
$\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}[y/x]), Env[y/x]\ \texttt{in}\ x_1$
$\texttt{letrec}\ x = y\ \texttt{in}\ (\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), Env\ \texttt{in}\ x_1)\ \xrightarrow{lll}$
$\texttt{letrec}\ x = y, x_1 = (c\ \overrightarrow{t}), Env\ \texttt{in}\ x_1\ \xrightarrow{ve}$
$\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}[y/x]), Env[y/x]\ \texttt{in}\ x_1$

$\texttt{letrec}\ x_1 = (\texttt{letrec}\ x = y\ \texttt{in}\ (c\ \overrightarrow{t})), Env\ \texttt{in}\ x_1\ \xrightarrow{ve2}$
$\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}[y/x]), Env[y/x]\ \texttt{in}\ x_1$
$\texttt{letrec}\ x_1 = (\texttt{letrec}\ x = y\ \texttt{in}\ (c\ \overrightarrow{t})), Env\ \texttt{in}\ x_1\ \xrightarrow{lll}$
$\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), x = y, Env\ \texttt{in}\ x_1\ \xrightarrow{ve}$
$\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}[y/x]), Env[y/x]\ \texttt{in}\ x_1$
Note that $Env[y/x] = Env$

$\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), Env\ \texttt{in}\ (\texttt{letrec}\ y = x_1\ \texttt{in}\ y)\ \xrightarrow{ve2}$
$\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), Env\ \texttt{in}\ x_1$
$\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), Env\ \texttt{in}\ (\texttt{letrec}\ y = x_1\ \texttt{in}\ y)\ \xrightarrow{lll}$
$\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), Env, y = x_1\ \texttt{in}\ y\ \xrightarrow{ve}$
$\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), Env\ \texttt{in}\ x_1$

□

Proposition 4.13. *The transformation (ve) preserves contextual equivalence, i.e. if* $s \xrightarrow{ve} t$ *then* $s \sim_c t$.

Proof. By the Context Lemma 3.2 it is sufficient to consider (ve) in a surface context. Commuting and forking diagrams summarize such cases.
Let $s \xrightarrow{ve} t$ where the step takes place in a surface context. Suppose $R[s] \xrightarrow{no,*} s'$ where s' is a WHNF. By Lemma 4.2 $R[s] \xrightarrow{ve} R[t]$, where the step also takes place in a surface context. We show that we can transform the sequence $R[t] \xleftarrow{ve} R[s] \xrightarrow{no,*} s'$ into a sequence $R[t] \xrightarrow{no,*} t' \xleftarrow{ve?} s'$, where ? denotes one or zero occurrences of a step and t' is a WHNF.
We use the forking diagrams to transform the given sequence by replacing a segment matching a diagram by the result of the diagram. The diagrams are applied

at the leftmost matching occurrence in the sequence. Each of the applications of forking diagrams reduces the number of normal order reduction steps to the right of the (ve) step in the sequence. If there is no (ve) steps, the measure is 0 and the process stops. Note that there may not be more than one (ve) step since none of the forking diagrams duplicate (ve) steps.
Below are the cases of the forking diagrams:

– $\xleftarrow{ve} \xrightarrow{no} \rightsquigarrow \xrightarrow{no} \xleftarrow{ve}$ - the number of normal order steps to the right of (ve) is reduced by 1.
– $\xleftarrow{ve} \xrightarrow{no} \rightsquigarrow \xrightarrow{no}$ - the (ve) step is eliminated, the resulting sequence is normal order.
– $\xleftarrow{ve2} \xrightarrow{no,lll} \rightsquigarrow \xleftarrow{ve}$ - the number of normal order steps to the right of (ve) is reduced by 1.

The procedure stops when the sequence is as follows: $R[t] \xrightarrow{no,*} t' \xleftarrow{ve?} s'$. Since s' is a WHNF, by Lemma 4.12 so is t'. Thus we have shown that $s \leq_c t$.
For the other direction of the lemma let $R[s] \xrightarrow{ve} R[t]$ and suppose $R[t]$ evaluates to WHNF. We show that, given a normal order sequence reduction $R[t] \xrightarrow{no,*} t'$, where t' is a WHNF, we can transform the sequence $R[s] \xrightarrow{ve} R[t] \xrightarrow{no,*} t'$ into a sequence $R[s] \xrightarrow{no,*} s' \xrightarrow{ve,*} t'$ using commuting diagrams in Lemma 4.11. We use the same measure as we used for the forking diagrams. The cases of commuting diagrams are as follows:

– $\xrightarrow{ve} \xrightarrow{no} \rightsquigarrow \xrightarrow{no} \xrightarrow{ve}$ - the number of normal order steps to the right of (ve) is reduced by 1.
– $\xrightarrow{ve} \xrightarrow{no} \rightsquigarrow \xrightarrow{no}$ - the (ve) step is eliminated, the resulting sequence is normal order.
– $\xrightarrow{ve} \xrightarrow{no} \rightsquigarrow \xrightarrow{lll} \xrightarrow{no} \xrightarrow{ve}$ - the number of normal order steps to the right of (ve) is reduced by 1.

The procedure stops when the sequence is as follows: $R[s] \xrightarrow{no,*} s' \xrightarrow{ve?} t'$, where t' is a WHNF. Then by Lemma 4.12 either s' is a WHNF or there exists s'' s.t. $s' \xrightarrow{lll} s'' \xrightarrow{ve} t'$. Thus $R[s]$ reduces to a WHNF by a normal order reduction, and therefore $t \leq_c s$.
Combining the two direction of the proof and applying the context lemma, we conclude that $s \sim_c t$. □

The following lemma is used in further proofs:

Lemma 4.14. *1. If $s_1 \xleftrightarrow{ve1,*} s_2 \xrightarrow{no} s_3$ then there exists s_4 s.t. $s_1 \xrightarrow{no} s_4 \xleftrightarrow{ve1,*} s_3$.*
2. If $s_1 \xrightarrow{no} s_2 \xleftrightarrow{ve1,*} s_3$ then there exists s_4 s.t. $s_1 \xleftrightarrow{ve1,*} s_4 \xrightarrow{no} s_3$.
The properties are summarized by the following diagrams:

$$\begin{tikzcd}
\cdot \arrow[r, leftrightarrow, "ve1{,}*"] \arrow[d, "no{,}a"'] & \cdot \arrow[d, dashed, "no{,}a"] \\
\cdot \arrow[r, leftrightarrow, dashed, "ve1{,}*"] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, leftrightarrow, "ve1{,}*"] \arrow[d, dashed, "no{,}a"'] & \cdot \arrow[d, "no{,}a"] \\
\cdot \arrow[r, leftrightarrow, dashed, "ve1{,}*"] & \cdot
\end{tikzcd}$$

Proof. the non-trivial diagrams (the last forking diagram and the last commuting diagram) happen only when the given step is a (ve2), and a (ve1) may not turn into a (ve2), therefore only the first and the second diagrams (both forking and commuting ones) take place for (ve1). A straightforward induction on the number of (ve1) steps gives the desired properties. □

4.8 Correctness of (abs1)

The (abs1) transformation has the following commuting and forking diagrams:

$$\begin{tikzcd} \cdot \arrow[r, "abs1"] \arrow[d, "{no,a}"'] & \cdot \arrow[d, "{no,a}"] \\ \cdot \arrow[r, "abs1"] & \cdot \end{tikzcd} \qquad \begin{tikzcd} \cdot \arrow[r, "abs1"] \arrow[d, "{no,a}"'] & \cdot \arrow[dl, "{no,a}"] \\ \cdot & \end{tikzcd} \qquad \begin{tikzcd} \cdot \arrow[r, "abs1"] \arrow[d, "{no,case}"'] & \cdot \arrow[d, "{no,case}"] \\ \cdot & \cdot \arrow[l, "{ve1,*}"] \end{tikzcd}$$

The last diagram corresponds to interactions between (abs1) and (case). A typical example is as follows. Note that the number of (ve) steps in each sequence always equals to the number of arguments of the constructor.

$\texttt{letrec}\ x = (c\ t_1\ t_2)\ \texttt{in case}\ x(c\ y_1\ y_2) \to s \xrightarrow{case}$
$\texttt{letrec}\ x = (c\ z_1\ z_2), z_1 = t_1, z_2 = t_2\ \texttt{in letrec}\ y_1 = z_1, y_2 = z_2\ \texttt{in}\ s$

$\texttt{letrec}\ x = (c\ t_1\ t_2)\ \texttt{in case}\ x(c\ y_1\ y_2) \to s \xrightarrow{abs1}$
$\texttt{letrec}\ x = (c\ z_1\ z_2), z_1 = t_1, z_2 = t_2\ \texttt{in case}\ x(c\ y_1\ y_2) \to s \xrightarrow{case}$
$\texttt{letrec}\ x = (c\ q_1\ q_2), q_1 = z_1, q_2 = z_2, z_1 = t_1, z_2 = t_2$
$\ \texttt{in letrec}\ y_1 = q_1, y_2 = q_2\ \texttt{in}\ s \xrightarrow{ve1,*}$
$\texttt{letrec}\ x = (c\ z_1\ z_2), z_1 = t_1, z_2 = t_2\ \texttt{in letrec}\ y_1 = z_1, y_2 = z_2\ \texttt{in}\ s$

Note that (case-c) does not have `letrec` as a part of its redex, thus it does not interact with (abs1).

Lemma 4.15. *If $s \xrightarrow{abs1} t$ then s is a WHNF if and only if t is a WHNF.*

Proof. By cases of definition of WHNF. □

The correctness proof for (abs1) requires diagrams for both (abs1) and (ve1) since (abs1) is transformed into (ve1) in diagram 3 above. We use the diagrams for (ve1) given in Lemma 4.14.

Proposition 4.16. *If $s \xrightarrow{abs1} t$ then $s \sim_c t$.*

Proof. By Context Lemma 3.2 it is sufficient to consider cases when (abs1) takes place in a surface context which are given in the commuting and forking diagrams. Commuting and forking diagrams for (abs1) and (ve1) are given in Sections 4.7 and earlier in this section. We use the diagrams in Lemma 4.14 for (ve1). The diagrams used in this proof are as follows:

– Commuting diagrams:

$$\begin{array}{ll}
\xrightarrow{abs1}\ \xrightarrow{no,a} & \leadsto\ \xrightarrow{no,a}\ \xrightarrow{abs1} \\
\xrightarrow{abs1}\ \xrightarrow{no,a} & \leadsto\ \xrightarrow{no,a} \\
\xrightarrow{abs1}\ \xrightarrow{no,case} & \leadsto\ \xrightarrow{no,case}\ \xleftarrow{ve1,*} \\
\xleftrightarrow{ve1,*}\ \xrightarrow{no,a} & \leadsto\ \xrightarrow{no,a}\ \xleftrightarrow{ve1,*}
\end{array}$$

– Forking diagrams:

$$\begin{array}{ll}
\xleftarrow{abs1}\ \xrightarrow{no,a} & \leadsto\ \xrightarrow{no,a}\ \xleftarrow{abs1} \\
\xleftarrow{abs1}\ \xrightarrow{no,a} & \leadsto\ \xrightarrow{no,a} \\
\xleftarrow{abs1}\ \xrightarrow{no,case} & \leadsto\ \xrightarrow{no,case}\ \xrightarrow{ve1,*} \\
\xleftrightarrow{ve1,*}\ \xrightarrow{no,a} & \leadsto\ \xrightarrow{no,a}\ \xleftrightarrow{ve1,*}
\end{array}$$

Let $s \xrightarrow{abs1} t$. We would like to show that for any reduction context R if $R[s]\downarrow$ then $R[t]\downarrow$, and also that if $R[t]\downarrow$ then $R[s]\downarrow$.
We use the diagrams above to convert a given evaluation of $R[s]$ into an evaluation of $R[t]$ and the other way around. We use Lemma 4.2: if $s \xrightarrow{S,abs1} t$ then $R[s] \xrightarrow{S',abs1} R[t]$ (where S and S' are surface contexts) so we can apply the diagrams to the reduction sequences for $R[s]$ and $R[t]$.
None of the commuting and forking diagrams change the number of normal order steps. The diagrams also guarantee that at any given moment the reduction sequence has either a single (abs1) step or a single (ve1) segment (a sequence of (ve1) steps in either direction) or no transformation steps at all. A sequence of (ve1) steps acts as a single step since the entire sequence can be switched with a normal order step at once by Lemma 4.14. Thus the proof is trivial by induction on the number of steps to the right of the (abs1) step or the (ve1) segment.
By Lemmas 4.12 and 4.15 weak head normal forms are preserved by both (abs1) and (ve1) so the base case is valid. □

4.9 Correctness of a Surface Version of (case-in), (case-e)

Recall that the (case-c) transformation is correct by Lemma 4.1. There are two kinds of (case-in) and (case-e) reductions in a surface context:

– The "surface" case, i.e. when the context C in the definition of the rule is a surface context as well; this version is denoted as (caseS). For example,

$$\texttt{letrec}\ x = (c\ t_1\ t_2)\ \texttt{in}\ (\texttt{seq}\ y\ (\texttt{case}\ x\ ((c\ z_1\ z_2) \rightarrow z_1\ z_2)))$$

– The "deep" case when, although the entire (case) reduction takes place in a surface context, the context C in the definition of the rule is not a surface context; this version is denoted as (caseD). For example,

$$\texttt{letrec}\ x = (c\ t_1\ t_2)\ \texttt{in}\ \lambda y.(\texttt{case}\ x\ ((c\ z_1\ z_2) \rightarrow z_1\ z_2))$$

In this section we prove correctness of a specialization (caseS). The proof for (caseD) requires an additional transformation (cpcx) and is given in subsection 4.11, where also more examples can be found.
The complete set of forking and commuting diagrams for (caseS) are given below.

$$\begin{tikzcd}
\cdot \arrow[r, "caseS"] \arrow[d, "{no,a}"'] & \cdot \arrow[d, "{no,a}"] \\
\cdot \arrow[r, "caseS"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "caseS"] \arrow[d, "{no,a_1}"'] & \cdot \arrow[dl, "{no,a_1}"] \\
\cdot &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "caseS"] \arrow[d, "{no,a_1}"'] & \cdot \arrow[d, "{no,a_1}"] \\
\cdot \arrow[r, "abs1"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "caseS"] \arrow[d, "{no,case}"'] & \cdot \arrow[d, "{no,case}"] \\
\cdot & \cdot \arrow[l, "{ve1,*}"']
\end{tikzcd}$$

$$\begin{tikzcd}
\cdot \arrow[rr, "caseS"] \arrow[d, "{no,case}"'] & & \cdot \arrow[d, "{no,case}"] \\
\cdot \arrow[r, "caseS"'] & \cdot \arrow[r, leftrightarrow, "{ve1,*}"'] & \cdot
\end{tikzcd}$$

Here $a_1 \in \{$seq-c, choice, case$\}$. In the first and last diagram the resulting (case) step may or may not be a normal order reduction.
The third diagram happens when a (case) target is removed by a normal order step but the `letrec` where the constructor is bound remains. Below is an example with a normal order (choice) reduction. Similar examples may happen for (seq) and (case) normal order reductions.

$$\begin{array}{ll}
\texttt{letrec}\ x = (c\ t)\ \texttt{in}\ (\text{choice}\ (\texttt{case}\ x\ (c\ y) \to s)\ u) & \xrightarrow{no,choice} \\
\texttt{letrec}\ x = (c\ t)\ \texttt{in}\ u & \xrightarrow{abs1} \\
\texttt{letrec}\ x = (c\ z), z = t\ \texttt{in}\ u &
\end{array}$$

$$\begin{array}{ll}
\texttt{letrec}\ x = (c\ t)\ \texttt{in}\ (\text{choice}\ (\texttt{case}\ x\ (c\ y) \to s)\ u) & \xrightarrow{case-in} \\
\texttt{letrec}\ x = (c\ z), z = t\ \texttt{in}\ (\text{choice}\ (\texttt{letrec}\ y = z\ \texttt{in}\ s)\ u) & \xrightarrow{no,choice} \\
\texttt{letrec}\ x = (c\ z), z = t\ \texttt{in}\ u &
\end{array}$$

The fourth diagram arises when two case expressions share the constructor and one of the case expressions is in the pattern expression that does not get matched. Here c' is any constructor other than c.

$$\begin{array}{ll}
\texttt{letrec}\ x = (c\ t)\ \texttt{in} & \\
(\texttt{case}\ x\ ((c\ y) \to s)((c'\ y') \to (\texttt{case}\ x\ (c\ z) \to s'))) & \xrightarrow{case-in} \\
\texttt{letrec}\ x = (c\ u), u = t\ \texttt{in} & \\
(\texttt{case}\ x\ ((c\ y) \to s)((c'\ y') \to (\texttt{letrec}\ z = u\ \texttt{in}\ s'))) & \xrightarrow{no,case-in} \\
\texttt{letrec}\ x = (c\ w), w = u, u = t\ \texttt{in}\ \texttt{letrec}\ y = w\ \texttt{in}\ s & \xrightarrow{ve1} \\
\texttt{letrec}\ x = (c\ u), u = t\ \texttt{in}\ \texttt{letrec}\ y = u\ \texttt{in}\ s &
\end{array}$$

$$\begin{array}{ll}
\texttt{letrec}\ x = (c\ t)\ \texttt{in} & \\
(\texttt{case}\ x\ ((c\ y) \to s)((c'\ y') \to (\texttt{case}\ x\ (c\ z) \to s'))) & \xrightarrow{no,case-in} \\
\texttt{letrec}\ x = (c\ u), u = t\ \texttt{in}\ \texttt{letrec}\ y = u\ \texttt{in}\ s &
\end{array}$$

The number of (ve1) steps is the same as the number of arguments of the constructor. This situation arises for both (case-in) and (case-e) reductions.

The fifth diagram arises in situations illustrated by the following example:

$$\begin{array}{ll}
\texttt{letrec}\ x = (c\ t_1\ t_2)\ \texttt{in}\ (\texttt{case}\ x\ ((c\ y_1\ y_2) \to \texttt{case}\ x\ ((c\ z_1\ z_2) \to s))) & \xrightarrow{no,case} \\
\texttt{letrec}\ x = (c\ q_1, q_2), q_1 = t_1, q_2 = t_2\ \texttt{in} & \\
(\texttt{letrec}\ y_1 = q_1, y_2 = q_2\ \texttt{in}\ \texttt{case}\ x\ ((c\ z_1\ z_2) \to s)) & \xrightarrow{case} \\
\texttt{letrec}\ x = (c\ q_3, q_4), q_1 = t_1, q_2 = t_2, q_3 = q_1, q_4 = q_2\ \texttt{in} & \\
(\texttt{letrec}\ y_1 = q_1, y_2 = q_2\ \texttt{in}\ \texttt{letrec}\ z_1 = q_3, z_2 = q_4\ \texttt{in}\ s) & \xrightarrow{ve1,*} \\
\texttt{letrec}\ x = (c\ q_1, q_2), q_1 = t_1, q_2 = t_2\ \texttt{in} & \\
(\texttt{letrec}\ y_1 = q_1, y_2 = q_2\ \texttt{in}\ (\texttt{letrec}\ z_1 = q_1, z_2 = q_2\ \texttt{in}\ s)) &
\end{array}$$

$$\begin{array}{ll}
\texttt{letrec}\ x = (c\ t_1\ t_2)\ \texttt{in}\ (\texttt{case}\ x\ ((c\ y_1\ y_2) \to \texttt{case}\ x\ ((c\ z_1\ z_2) \to s))) & \xrightarrow{case} \\
\texttt{letrec}\ x = (c\ q_1, q_2), q_1 = t_1, q_2 = t_2\ \texttt{in} & \\
(\texttt{case}\ x\ ((c\ y_1\ y_2) \to (\texttt{letrec}\ z_1 = q_1, z_2 = q_2\ \texttt{in}\ s))) & \xrightarrow{no,case} \\
\texttt{letrec}\ x = (c\ q_3, q_4), q_1 = t_1, q_2 = t_2, q_3 = q_1, q_4 = q_2\ \texttt{in} & \\
(\texttt{letrec}\ y_1 = q_3, y_2 = q_4\ \texttt{in}\ (\texttt{letrec}\ z_1 = q_1, z_2 = q_2\ \texttt{in}\ s)) & \xrightarrow{ve1,*} \\
\texttt{letrec}\ x = (c\ q_1, q_2), q_1 = t_1, q_2 = t_2\ \texttt{in} & \\
(\texttt{letrec}\ y_1 = q_1, y_2 = q_2\ \texttt{in}\ (\texttt{letrec}\ z_1 = q_1, z_2 = q_2\ \texttt{in}\ s)) &
\end{array}$$

The number of (ve1) steps equals to the number of variables that need to be eliminated, i.e. the number of arguments to the constructor. The resulting (case) reduction on the bottom of the diagram is never a normal order reduction since the marking algorithm on the term resulting from a normal-order case will mark the newly added `letrec` with S, so the next normal order step would be (llet-in) or (llet-e).

The following lemma is easily derived by considering cases in definition of WHNF:

Lemma 4.17. *If $s \xrightarrow{case} t$ is not a normal order reduction then s is a WHNF if and only if t is a WHNF.*

Proposition 4.18. *The transformation (caseS), i.e. the surface version of (case-in) and (case-e), is correct, i.e. if $s \xrightarrow{caseS} t$ then $s \sim_c t$.*

Proof. We show that (caseS) preserves convergence in a surface context S, then by the Context Lemma 3.2 and by the Corollary 3.5 it preserves convergence in any context C.

We start by listing all commuting and forking diagrams involved in the proof. As in the above diagrams, $a_1 \in$ {seq-c, choice, case}.

– Commuting diagrams:

$$\begin{array}{ll}
(1) & \xrightarrow{caseS} \xrightarrow{no,a} \leadsto \xrightarrow{no,a} \xrightarrow{caseS} \\
(2) & \xrightarrow{caseS} \xrightarrow{no,a} \leadsto \xrightarrow{no,a} \\
(3) & \xrightarrow{caseS} \xrightarrow{no,a_1} \leadsto \xrightarrow{no,a_1} \xrightarrow{abs1} \\
(4) & \xrightarrow{caseS} \xrightarrow{no,case} \leadsto \xrightarrow{no,case} \xleftarrow{ve1,*} \\
(5) & \xrightarrow{caseS} \xrightarrow{no,case} \leadsto \xrightarrow{no,case} \xrightarrow{caseS} \xleftrightarrow{ve1,*} \\
(6) & \xrightarrow{abs1} \xrightarrow{no,a} \leadsto \xrightarrow{no,a} \xrightarrow{abs1} \\
(7) & \xrightarrow{abs1} \xrightarrow{no,a} \leadsto \xrightarrow{no,a} \\
(8) & \xrightarrow{abs1} \xrightarrow{no,case} \leadsto \xrightarrow{no,case} \xleftarrow{ve1,*} \\
(9) & \xleftrightarrow{ve1,*} \xrightarrow{no,a} \leadsto \xrightarrow{no,a} \xleftrightarrow{ve1,*}
\end{array}$$

– Forking diagrams:

$$\begin{array}{ll}
(1) & \xleftarrow{caseS} \xrightarrow{no,a} \leadsto \xrightarrow{no,a} \xleftarrow{caseS} \\
(2) & \xleftarrow{caseS} \xrightarrow{no,a} \leadsto \xrightarrow{no,a} \\
(3) & \xleftarrow{caseS} \xrightarrow{no,a_1} \leadsto \xrightarrow{abs1} \xleftarrow{no,a} \\
(4) & \xleftarrow{caseS} \xrightarrow{no,case} \leadsto \xrightarrow{no,case} \xrightarrow{ve1,*} \\
(5) & \xleftarrow{caseS} \xrightarrow{no,case} \leadsto \xrightarrow{no,case} \xleftrightarrow{ve1,*} \xleftarrow{caseS} \\
(6) & \xleftarrow{abs1} \xrightarrow{no,a} \leadsto \xrightarrow{no,a} \xleftarrow{abs1} \\
(7) & \xleftarrow{abs1} \xrightarrow{no,a} \leadsto \xrightarrow{no,a} \\
(8) & \xleftarrow{abs1} \xrightarrow{no,case} \leadsto \xrightarrow{no,case} \xrightarrow{ve1,*} \\
(9) & \xleftrightarrow{ve1,*} \xrightarrow{no,a} \leadsto \xrightarrow{no,a} \xleftrightarrow{ve1,*}
\end{array}$$

Assume $R[t]\Downarrow$, i.e. there exists a normal order reduction sequence $R[t] \xrightarrow{*} t'$, where t' is a WHNF. We consider a reduction sequence $R[s] \xrightarrow{caseS} R[t] \xrightarrow{*} t'$, where $R[t] \xrightarrow{*} t'$ is the given evaluation of $R[t]$. By Lemma 4.2 $R[t] \xrightarrow{caseS} R[s]$ in a surface context, thus we can use the above commuting diagrams to transform the given sequence into a normal order sequence $R[s] \xrightarrow{*} s'$, where s' is a WHNF. We apply the transformations at the leftmost matching position of the sequence. In a pattern that includes a multidirectional (ve1) sequence (diagram 9) we always select the longest (ve1) sequence that matches.

We call each of the non-normal-order $\xrightarrow{caseS}$, $\xrightarrow{abs1}$, and $\xleftrightarrow{ve1,*}$ a *segment*. The *counter* of a segment is the number of normal order steps to the right of that segment. The measure is the string of counters for all segments in a reduction sequence, where the segments are ordered left-to-right. For instance, the sequence

$$\xrightarrow{caseS} \xrightarrow{no,a_1} \xleftrightarrow{ve1} \xrightarrow{no,a_2}$$

where the (case) reduction is not normal order, has the measure $(2, 1)$ since there are two normal order steps to the right of (case) and one to the right of the (ve1) segment.

The sequences of counters ordered lexicographically form an ordered sequence measure defined in Definition 4.3. The measure strictly decreases with each transformation in the commuting diagrams above:

- In diagrams 1 (in the case when the bottom (case) step is not a normal order step), 3, 4, 6, 8, and 9 a normal order step is switched with a segment (the kind of a segment may change during the switch). Then the new segment has the counter 1 less than the old one and no other counters change so the measure decreases. Note that in diagrams 4, 8, 9 the resulting (ve1) segment may be merged with another (ve1) segment in the following situation (shown for diagram 4):

  $$\xrightarrow{caseS}\xrightarrow{no,a}\xleftrightarrow{ve1} \rightsquigarrow \xrightarrow{no,a}\xleftrightarrow{ve1}\xleftrightarrow{ve1} = \xrightarrow{no,a}\xleftrightarrow{ve1}$$

  In this case the measure decreases even further since the total number of segments is reduced as well.
- In diagram 1 it is also possible that the resulting (caseS) becomes a normal order step. Since there is only one (caseS) step in the sequence and there are no (abs) or (ve1) segments to the left of it, no counters increase by this transformation. The segment corresponding to (caseS) step is removed. Thus the measure decreases.
- In diagrams 2, 7 one of the segments is removed, and the next segment to the right (if any) has a counter that is at least 1 less than the removed segment.
- In diagram 5 one segment is replaced by two, each with a counter lower by 1 than that of the removed segment. Since the sequences of counters are ordered lexicographically, the new sequence is less than the one it replaced. Note that it may be possible that the (caseS) is turned into a normal-order reduction. In this case the arguments are similar as for diagram 1.

  Alternatively in this case it is possible that the (ve1) segment is merged with a subsequent one, as discussed for diagrams 4, 8, 9 above. In this case the counter for the (ve1) segment does not change and there is only one new segment, the non-normal-order (case) step, which has a counter one less than the counter for the old (case) step.

Since by Lemma 4.4 there is no infinite descending chain of sequences of counters, the transformation process terminates. Therefore we have constructed a sequence $R[s] \xrightarrow{*} s' \xrightarrow{a,*} t'$ where $a \in \{\text{case, abs1, ve1}\}$. Recall that t' is a WHNF. Then by Lemmas 4.12,4.15, and 4.17 s' is also a WHNF and one direction of the lemma is proven.

The other direction is analogous, using the forking diagrams and the same measure. □

4.10 Correctness of (cpcx).

The (cpcx) transformation is needed in further proofs. It has the following commuting and forking diagrams, taking into account the (ve1) rule:

$$\begin{tikzcd}
\cdot \arrow[r, "cpcx"] \arrow[d, "{no,a}"'] & \cdot \arrow[d, "{no,a}"] \\
\cdot \arrow[r, "cpcx"] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "cpcx"] \arrow[d, "{no,a_1}"'] & \cdot \arrow[dl, "{no,a_1}"] \\
\cdot &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "cpcx"] \arrow[d, "{no,a_1}"'] & \cdot \arrow[d, "{no,a_1}"] \\
\cdot \arrow[r, "abs1"] & \cdot
\end{tikzcd}$$

$$\begin{tikzcd}
\cdot \arrow[r, "cpcx"] \arrow[d, "{no,case}"'] & \cdot \arrow[d, "{no,case}"] \\
\cdot & \cdot \arrow[l, "{ve1,*}"]
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[rrr, "cpcx"] \arrow[d, "{no,cp}"'] & & & \cdot \arrow[d, "{no,cp}"] \\
\cdot \arrow[r, "cpcx"'] & \cdot \arrow[r, "cpcx"'] & \cdot \arrow[r, "{ve1,*}"'] & \cdot
\end{tikzcd}$$

$$\begin{tikzcd}
\cdot \arrow[rrr, "cpcx"] \arrow[d, "{no,case}"'] & & & \cdot \arrow[d, "{no,case}"] \\
\cdot \arrow[r, "cpcx"] & \cdot \arrow[r, "{ve1,*}"] & \cdot & \cdot \arrow[l, "{ve1,*}"']
\end{tikzcd}$$

Where $a_1 \in \{\text{seq, case, choice}\}$

An example for the fourth diagram is when the target of (cpcx) occurs in an unused alternative of a `case`:

$$\begin{array}{ll}
\texttt{letrec}\ x = (c_1\ t)\ \texttt{in}\ (\texttt{case}\ x\ (c_1 y \to s)(c_2\ y' \to C[x])) & \xrightarrow{cpcx} \\
\texttt{letrec}\ x = (c_1\ z), z = t\ \texttt{in}\ (\texttt{case}\ x\ (c_1 y \to s)(c_2\ y' \to C[c_1\ z])) & \xrightarrow{no,case} \\
\texttt{letrec}\ x = (c_1\ z'), z' = z, z = t\ \texttt{in}\ \texttt{letrec}\ y = z'\ \texttt{in}\ s & \xrightarrow{ve1} \\
\texttt{letrec}\ x = (c_1\ z), z = t\ \texttt{in}\ \texttt{letrec}\ y = z\ \texttt{in}\ s &
\end{array}$$

$$\begin{array}{ll}
\texttt{letrec}\ x = (c_1\ t)\ \texttt{in}\ (\texttt{case}\ x\ (c_1 y \to s)(c_2\ y' \to C[x])) & \xrightarrow{no,case} \\
\texttt{letrec}\ x = (c_1\ z), z = t\ \texttt{in}\ \texttt{letrec}\ y = z\ \texttt{in}\ s &
\end{array}$$

The fifth diagram occurs when a (cpcx) target variable is duplicated using a (cp) step. The (ve1) steps are needed to remove the extra chain variables generated by the second (cpcx) in the bottom row of the diagram.

One non-trivial example of the last diagram is when the target of (cpcx) is a part of a variable chain:

$$
\begin{array}{ll}
\texttt{letrec}\ x = (c\ t), y = x\ \texttt{in}\ (\texttt{case}\ y\ (c\ z \to s)) & \xrightarrow{cpcx} \\
\texttt{letrec}\ x = (c\ q), q = t, y = (c\ q)\ \texttt{in}\ (\texttt{case}\ y\ (c\ z \to s)) & \xrightarrow{no,case} \\
\texttt{letrec}\ x = (c\ q'), q' = q, q = t, y = (c\ q)\ \texttt{in}\ \texttt{letrec}\ z = q'\ \texttt{in}\ s & \xrightarrow{ve1} \\
\texttt{letrec}\ x = (c\ q), q = t, y = (c\ q)\ \texttt{in}\ \texttt{letrec}\ z = q\ \texttt{in}\ s &
\end{array}
$$

$$
\begin{array}{ll}
\texttt{letrec}\ x = (c\ t), y = x\ \texttt{in}\ (\texttt{case}\ y\ (c\ z \to s)) & \xrightarrow{no,case} \\
\texttt{letrec}\ x = (c\ q), q = t, y = x\ \texttt{in}\ \texttt{letrec}\ z = q\ \texttt{in}\ s & \xrightarrow{cpcx} \\
\texttt{letrec}\ x = (c\ q'), q' = q, q = t, y = (c\ q')\ \texttt{in}\ \texttt{letrec}\ z = q\ \texttt{in}\ s & \xrightarrow{ve1} \\
\texttt{letrec}\ x = (c\ q), q = t, y = (c\ q)\ \texttt{in}\ \texttt{letrec}\ z = q\ \texttt{in}\ s &
\end{array}
$$

Lemma 4.19. *If $s \xrightarrow{cpcx} t$ then s is a WHNF if and only if t is.*

Proof. The first two cases of WHNF are trivial. Consider the last case and the (cpcx) reduction. For simplicity we are showing only one argument of the constructor.

$$
\begin{array}{ll}
\texttt{letrec}\ x_1 = (c\ t), \{x_i = x_{i-1}\}_{i=2}^{n}, Env\ \texttt{in}\ x_n & \xrightarrow{cpcx} \\
\texttt{letrec}\ x_1 = (c\ y), y = t, \{x_i = x_{i-1}\}_{i=2}^{n}, Env\ \texttt{in}\ (c\ y) &
\end{array}
$$

Since $(c\ y)$ is a value, the resulting term fits case 2 of the definition. □

Proposition 4.20. *If $s \xrightarrow{cpcx} t$ then $s \sim_c t$.*

Proof. By the Context lemma it is sufficient to show that $R[t]\!\downarrow$ if and only if $R[s]\!\downarrow$. Moreover, by Lemma 4.2 $R[s] \xrightarrow{cpcx} R[t]$ in a surface context, thus the commuting and forking diagrams are applicable.
Assume that $R[t]\!\downarrow$. We would like to prove that $R[s]\!\downarrow$.
Commuting diagrams are generated from their pictorial representation above. We also need diagrams for (abs1) given in Section 4.8 and for (ve1) given in Section 4.7.

$$
\begin{array}{ll}
(1) & \xrightarrow{cpcx}\ \xrightarrow{no,a}\ \leadsto\ \xrightarrow{no,a}\ \xrightarrow{cpcx} \\
(2) & \xrightarrow{cpcx}\ \xrightarrow{no,a_1}\ \leadsto\ \xrightarrow{no,a_1} \\
(3) & \xrightarrow{cpcx}\ \xrightarrow{no,a_1}\ \leadsto\ \xrightarrow{no,a_1}\ \xrightarrow{abs1} \\
(4) & \xrightarrow{cpcx}\ \xrightarrow{no,case}\ \leadsto\ \xrightarrow{no,case}\ \xleftarrow{ve1,*} \\
(5) & \xrightarrow{cpcx}\ \xrightarrow{no,case}\ \leadsto\ \xrightarrow{no,case}\ \xrightarrow{cpcx}\ \xleftrightarrow{ve1,*} \\
(6) & \xrightarrow{cpcx}\ \xrightarrow{no,cp}\ \leadsto\ \xrightarrow{no,cp}\ \xrightarrow{cpcx}\ \xrightarrow{cpcx}\ \xleftrightarrow{ve1,*} \\
(7) & \xrightarrow{abs1}\ \xrightarrow{no,a}\ \leadsto\ \xrightarrow{no,a}\ \xrightarrow{abs1} \\
(8) & \xrightarrow{abs1}\ \xrightarrow{no,a}\ \leadsto\ \xrightarrow{no,a} \\
(9) & \xrightarrow{abs1}\ \xrightarrow{no,case}\ \leadsto\ \xrightarrow{no,case}\ \xleftarrow{ve1,*} \\
(10) & \xleftrightarrow{ve1,*}\ \xrightarrow{no,a}\ \leadsto\ \xrightarrow{no,a}\ \xleftrightarrow{ve1,*}
\end{array}
$$

We introduce a counter-based measure, similar to that in the proof of Proposition 4.18. There are three kinds of segments of non-normal-order steps: each of (cpcx) and (abs1) forms its own segment and every contiguous sequence of forward and backward (ve1) steps is a segment. For each such segment we define a counter as the total number of normal-order steps to the right of the segment in the reduction sequence. As in the proof of Proposition 4.18, we consider an ordered (left-to-right) sequence of these numbers and order these sequences lexicographically. This is an ordered sequence measure (see Definition 4.3) and by Lemma 4.4 there is no infinitely decreasing chain of such sequences.
Commuting diagrams can be applied at any matching position in a reduction sequence. We show that every transformation modifies a part of the reduction sequence in such a way that the measure strictly decreases. We consider cases of the diagram. Note that we have to assume that there possibly are other (cpcx), (abs1), and (ve) steps generated in earlier transformations. However, since the transformations do not generate normal order steps, only a part of the reduction sequence that actually gets transformed changes its corresponding numbers in the measure.
Below we consider cases of commuting diagrams. The case numbers correspond to the numbering of the commuting diagrams above.

- Cases 1, 3, and 7 of the commuting diagrams remove a segment to the left of a normal order step and create a segment to the right of it so the counter for that segment is one less than the counter of the one removed, and the rest of the counters are not changed.
- Cases 2, 8 remove a segment and don't change any other counters, thus the measure decreases.
- Cases 4, 9, and 10 are similar to cases 1, 3, and 7 in that the counter of the generated (ve1) segment is one less than the counter of the one removed, with the difference that if there is another (ve1) segment directly following the transformed sequence than the two (ve1) segments are merged and the measure decreases even further.
- Cases 5 and 6 remove a (cpcx) segment on the left of a normal order step and replace it by several segments to the right. Each of the new segments has a counter one less than that of the removed one, thus the measure decreases.

Thus we have shown that if there is a normal order reduction of $R[t]$ to a WHNF t' then there is a normal order reduction of $R[s]$ that leads to a term s' connected to t' by a sequence of (cpcx), (abs1), and (ve) reductions. By Lemmas 4.12,4.15, and 4.19 s' is a WHNF so the claim holds.
The proof in the other direction (i.e. assuming that $R[s]\downarrow$ and proving that $R[t]\downarrow$ is similar, using the forking diagrams and the same measure.
By the Context Lemma 3.2 it follows that $s \sim_c t$. $\square$

4.11 Correctness of (caseD)

Recall that (caseD) refers to the "deep" version of (case-in) and (case-e), i.e. the version where the entire reduction takes place in a surface context, but the context C in the rule is not a surface context. The following example shows that diagrams for (caseD) may duplicate (case) steps.

$$
\begin{array}{lr}
\texttt{letrec}\ x = (\lambda z.\texttt{case}\ y\ (c\ u_1\ u_2) \to s), y = (c\ s_1\ s_2)\ \texttt{in}\ (x\ t) & \xrightarrow{caseD} \\
\texttt{letrec}\ x = (\lambda z.\texttt{letrec}\ u_1 = q_1, u_2 = q_2\ \texttt{in}\ s), y = (c\ q_1, q_2), & \\
\quad q_1 = s_1, q_2 = s_2\ \texttt{in}\ (x\ t) & \xrightarrow{no,cp} \\
\texttt{letrec}\ x = (\lambda z.\texttt{letrec}\ u_1 = q_1, u_2 = q_2\ \texttt{in}\ s), y = (c\ q_1, q_2), & \\
\quad q_1 = s_1, q_2 = s_2\ \texttt{in}\ ((\lambda z.\texttt{letrec}\ u_1 = q_1, u_2 = q_2\ \texttt{in}\ s)\ t) &
\end{array}
$$

$$
\begin{array}{lr}
\texttt{letrec}\ x = (\lambda z.\texttt{case}\ y\ (c\ u_1\ u_2) \to s), y = (c\ s_1\ s_2)\ \texttt{in}\ (x\ t) & \xrightarrow{no,cp} \\
\texttt{letrec}\ x = (\lambda z.\texttt{case}\ y\ (c\ u_1\ u_2) \to s), y = (c\ s_1\ s_2)\ \texttt{in} & \\
\quad (((\lambda z.\texttt{case}\ y\ (c\ u_1\ u_2) \to s))\ t) & \xrightarrow{caseD} \\
\texttt{letrec}\ x = (\lambda z.\texttt{letrec}\ u_1 = q_1, u_2 = q_2\ \texttt{in}\ s) \to s), y = (c\ q_1\ q_2), & \\
\quad q_1 = s_1, q_2 = s_2\ \texttt{in}\ (((\lambda z.\texttt{case}\ y\ (c\ u_1\ u_2) \to s))\ t) & \xrightarrow{caseD} \\
\texttt{letrec}\ x = (\lambda z.\texttt{letrec}\ u_1 = q_1, u_2 = q_2\ \texttt{in}\ s) \to s), y = (c\ q_1'\ q_2'), q_1' = q_1, & \\
\quad q_2' = q_2, q_1 = s_1, q_2 = s_2\ \texttt{in}\ ((\lambda z.\texttt{letrec}\ u_1 = q_1', u_2 = q_2'\ \texttt{in}\ s) & \xrightarrow{ve1*} \\
\texttt{letrec}\ x = (\lambda z.\texttt{letrec}\ u_1 = q_1, u_2 = q_2\ \texttt{in}\ s), y = (c\ q_1, q_2), & \\
\quad q_1 = s_1, q_2 = s_2\ \texttt{in}\ ((\lambda z.\texttt{letrec}\ u_1 = q_1, u_2 = q_2\ \texttt{in}\ s)\ t) &
\end{array}
$$

Note that, unlike in a similar case for (seq), multiple copies of (caseD) may also be interleaved with (abs) and (ve1) steps. This complicates constructing a sequence where only leftmost (case) steps may become normal order reductions, which in turn complicates the measure-based argument. Instead we prove correctness of (caseD) by representing it as a sequence of steps whose correctness we have already proven.

Proposition 4.21. *(caseD) is a correct program transformation.*

Proof. Proposition 4.1 shows that (case-c) is a correct program transformation. From Propositions 4.13 and 4.20 above we obtain we obtain that (ve) and (cpcx) are correct program transformations. We show by induction that (caseD) is correct by using the correctness of the transformations (cpcx), (case-c) and (ve). The induction is on the length of the variable chain in the reduction (caseD). We use (cpcx) to copy the constructor into each of the variables in the chain, one by one. For the base case the (caseD) reduction can also be performed by the sequence of reductions: $\xrightarrow{cpcx} \cdot \xrightarrow{case-c}$

$$
\begin{array}{rl}
 & (\texttt{letrec}\ x = c\ t, Env\ \texttt{in}\ C[\texttt{case}_t\ x\ (c\ z \to s)\ alts]) \\
\xrightarrow{cpcx} & (\texttt{letrec}\ x = c\ y, y = t, Env\ \texttt{in}\ C[\texttt{case}_T\ (c\ y)\ (c\ z \to s)\ alts]) \\
\xrightarrow{case-c} & (\texttt{letrec}\ x = c\ y, y = t, Env\ \texttt{in}\ C[(\texttt{letrec}\ z = y\ \texttt{in}\ s)])
\end{array}
$$

For the induction we replace a (caseD) reduction operating on a chain $\{x_i = x_{i-1}\}_{i=2}^m$ with the sequence $\xrightarrow{cpcx} \cdot \xrightarrow{caseD} \cdot \xrightarrow{ve^n} \cdot \xleftarrow{ve^n} \cdot \xleftarrow{cpcx}$, where

n is the arity of the constructor and the (caseD) reduction operates on the chain $\{x_i = x_{i-1}\}_{i=3}^m$:

$$
\begin{array}{ll}
 & (\texttt{letrec}\ x_1 = c\ \overrightarrow{t}, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}\ \texttt{in}\ C[\texttt{case}_T\ x_m\ (c\ \overrightarrow{z} \to s)\ \mathit{alts}]) \\
\xrightarrow{cpcx} & \texttt{letrec}\ x_1 = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^n, x_2 = c\ \overrightarrow{y}, \{x_i = x_{i-1}\}_{i=3}^m, \mathit{Env} \\
 & \texttt{in}\ C[\texttt{case}_T\ x_1\ (c\ \overrightarrow{z} \to s)\ \mathit{alts}] \\
\xrightarrow{caseD} & \texttt{letrec}\ x_1 = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^n, x_2 = c\ \overrightarrow{y'}, \{y'_i = y_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=3}^m, \mathit{Env} \\
 & \texttt{in}\ C[(\texttt{letrec}\ \{z_i = y'_i\}_{i=1}^n\ \texttt{in}\ s)] \\
\xrightarrow{ve,*} & \texttt{letrec}\ x_1 = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^n, x_2 = c\ \overrightarrow{y'}, \{y'_i = y_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=3}^m, \mathit{Env} \\
 & \texttt{in}\ C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^n\ \texttt{in}\ s)] \\
\xleftarrow{ve,*} & \texttt{letrec}\ x_1 = c\ \overrightarrow{y'}, \{y_i = t_i\}_{i=1}^n, x_2 = c\ \overrightarrow{y'}, \{y'_i = y_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=3}^m, \mathit{Env} \\
 & \texttt{in}\ C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^n\ \texttt{in}\ s)] \\
\xleftarrow{cpcx} & \texttt{letrec}\ x_1 = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env} \\
 & \texttt{in}\ C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^n\ \texttt{in}\ s)] \qquad \square
\end{array}
$$

By the inductive hypothesis the (caseD) step above may in turn be replaced by a sequence of (cpcx), (caseD), and (ve) steps. Since the length of the variable chain decreases with every induction step, eventually the base case (no variable chain) will be reached.

Proposition 4.22. *The reduction (case) is a correct program transformation.*

Proof. Follows from Propositions 4.21 and 4.1. □

4.12 Correctness of (gc), (ucp), and (abs2)

In this section we prove correctness of transformation rules (gc) and (ucp). The (gc) rule has two versions, denoted (gc1) and (gc2), and the (ucp) rule has three versions, denoted (ucp1), (ucp2), and (ucp3). The rules are defined in Figure 3. We use a notation ucp/gc to denote a step that can be either a ucp or a gc step. As we discuss further in this section, ucp/gc transformation rules are different from the ones considered earlier in that they may perform a step that would otherwise require a normal-order reduction. At the end of the section we also show correctness of (abs2) based on correctness of (ucp).
We start by showing forking diagrams for (ucp) and (gc) rules. Commuting diagrams are more complex. They are given in Lemma 4.25.
A complete set of forking diagrams for (gc) rules is as follows:

$$\begin{tikzcd}
\cdot \arrow[r, "gc"] \arrow[d, "no,a"'] \arrow[dr, phantom, "A"] & \cdot \arrow[d, dashed, "no,a"] \\
\cdot \arrow[r, dashed, "gc"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "gc"] \arrow[d, "no,a_1"'] \arrow[dr, phantom, "B"] & \cdot \arrow[dl, dashed, "no,a_1"] \\
\cdot &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "gc2"] \arrow[d, "no,lll"'] \arrow[dr, phantom, "C"] & \cdot \\
\cdot \arrow[ur, dashed, "gc"'] &
\end{tikzcd}$$

where $a_1 \in \{\text{seq, choice, case}\}$.

An example for diagram C is as follows:

$$\begin{array}{l}\texttt{letrec}\ x = s\ \texttt{in}\ (\texttt{letrec}\ Env\ \texttt{in}\ t) \xrightarrow{gc2} \\ \texttt{letrec}\ Env\ \texttt{in}\ t\end{array}$$

$$\begin{array}{l}\texttt{letrec}\ x = s\ \texttt{in}\ (\texttt{letrec}\ Env\ \texttt{in}\ t) \xrightarrow{llet-in} \\ \texttt{letrec}\ x = s, Env\ \texttt{in}\ t \xrightarrow{gc1} \\ \texttt{letrec}\ Env\ \texttt{in}\ t\end{array}$$

A complete set of forking diagrams for (ucp) is as follows:

$$\begin{tikzcd}\cdot \arrow[r, "ucp"] \arrow[d, "no{,}a"'] \arrow[dr, phantom, "A"] & \cdot \arrow[d, dashed, "no{,}a"] \\ \cdot \arrow[r, dashed, "ucp"] & \cdot\end{tikzcd}
\qquad
\begin{tikzcd}\cdot \arrow[r, "ucp"] \arrow[d, "no{,}a_1"'] & \cdot \arrow[dl, dashed, "no{,}a_1"] \\ \cdot & \end{tikzcd}
\qquad
\begin{tikzcd}\cdot \arrow[r, "ucp"] \arrow[d, "no{,}cp"'] & \cdot \\ \cdot \arrow[ur, dashed, "gc"'] & \end{tikzcd}$$

$$\begin{tikzcd}\cdot \arrow[r, "ucp"] \arrow[d, "no{,}lll+"'] \arrow[dr, phantom, "D"] & \cdot \arrow[d, dashed, "no{,}lll*"] \\ \cdot \arrow[r, dashed, "ucp"] & \cdot\end{tikzcd}
\qquad
\begin{tikzcd}\cdot \arrow[r, "ucp"] \arrow[d, "no{,}a"'] \arrow[dr, phantom, "E"] & \cdot \arrow[d, dashed, "no{,}a"] \\ \cdot \arrow[r, dashed, "gc"] & \cdot\end{tikzcd}
\qquad
\begin{tikzcd}\cdot \arrow[rr, "ucp"] \arrow[d, "no{,}case"'] & & \cdot \arrow[d, dashed, "no{,}case"] \\ \cdot \arrow[d, "no{,}llet"'] & F & \cdot \arrow[d, dashed, "no{,}llet?"] \\ \cdot \arrow[r, dashed, "gc"] & \cdot \arrow[r, dashed, "ve1*"] & \cdot\end{tikzcd}$$

where $a_1 \in$ {seq, choice, case}.
An example for diagram C is as follows:

$$\begin{array}{l}\texttt{letrec}\ x = v, Env\ \texttt{in}\ x \xrightarrow{ucp1} \\ \texttt{letrec}\ Env\ \texttt{in}\ v\end{array}$$

$$\begin{array}{l}\texttt{letrec}\ x = v, Env\ \texttt{in}\ x \xrightarrow{no,cp} \\ \texttt{letrec}\ x = v, Env\ \texttt{in}\ v \xrightarrow{gc1} \\ \texttt{letrec}\ Env\ \texttt{in}\ v\end{array}$$

There is also a similar example that involves (ucp3) and (gc2).
A example for diagram D is as follows (s is a constructor):

$$\begin{array}{l}\texttt{letrec}\ y = (\texttt{letrec}\ x = s\ \texttt{in}\ x)\ \texttt{in}\ y \xrightarrow{ucp3} \\ \texttt{letrec}\ y = s\ \texttt{in}\ y\end{array}$$

$$\begin{array}{l}\texttt{letrec}\ y = (\texttt{letrec}\ x = s\ \texttt{in}\ x)\ \texttt{in}\ y \xrightarrow{no,llet-e} \\ \texttt{letrec}\ y = x, x = s\ \texttt{in}\ y \xrightarrow{ucp} \\ \texttt{letrec}\ y = s\ \texttt{in}\ y\end{array}$$

The following example illustrates diagram D for cases when the given normal order sequence is non-empty. A similar situation occurs also for (lseq) and (lcase).

$$\begin{array}{ll}
\texttt{letrec}\ x = (\texttt{letrec}\ y = s\ \texttt{in}\ t), Env\ \texttt{in}\ (x\ r) & \xrightarrow{ucp1} \\
\texttt{letrec}\ Env\ \texttt{in}\ ((\texttt{letrec}\ y = s\ \texttt{in}\ t)\ r) & \xrightarrow{no,lapp} \\
\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{letrec}\ y = s\ \texttt{in}\ (t\ r)) & \xrightarrow{no,llet-in} \\
\texttt{letrec}\ Env, y = s\ \texttt{in}\ (t\ r) &
\end{array}$$

$$\begin{array}{ll}
\texttt{letrec}\ x = (\texttt{letrec}\ y = s\ \texttt{in}\ t), Env\ \texttt{in}\ (x\ r) & \xrightarrow{llet-e} \\
\texttt{letrec}\ Env, x = t, y = s\ \texttt{in}\ (x\ r) & \xrightarrow{ucp1} \\
\texttt{letrec}\ Env, y = s\ \texttt{in}\ (t\ r) &
\end{array}$$

Another case for diagram D is as follows (analogous to that in [SSS06]). We assume that x is not referenced anywhere else in the term.

$$\begin{array}{ll}
\texttt{letrec}\ x = (\texttt{letrec}\ y = s\ \texttt{in}\ t), z = R[x], Env\ \texttt{in}\ R'[z] & \xrightarrow{ucp2} \\
\texttt{letrec}\ z = R[(\texttt{letrec}\ y = s\ \texttt{in}\ t)], Env\ \texttt{in}\ R'[z] & \xrightarrow{no,llet+} \\
\texttt{letrec}\ y = s, z = R[t], Env\ \texttt{in}\ R'[z] &
\end{array}$$

$$\begin{array}{ll}
\texttt{letrec}\ x = (\texttt{letrec}\ y = s\ \texttt{in}\ t), z = R[x], Env\ \texttt{in}\ R'[z] & \xrightarrow{no,llet} \\
\texttt{letrec}\ y = s, x = t, z = R[x], Env\ \texttt{in}\ R'[z] & \xrightarrow{ucp2} \\
\texttt{letrec}\ y = s, z = R[t], Env\ \texttt{in}\ R'[z] &
\end{array}$$

Here the number of (lll) steps in the sequence depends on the structure of the R context, lifting the letrec through the context. Note that for the term

$$\texttt{letrec}\ x = (\texttt{letrec}\ y = t\ \texttt{in}\ s)\ \texttt{in}\ (\text{choice}\ x\ r)$$

the normal order reduction is (choice), not (llet-e), so diagrams for (lll) do not need to take into account interactions with choice.

A sample case for diagram E is as follows:

$$\begin{array}{l}
\texttt{letrec}\ x = s\ \texttt{in}\ (\text{choice}\ x\ r) \xrightarrow{ucp3} \\
(\text{choice}\ s\ r) \xrightarrow{no,choice} \\
r
\end{array}$$

$$\begin{array}{l}
\texttt{letrec}\ x = s\ \texttt{in}\ (\text{choice}\ x\ r) \xrightarrow{no,choice} \\
\texttt{letrec}\ x = s\ \texttt{in}\ r \xrightarrow{gc2} \\
r
\end{array}$$

Diagram F arises in the following and similar cases:

$$\begin{array}{lr}
\texttt{letrec}\ x = (c\, t_1\ t_2), Env\ \texttt{in case}\ \ x\ ((c\, y_1\ y_2) \to s) & \xrightarrow{ucp1} \\
\texttt{letrec}\ x = (c\, t_1\ t_2), Env\ \texttt{in case}\ (c\, t_1\ t_2)\ ((c\, y_1\ y_2) \to s) & \xrightarrow{no,case} \\
\texttt{letrec}\ Env\ \texttt{in letrec}\ y_1 = t_1, y_2 = t_2\ \texttt{in}\ s & \xrightarrow{no,llet} \\
\texttt{letrec}\ Env, y_1 = t_1, y_2 = t_2\ \texttt{in}\ s &
\end{array}$$

$$\begin{array}{lr}
\texttt{letrec}\ x = (c\, t_1\ t_2), Env\ \texttt{in case}\ \ x\ ((c\, y_1\ y_2) \to s) & \xrightarrow{no,case} \\
\texttt{letrec}\ x = (c\ z_1\ \ z_2), z_1 = t_1, z_2 = t_2, Env\ \texttt{in letrec}\ y_1 = z_1, y_2 = z_2\ \texttt{in}\ s & \xrightarrow{llet} \\
\texttt{letrec}\ x = (c\ z_1\ \ z_2), z_1 = t_1, z_2 = t_2, Env y_1 = z_1, y_2 = z_2\ \texttt{in}\ s & \xrightarrow{gc1} \\
\texttt{letrec}\ Env, z_1 = t_1, z_2 = t_2, y_1 = z_1, y_2 = z_2\ \texttt{in}\ s & \xrightarrow{ve1,*} \\
\texttt{letrec}\ Env, z_1 = t_1, z_2 = t_2\ \texttt{in}\ s[z_1/y_1, z_2/y_2] &
\end{array}$$

The two resulting terms are alpha-equivalent. Note that (llet) steps in this and similar examples are normal order since (case) is a normal order reduction. A variant of this example with an empty environment uses a (ucp3) step instead of a (ucp1) and does not have the first (llet) step:

$$\begin{array}{lr}
\texttt{letrec}\ x = (c\, t_1\ t_2)\ \texttt{in case}\ \ x\ ((c\, y_1\ y_2) \to s) & \xrightarrow{ucp3} \\
\texttt{case}\ (c\, t_1\ t_2)\ ((c\, y_1\ y_2) \to s) & \xrightarrow{no,case} \\
\texttt{letrec}\ y_1 = t_1, y_2 = t_2\ \texttt{in}\ s &
\end{array}$$

$$\begin{array}{lr}
\texttt{letrec}\ x = (c\, t_1\ t_2)\ \texttt{in case}\ \ x\ ((c\, y_1\ y_2) \to s) & \xrightarrow{no,case} \\
\texttt{letrec}\ x = (c\ z_1\ \ z_2), z_1 = t_1, z_2 = t_2\ \texttt{in letrec}\ y_1 = z_1, y_2 = z_2\ \texttt{in}\ s & \xrightarrow{no,llet} \\
\texttt{letrec}\ x = (c\ z_1\ \ z_2), z_1 = t_1, z_2 = t_2, y_1 = z_1, y_2 = z_2\ \texttt{in}\ s & \xrightarrow{gc1} \\
\texttt{letrec}\ z_1 = t_1, z_2 = t_2, y_1 = z_1, y_2 = z_2\ \texttt{in}\ s & \xrightarrow{ve1,*} \\
\texttt{letrec}\ z_1 = t_1, z_2 = t_2\ \texttt{in}\ s[z_1/y_1, z_2/y_2] &
\end{array}$$

Note that forking diagrams C for (gc) and C and a particular case of D for (ucp) replace a normal order step by transformation steps only. We call such diagrams *erasing* since they "erase" a normal order step.

Definition 4.23. *A forking diagram in which the resulting sequence does not contain a normal order step is called erasing.*

Diagram D becomes an erasing one in cases when the given normal order (lll) sequence is not empty and the resulting sequence does not have any normal order steps, i.e. it is of the form $\xleftarrow{ucp} \xrightarrow{lll,no,+} \rightsquigarrow \xleftarrow{ucp}$, such as in the first example for diagram D above.
The presence of erasing diagrams complicates commuting diagrams since an erasing diagram considered in the reverse direction allows a transformation step to turn into a normal order step followed by another transformation step. For instance, a (ucp) step is transformed by diagram C into a normal-order (cp) step followed by a (gc) step. We refer to the reverse forms of erasing diagrams

as *generating* diagrams since they generate a normal order step. The generating diagrams for (gc) and (ucp), labeled by G_1, G_2, G_3, are as follows:

$$\begin{tikzcd}
\cdot \arrow[r, "gc"] \arrow[d, dashed, "{no,lll}"'] \arrow[dr, phantom, "G_1"] & \cdot \\
\cdot \arrow[ur, dashed, "gc"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "ucp"] \arrow[d, dashed, "{no,lll}"'] \arrow[dr, phantom, "G_2"] & \cdot \\
\cdot \arrow[ur, dashed, "ucp"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "ucp"] \arrow[d, dashed, "{no,cp}"'] \arrow[dr, phantom, "G_3"] & \cdot \\
\cdot \arrow[ur, dashed, "gc"'] &
\end{tikzcd}$$

When constructing commuting diagrams, we need to take generating diagrams into account. For instance, to construct a commuting diagram for a sequence $\xrightarrow{ucp}\xrightarrow{no,a}$, we need to take into consideration that the (ucp) step may be transformed into a (cp) step followed by a (gc) step, so one possible way to transform $\xrightarrow{ucp}\xrightarrow{no,a}$ is to a sequence $\xrightarrow{no,cp}\xrightarrow{gc}\xrightarrow{no,a}$.

An example of a commuting diagram that involves such extra steps (in this case generated by diagram G_2) is as follows, based on diagram A:

$$\begin{array}{ll}
\texttt{letrec}\ x = (\texttt{letrec}\ y = c\ \overrightarrow{t}\ \texttt{in}\ y)\ \texttt{in case}\ x\ (c\ \overrightarrow{z} \to s) & \xrightarrow{ucp} \\
\texttt{letrec}\ x = c\ \overrightarrow{t}\ \texttt{in case}\ x\ (c\ \overrightarrow{z} \to s) & \xrightarrow{no,case} \\
\texttt{letrec}\ x = c\ \overrightarrow{u}, \overrightarrow{u} = \overrightarrow{t}\ \texttt{in letrec}\ \overrightarrow{z} = \overrightarrow{u}\ \texttt{in}\ s &
\end{array}$$

$$\begin{array}{ll}
\texttt{letrec}\ x = (\texttt{letrec}\ y = c\ \overrightarrow{t}\ \texttt{in}\ y)\ \texttt{in case}\ x\ (c\ \overrightarrow{z} \to s) & \xrightarrow{no,lll} \\
\texttt{letrec}\ x = y, y = c\ \overrightarrow{t}\ \texttt{in case}\ x\ (c\ \overrightarrow{z} \to s) & \xrightarrow{no,case} \\
\texttt{letrec}\ x = y, y = c\ \overrightarrow{u}, \overrightarrow{u} = \overrightarrow{t}\ \texttt{in letrec}\ \overrightarrow{z} = \overrightarrow{u}\ \texttt{in}\ s & \xrightarrow{ucp} \\
\texttt{letrec}\ x = c\ \overrightarrow{u}, \overrightarrow{u} = \overrightarrow{t}\ \texttt{in letrec}\ \overrightarrow{z} = \overrightarrow{u}\ \texttt{in}\ s &
\end{array}$$

After some discussion and preliminary lemmas we present a complete set of commuting diagrams for ucp/gc in Lemma 4.25.

We introduce the following notation: we use $\xrightarrow{no,lll/cp?,*}$ denote a sequence of the form $\xrightarrow{no,lll,*}\xrightarrow{no,cp?}\xrightarrow{no,lll,*}$, i.e. a sequence of 0 or more (lll) steps and at most one (cp) step. The sequence may be empty.

Lemma 4.24 below allows us to construct commuting diagrams for (gc) and (ucp).

Lemma 4.24. *If* $s_1 \xrightarrow{ucp/gc} s_2$ *and an arbitrary number of generating diagrams takes place, then the resulting sequence is as follows: there exists* s_3 *s.t.* $s_1 \xrightarrow{no,lll/cp?,*} s_3 \xrightarrow{ucp/gc} s_2$.

Proof. The proof is straightforward by examining the generating diagrams G_1, G_2, and G_3. Note that the only diagram that generates a (cp) step is G_3 that changes a (ucp) step into a (gc) step. Thus there may be at most one (cp) step. □

The following example combines a forking diagram F with the generating diagram G_2:

$$
\begin{array}{lr}
\texttt{case}\ (\texttt{letrec}\ x = c\ \overrightarrow{t}\ \texttt{in}\ x)\ ((c\ \overrightarrow{y}) \to s) & \xrightarrow{ucp} \\
\texttt{case}\ c\ \overrightarrow{t}\ ((c\ \overrightarrow{y}) \to s) & \xrightarrow{no,case-c} \\
\texttt{letrec}\ \overrightarrow{y} = \overrightarrow{t}\ \texttt{in}\ s &
\end{array}
$$

$$
\begin{array}{lr}
\texttt{case}\ (\texttt{letrec}\ x = c\ \overrightarrow{t}\ \texttt{in}\ x)\ ((c\ \overrightarrow{y}) \to s) & \xrightarrow{no,lcase} \\
\texttt{letrec}\ x = c\ \overrightarrow{t}\ \texttt{in case}\ x\ ((c\ \overrightarrow{y}) \to s) & \xrightarrow{no,case-in} \\
\texttt{letrec}\ x = c\ \overrightarrow{z}, \overrightarrow{z} = \overrightarrow{t}\ \texttt{in letrec}\ \overrightarrow{y} = \overrightarrow{z}\ \texttt{in}\ s & \xrightarrow{llet} \\
\texttt{letrec}\ x = c\ \overrightarrow{z}, \overrightarrow{z} = \overrightarrow{t}, \overrightarrow{y} = \overrightarrow{z}\ \texttt{in}\ s & \xrightarrow{gc1} \\
\texttt{letrec}\ \overrightarrow{z} = \overrightarrow{t}, \overrightarrow{y} = \overrightarrow{z}\ \texttt{in}\ s & \xrightarrow{ve1,*} \\
\texttt{letrec}\ \overrightarrow{z} = \overrightarrow{t}\ \texttt{in}\ s[z_1/y_1, \ldots, z_n/y_n] &
\end{array}
$$

Lemma 4.25. *The complete set of commuting diagrams for ucp/gc is as follows:*

$$\begin{tikzcd}
\cdot \arrow[r, "ucp/gc"] \arrow[d, dashed, "{no,lll/cp_?,*}"'] & \cdot \arrow[dd, "{no,a}"] \\
\cdot \arrow[d, dashed, "{no,a}"'] & \\
\cdot \arrow[r, dashed, "ucp/gc"] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "ucp/gc"] \arrow[d, dashed, "{no,lll/cp_?,*}"'] & \cdot \arrow[ddl, "{no,a}"] \\
\cdot \arrow[d, dashed, "{no,a}"'] & \\
\cdot &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[rr, "ucp/gc"] \arrow[d, dashed, "{no,lll/cp_?,*}"'] & & \cdot \arrow[dd, "{no,case}"] \\
\cdot \arrow[d, dashed, "{no,case}"'] & & \\
\cdot \arrow[d, dashed, "{no,llet}"'] & & \cdot \arrow[d, "{no,llet?}"] \\
\cdot \arrow[r, dashed, "gc"] & \cdot \arrow[r, dashed, "{ve1,*}"] & \cdot
\end{tikzcd}$$

Proof. The following is a graphical representation of the proof for each of the diagrams. Note that there are two ways of obtaining the first commuting diagram: one is by using diagrams A (gc, ucp) or E (ucp), the other one is by using diagram D (ucp).

$$\begin{tikzcd}
\cdot \arrow[r, "ucp/gc"] \arrow[d, dashed, "{lll/cp_?,*}"'] & \cdot \arrow[dd, "{no,a}"] \\
\cdot \arrow[ur, dashed, "ucp/gc"'] \arrow[d, dashed, "{no,a}"'] \arrow[r, phantom, "2"] & \\
\cdot \arrow[r, dashed, "ucp/gc"] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "ucp/gc"] \arrow[d, dashed, "{lll/cp_?,*}"'] & \cdot \arrow[dd, "{no,lll}"] \\
\cdot \arrow[ur, dashed, "ucp/gc"'] \arrow[d, dashed, "{no,lll,+}"'] \arrow[r, phantom, "3"] & \\
\cdot \arrow[r, dashed, "ucp/gc"] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "ucp/gc"] \arrow[d, dashed, "{lll/cp_?,*}"'] & \cdot \arrow[ddl, "{no,a}"] \\
\cdot \arrow[ur, dashed, "ucp/gc"'] \arrow[d, dashed, "{no,a}"'] & \\
\cdot &
\end{tikzcd}$$

$$\begin{tikzcd}
\cdot \arrow[rr, "ucp/gc"] \arrow[d, dashed, "{lll/cp_?,*}"'] & & \cdot \arrow[dd, "{no,case}"] \\
\cdot \arrow[urr, dashed, "ucp/gc"'] \arrow[d, dashed, "{no,case}"'] & 5 & \\
\cdot \arrow[d, dashed, "{no,llet}"'] & & \cdot \arrow[d, "{no,llet?}"] \\
\cdot \arrow[r, dashed, "gc"] & \cdot \arrow[r, dashed, "{ve1,*}"] & \cdot
\end{tikzcd}$$

The diagrams are completed using the following facts:

- 1 - by Lemma 4.24,

– 2 - by diagrams A for (gc) and (ucp) and E for (ucp),
– 3 - by diagram D for (ucp),
– 4 - by diagrams B for (gc) and (ucp),
– 5 - by diagram F for (ucp). □

Before we prove correctness of ucp/gc, we need to prove properties of WHNF with respect to these transformations. The needed property for (gc) is proven in Lemma 4.26, the corresponding property for (ucp) is proven in Lemma 4.27

Lemma 4.26. *If $s \xrightarrow{gc} t$ then:*

– *if s is a WHNF then t is a WHNF.*
– *if t is a WHNF then either s is a WHNF, or $s \xrightarrow{no,lll} s' \xrightarrow{gc?} s''$, and s', s'' are WHNFs.*

Proof. Suppose $s \xrightarrow{gc} t$ and s is a WHNF. If s is a value then t is also a value. If $s = \texttt{letrec}\ Env\ \texttt{in}\ v$ then either $t = \texttt{letrec}\ Env'\ \texttt{in}\ v$ or $t = v$. If $s = \texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^n, Env\ \texttt{in}\ x_n$ then $t = \texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^n, Env'\ \texttt{in}\ x_n$ or $t = \texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^n, Env'\ \texttt{in}\ x_n$ since x and variables along the chain are referenced and thus cannot be removed by (gc). In all of these cases t is a WHNF.
Suppose t is a WHNF. Cases when s is also a WHNF are listed in the other direction of the proof. The remaining cases are shown below. For simplicity we assume that (gc) removes only a single binding, the case of multiple bindings is analogous). We show the completion of the diagram in the first case (below).

$$\begin{aligned} s =&\ \texttt{letrec}\ Env\ \texttt{in}\ (\texttt{letrec}\ x = r\ \texttt{in}\ v) \quad \xrightarrow{gc} \\ t =&\ \texttt{letrec}\ Env\ \texttt{in}\ v \end{aligned}$$

$$\begin{aligned} s =&\ \texttt{letrec}\ Env\ \texttt{in}\ (\texttt{letrec}\ x = r\ \texttt{in}\ v) \xrightarrow{llet-in} \\ s' =&\ \texttt{letrec}\ Env, x = r\ \texttt{in} \quad \xrightarrow{gc} \\ s'' =&\ \texttt{letrec}\ Env\ \texttt{in}\ v \end{aligned}$$

The remaining cases are listed below and are completed in a similar manner by a (llet-in) and a (gc) steps so that the result of the (llet-in) step is a WHNF. We assume that y is not referenced in the body of the corresponding `letrec`.

$$\begin{aligned} s =&\ \texttt{letrec}\ x = r\ \texttt{in}\ (\texttt{letrec}\ Env\ \texttt{in}\ v) \\ s =&\ \texttt{letrec}\ y = r\ \texttt{in}\ \texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^n, Env\ \texttt{in}\ x_n \\ s =&\ \texttt{letrec}\ x_1 = (\texttt{letrec}\ y = r\ \texttt{in}\ (c\ \overrightarrow{t})), \{x_i = x_{i-1}\}_{i=2}^n, Env\ \texttt{in}\ x_n \\ s =&\ \texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^n, Env\ \texttt{in}\ (\texttt{letrec}\ y = r\ \texttt{in}\ x_n) \\ s =&\ \texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_2^{m-1}, (\texttt{letrec}\ y = r\ \texttt{in}\ x_m), \\ &\quad \{x_i = x_{i-1}\}_{m+1}^n, Env\ \texttt{in}\ x_n \end{aligned}$$

□

Lemma 4.27. *If* $s \xrightarrow{ucp} t$ *then:*

– *if* s *is a WHNF then* t *is a WHNF.*
– *if* t *is a WHNF then either* s *is a WHNF, or* $s \xrightarrow{no,llet?}\xrightarrow{no,cp?} s' \xrightarrow{ucp/gc} t$ *and* s' *is a WHNF.*

The following diagram summarizes lemma 4.27:

$$\begin{tikzcd}
s \arrow[rr, "ucp"'] \arrow[d, "no{,}lll?"'] & & t(WHNF) \\
\cdot \arrow[d, "no{,}cp?"'] & & \\
s'(WHNF) \arrow[rruu, "ucp/gc"'] & &
\end{tikzcd}$$

Proof. Let s be a WHNF. A (ucp) step may transform case 1 of definition of a WHNF (a value) into case 1, case 2 into case 1 or case 2, and case 3 into cases 1,2, or 3. There are no other possibilities.
Let t be a WHNF. Possibilities when s is not a WHNF include:

– $t = \texttt{letrec}\ Env\ \texttt{in}\ c\ \overrightarrow{t}$

$$\begin{array}{l} s = \texttt{letrec}\ x = c\ \overrightarrow{t}\ \texttt{in}\ (\texttt{letrec}\ Env\ \texttt{in}\ x) \xrightarrow{lll} \\ s' = \texttt{letrec}\ x = c\ \overrightarrow{t}, Env\ \texttt{in}\ x \xrightarrow{ucp} \texttt{letrec}\ Env\ \texttt{in}\ c\ \overrightarrow{t} = t \end{array}$$

– $t = \texttt{letrec}\ Env\ \texttt{in}\ c\ \overrightarrow{t}$

$$\begin{array}{l} s = \texttt{letrec}\ Env\ \texttt{in}\ (\texttt{letrec}\ x = c\ \overrightarrow{t}\ \texttt{in}\ x) \xrightarrow{lll} \\ s' = \texttt{letrec}\ Env, x = c\ \overrightarrow{t}\ \texttt{in}\ x \xrightarrow{ucp} \texttt{letrec}\ Env\ \texttt{in}\ c\ \overrightarrow{t} = t \end{array}$$

– $t = \texttt{letrec}\ Env\ \texttt{in}\ \lambda y.r$

$$\begin{array}{l} s = \texttt{letrec}\ x = \lambda y.r\ \texttt{in}\ (\texttt{letrec}\ Env\ \texttt{in}\ x) \xrightarrow{lll} \\ \texttt{letrec}\ x = \lambda y.r, Env\ \texttt{in}\ x \xrightarrow{cp} \\ s' = \texttt{letrec}\ x = \lambda y.r, Env\ \texttt{in}\ \lambda y.r \xrightarrow{gc} \texttt{letrec}\ Env\ \texttt{in}\ \lambda y.r = t \end{array}$$

– $t = \texttt{letrec}\ Env\ \texttt{in}\ \lambda y.r$

$$\begin{array}{l} s = \texttt{letrec}\ Env, x = \lambda y.r\ \texttt{in}\ x \xrightarrow{cp} \\ s' = \texttt{letrec}\ Env, x = \lambda y.r\ \texttt{in}\ \lambda y.r \xrightarrow{gc} \\ \texttt{letrec}\ Env\ \texttt{in}\ \lambda y.r = t \end{array}$$

– $t = \texttt{letrec}\ x_1 = c\overrightarrow{t}, \{x_i = x_{i-1}\}_{i=2}^{m}\ \texttt{in}\ x_m$.

$$\begin{array}{l} s = \texttt{letrec}\ x_1 = (\texttt{letrec}\ y = c\overrightarrow{t}\ \texttt{in}\ y)\{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ x_m \xrightarrow{lll} \\ s' = \texttt{letrec}\ x_1 = y, y = c\overrightarrow{t}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ x_m \xrightarrow{ucp} \\ \texttt{letrec}\ x_1 = c\overrightarrow{t}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ x_m = t \end{array}$$

– $t = \texttt{letrec}\ x_1 = c\,\overrightarrow{t}, \{x_i = x_{i-1}\}_{i=2}^{m}\ \texttt{in}\ x_m$. The following situation may occur anywhere in a variable chain.

$$\begin{aligned}&s = \texttt{letrec}\ x_1 = c\,\overrightarrow{t}, x_2 = (\texttt{letrec}\ y = x_1\ \texttt{in}\ y), \{x_i = x_{i-1}\}_{i=3}^{m}\ \texttt{in}\ x_m \xrightarrow{lll}\\ &s' = \texttt{letrec}\ x_1 = c\,\overrightarrow{t}, x_2 = y, y = x_1, \{x_i = x_{i-1}\}_{i=3}^{m}\ \texttt{in}\ x_m \xrightarrow{ucp}\\ &\texttt{letrec}\ x_1 = c\,\overrightarrow{t}, \{x_i = x_{i-1}\}_{i=2}^{m}\ \texttt{in}\ x_m = t\end{aligned}$$

– $t = \texttt{letrec}\ x_1 = c\,\overrightarrow{t}, \{x_i = x_{i-1}\}_{i=2}^{m}\ \texttt{in}\ x_m$.

$$\begin{aligned}&s = \texttt{letrec}\ x_1 = c\,\overrightarrow{t}, \{x_i = x_{i-1}\}_{i=2}^{m}\ \texttt{in}\ (\texttt{letrec}\ y = x_m\ \texttt{in}\ y) \xrightarrow{lll}\\ &s' = \texttt{letrec}\ x_1 = c\,\overrightarrow{t}, \{x_i = x_{i-1}\}_{i=2}^{m}, y = x_m\ \texttt{in}\ y \xrightarrow{ucp}\\ &\texttt{letrec}\ x_1 = c\,\overrightarrow{t}, \{x_i = x_{i-1}\}_{i=2}^{m}\ \texttt{in}\ x_m = t\end{aligned}$$

All of the above cases follow the pattern stated in the lemma. □

Proposition 4.28. *If $s \xrightarrow{ucp/gc} t$ then $s \sim_c t$.*

Proof. Let $s \xrightarrow{ucp/gc} t$. We show that, for any context R, given a normal order reduction sequence $R[s] \xrightarrow{no,*} s'$, where s' is a WHNF, we can transform the sequence $R[t] \xleftarrow{ucp/gc} R[s] \xrightarrow{no,*} s'$ into $R[t] \xrightarrow{no,*} t' \xleftarrow{ucp/gc,*} s'$. By Lemma 4.2 $R[s] \xrightarrow{ucp/gc} R[t]$ in a surface context, thus the commuting and forking diagrams are applicable.
We consider forking diagrams given in the beginning of the section. We define a measure similar to the one used in the proof of Proposition 4.18: the measure is an ordered (left-to-right) sequence of counters for each segment, where a segment is either a ucp/gc step (note that there may be at most one) or a contiguous sequence of (ve1) steps. Each counter is the total number of normal order steps to the right of each the segment. The sequences of counters ordered lexicographically form the ordered sequence measure defined in Definition 4.3.
For each forking diagram the measure strictly decreases. Note that there are no (ve1) segments to the left of a ucp/gc steps, thus the counter for a (ve1) segment does not change when a ucp/gc step is moved.

– Diagrams A for (ucp) and (gc), E for (ucp): $\xleftarrow{ucp/gc} \xrightarrow{no,a} \rightsquigarrow \xrightarrow{no,a} \xleftarrow{ucp/gc}$. In this case the count decreases for the (ucp) step since it get switched with a normal order step.
– Diagrams B for (ucp) and (gc): $\xleftarrow{ucp/gc} \xrightarrow{no,a} \rightsquigarrow \xrightarrow{no,a}$. In this case one of the ucp/gc steps is removed, the counters for the other ones are left unchanged, so the measure decreases.
– Diagrams C for (ucp) and (gc): $\xleftarrow{ucp/gc} \xrightarrow{no,a} \rightsquigarrow \xleftarrow{ucp/gc}$. A normal order step is removed so the count for the resulting ucp/gc step is 1 less than for the given one.
– Diagram D for (ucp): $\xleftarrow{ucp/gc} \xrightarrow{no,lll,+} \rightsquigarrow \xrightarrow{no,lll,*} \xleftarrow{ucp/gc}$. Since all (lll) steps were moved to the left of the ucp/gc step, the measure decreases by the number of given (lll) steps.

– Diagram F for (ucp): $\xleftarrow{ucp} \xrightarrow{no,case} \xrightarrow{no,lll?} \rightsquigarrow \xrightarrow{no,case} \xrightarrow{no,lll} \xleftarrow{gc} \xleftarrow{ve1,*}$. The counter for the new (gc) step and for the (ve1) segment is at least 1 less than the counter for the given (ucp) step.
– By Lemma 4.14 switching a (ve1) segment with a normal order reduction step decreases the counter for the segment and does not change any other counter.

Since the measure decreases with every transformation step, by Lemma 4.4 the process terminates. We have constructed a sequence $R[t] \xrightarrow{no,*} t' \xleftarrow{ucp/gc} s'$. Since s' is a WHNF, by Lemmas 4.26, 4.27, and 4.12 t' is also a WHNF. Thus, we have shown that $R[t]$ has a normal order sequence that leads to a WHNF, so $s \leq_c t$.
For the other direction of the lemma let $s \xrightarrow{ucp/gc} t$ and suppose $R[t]$ evaluates to WHNF.
We show that, given a normal order sequence reduction $R[t] \xrightarrow{no,*} t'$, where t' is a WHNF, we can transform the sequence $R[s] \xrightarrow{ucp/gc} R[t] \xrightarrow{no,*} t'$ into a sequence $R[s] \xrightarrow{no,*} s' \xrightarrow{ucp/gc} t'$ using commutative diagrams in Lemma 4.25. The measure is similar to the other proof direction: each ucp/gc step has a counter equal to the total number of normal order steps to its right; the ordered sequences of such counters are ordered lexicographically.
The case analysis below shows that the measure decreases with every application of a commuting diagram.

1. $\xrightarrow{ucp/gc} \xrightarrow{no,a} \rightsquigarrow \xrightarrow{lll/cp?,*} \xrightarrow{no,a} \xrightarrow{ucp/gc}$ - the counter for the (ucp/gc) is decreased by 1.
2. $\xrightarrow{ucp/gc} \xrightarrow{no,a} \rightsquigarrow \xrightarrow{lll/cp?,*} \xrightarrow{no,a} \rightsquigarrow \xrightarrow{lll/cp,*} \xrightarrow{no,a}$ - the resulting sequence does not have a ucp/gc reduction so the counters appear only on (ve1) segments to the right of the transformed sequence. Since these counters are all smaller than the counter on ucp/gc before the transformation, the measure decreases.
3. $\xrightarrow{ucp} \xrightarrow{no,case} \xrightarrow{no,lll?} \rightsquigarrow \xrightarrow{lll/cp?,*} \xrightarrow{no,case} \xrightarrow{no,lll} \xrightarrow{gc} \xleftarrow{ve1,*}$. In this case a (ucp) step is replaced by a (gc) step and possibly a (ve1) segment. However, each of these steps has a smaller counter than the original (ucp) step.
4. By Lemma 4.14 switching a (ve1) segment with any normal order step decreases the measure.

We have shown that we can transform a sequence $R[s] \xrightarrow{ucp/gc} R[t] \xrightarrow{no,*} t'$, where t' is a WHNF, into a sequence $R[s] \xrightarrow{no,*} s' \xrightarrow{ucp/gc} t'$. By Lemmas 4.26, 4.27, and 4.12 there exists a WHNF s'' s.t. $s' \xrightarrow{lll/cp?,*} s'' \xrightarrow{ucp/gc} s''' \xrightarrow{ve1,*} t'$. Thus we have constructed a terminating sequence for $R[s]$ so by the Context Lemma 3.2 $t \leq_c s$.
Combining the two direction of the proof, we conclude that $s \sim_c t$. □

Observe that the rule (abs2) defined in Figure 3 is is just a sequence of reverse (ucp) steps: if $s \xrightarrow{abs2} t$ then $s \xleftarrow{ucp,*} t$. Note that since (abs2) introduces fresh variables x_i, each of these variables appears only once, hence the condition for (ucp) is satisfied. Thus the following result holds:

Proposition 4.29. *If $s \xrightarrow{abs2} t$ then $s \sim_c t$.*

The (abs2) rule is needed for conversion to a simpler calculus L_S in Section 5.

4.13 Property of (choice)

Proposition 4.30. *If $s \xrightarrow{choice} t$ then $t \leq_c s$.*

Proof. Without loss of generality assume that (choice-l) takes place, (choice-r) is completely analogous. By the Context Lemma 3.2 it is sufficient to consider the reduction in a reduction context R. Suppose $s = R[(\text{choice } s_1\ s_2)]$ and $t = R[s_1]$. By the labeling algorithm the normal order step from s is a choice reduction, which results in either $R[s_1]$ (if (choice-l) takes place) or $R[s_2]$ (if (choice-r) takes place). It trivially follows that if $R[s_1]{\downarrow}$ then $R[(\text{choice } s_1\ s_2){\downarrow}$ since $R[(\text{choice } s_1\ s_2)] \xrightarrow{choice-l} R[s_1]$.
By the Context Lemma 3.2 this implies that for any context C: $C[s_1]{\downarrow} \Rightarrow C[(\text{choice } s_1\ s_2)]{\downarrow}$ ◻

Note that the converse of the claim is not true: if $R[(\text{choice } s_1\ s_2){\downarrow}$, it is possible that the convergence path is $R[(\text{choice } s_1\ s_2)] \xrightarrow{choice-r} R[s_2]\ldots$, and $R[s_1]$ does not converge. However, the following obvious property holds for any reduction context R:

Lemma 4.31. *If $R[(\text{choice } s_1\ s_2)]{\downarrow}$ then at least one of $R[s_1]$ or $R[s_2]$ converges.*

Remark 4.32. Note that the lemma above does not hold in the general case. I.e. $C[(\text{choice } s_1\ s_2)]{\downarrow}$, but neither $C[s_1]{\downarrow}$ nor $C[s_2]{\downarrow}$. The expression

```
letrec  y = λx.choice True False
in if (y True) then ⊥ else
          (if (y True) then True else ⊥ )
```

is an example for such a behaviour.

4.14 Property of *Ω*.

In the following, we will use the technical observation that during a normal-order reduction of t, we can trace the bindings $x_i = r_i$ of a closed subexpression $s =$ (`letrec` $x_1 = s_1, \ldots, x_n = s_n$ `in` s') of t, if r occurs on the surface of t. The application of this observation allows to draw several nice and important conclusions. This technique is also used in a later chapter for the approximation. An $\mathcal{LR}$-context is defined as $\mathcal{LR} ::= [\cdot] \mid$ (`letrec` *Env* `in` $\mathcal{LR}$).

Proposition 4.33. *The expression Ω is the least element w.r.t. $\leq_c$, and for every closed expression s with $s{\Uparrow}$, the equation $s \sim_c \Omega$ holds.*

Proof. Due to the Context Lemma 3.2, it is sufficient to show that for all reduction contexts R and all closed expression s with $s\Uparrow$, we have $R[s]\Uparrow$. Assume otherwise, that $R[s]\downarrow$. We show that this is impossible. In the normal-order reduction corresponding to $R[s]\downarrow$, we distinguish the descendents of s and of R by labeling the descendents of s with †. The components are of two kinds: bindings $x_1 = s_1, \ldots, x_n = s_n$, and an expression s', where the expression s' is a descendent of s. It is in a reduction context or in an $\mathcal{LR}$-context, such that it is in a reduction context again after some (no,lll)-reductions. The cases of the labeling and the labeling operations are as follows, where we use an exponent notation to indicate the †-label.

- $R[(\texttt{letrec}\ x_1 = r_1, \ldots, x_n = r_n\ \texttt{in}\ u))^\dagger] \to R[(\texttt{letrec}\ x_1^\dagger = r_1^\dagger, \ldots, x_n^\dagger = r_n^\dagger\ \texttt{in}\ u^\dagger))^\dagger]$.
- $R[((\lambda x.r)\ u)^\dagger] \xrightarrow{no} R[(\texttt{letrec}\ x^\dagger = u^\dagger\ \texttt{in}\ r^\dagger)$.
  In the case that the (lbeta)-redex itself is in an environment, i.e if $R = (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ R'[\cdot])$, a subsequent (no,lll,*) achieves that u is again in a reduction context in the expression $(\texttt{letrec}\ \mathit{Env}, x_1^\dagger = r_1^\dagger, \ldots, x_n^\dagger = r_n^\dagger\ \texttt{in}\ R'[u^\dagger])$.
- $R[\texttt{case}\ (c\ u_1\ \ldots u_n)\ ((c\ x_1 \ldots x_n)\ \to\ r) \ldots] \xrightarrow{no} R[(\texttt{letrec}\ x_1^\dagger = u_1^\dagger, \ldots, x_n^\dagger = u_n^\dagger\ \texttt{in}\ r^\dagger)]$. Here the same remarks as above are valid.
- A (cp)-reduction $(\texttt{letrec}\ y = r^\dagger, \mathit{Env}\ \texttt{in}\ C[y]) \to (\texttt{letrec}\ y = r^\dagger, \mathit{Env}\ \texttt{in}\ C[r])$ does not copy the label into $C[r]$, only if the subterm y is labeled as an expression, then we have:
  $(\texttt{letrec}\ y = r^\dagger, \mathit{Env}\ \texttt{in}\ C[y^\dagger]) \to (\texttt{letrec}\ y = r^\dagger, \mathit{Env}\ \texttt{in}\ C[r^\dagger])$.

Note that a (llet)-reduction may produce a mixture of s-components and descendents of R in a common letrec-environment. Moreover, since the initial s is closed, it cannot happen that there are free variables in an †-labeled expression that are not themselves labeled with †. For every intermediate expression in the reduction, a complete s-descendent can be generated as $(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ s_{n+1})$, where $x_i = s_i$ are the bindings derived from s (i.e. the †-labeled ones), and s_{n+1} is the "in"-expression derived from s that is in a reduction context, and also †-labeled. Up to some (lll)-reductions, the normal-order-reduction of $R[s]\downarrow$ generates a reduction sequence of s, where all reductions are in surface contexts. The reduction of $R[s]\downarrow$ can only stop with a WHNF, if also the reduction sequence of s stops with a WHNF. The standardization theorem 4.35 then shows that $s\downarrow$, which is a contradiction. $\square$

Let (cpbot) be the transformations:

$$\begin{array}{ll}(\texttt{letrec}\ x = \Omega, \mathit{Env}\ \texttt{in}\ C[x]) & \to (\texttt{letrec}\ x = \Omega, \mathit{Env}\ \texttt{in}\ C[\Omega]) \\ (\texttt{letrec}\ x = \Omega, y = C[x], \mathit{Env}\ \texttt{in}\ r) & \to (\texttt{letrec}\ x = \Omega, y = C[\Omega], \mathit{Env}\ \texttt{in}\ r)\end{array}$$

It is permitted to copy closed bot-expressions:

Proposition 4.34. *The transformation (cpbot) is correct.*

Proof. If $t \xrightarrow{cpbot} t'$, then it is easy to see that the two reductions commute with normal-order reductions until a WHNF is reached for both terms, or in each expression, x is in a reduction context. In the latter case, both expressions do not have a WHNF, by Proposition 4.33. Since t, t' have the same convergence behavior in all contexts, we have $t \sim_c t'$. $\square$

4.15 Standardization

We summarize correctness of transformations and decreasing property of (choice) in the *standardization* result. This result is technically similar to standardization for deterministic calculi, where standardization means that if t can be evaluated to a WHNF using any reductions, then also normal-order can evaluate t. In a deterministic calculus, the reduction is unique, so it can be interpreted as a standardized evaluation, whereas in non-deterministic calculi, the meaning is that reduction sequences can be standardized. We formulate it only as convergence, but it also holds in the more general form that $t \xrightarrow{*} t'$ implies that there exists t'' s.t. $t \xrightarrow{no,*} t'' \xleftrightarrow{*} t'$ where the sequence $t'' \xleftrightarrow{*} t'$ contains only correct non-normal-order steps and correct transformations (in either direction), and thus $t'' \sim_c t'$. However, this is not required for the further results in this paper.

Theorem 4.35 (Standardization). *If $t \xrightarrow{*} t'$ where t' is a WHNF and the sequence $\xrightarrow{*}$ consists of any reduction from L in figures 1 and 2 and of transformation steps from figure 3, or (cpbot), then $t\downarrow$.*

Proof. Let $t \xrightarrow{*} t'$ be a sequence of reductions and transformations as above, where t' is a WHNF. By Lemmas and Propositions 4.1, 4.6, 4.7, 4.10, 4.13, 4.16, 4.18, 4.20, 4.28, 4.34 and 4.29, for every single reduction step $t_i \to t_{i+1}$, which are not choice-reductions, the relation $t_i \sim_c t_{i+1}$ holds.
By Proposition 4.30, an intermediate (choice)-reduction $t_i \to t_{i+1}$ decreases the terms, i.e. $t_i \geq_c t_{i+1}$ holds. Since $\leq_c$ is transitive, we obtain $t' \leq_c t$. Since $t'\downarrow$, this immediately implies $t'\downarrow$. $\square$

4.16 On Reduction Lengths

The following length property is used in further proofs. It states that forking diagrams for (cpcx), (cpS), (ve) do not increase the length of a normal-order-reduction:

Proposition 4.36. *Given a normal-order reduction $s\ U\ s'$, where s' is a WHNF, and a closed expression t such that $s \xrightarrow{a,\mathcal{S}} t$, with $a \in \{$(cpcx), (cpS), (ve)$\}$. Then there exists a normal-order reduction sequence U' of t to a WHNF, i.e. $t\ U'\ t'$ such that U' is not longer than U.*

Proof. This follows from the diagrams in Section 3.1, where (cpcx) requires $\xrightarrow{ve1,*}$, $\xleftarrow{ve1,*}$, and $\xrightarrow{abs1,*}$, which in turn requires $\xleftarrow{ve1,*}$. Scanning all these diagrams shows by induction on the length of U that U' can be constructed, such that U' does not require more normal-order reductions. □

5 A Simpler Calculus

We define a simpler calculus L_S that is used to produce a set of values for any closed expression. It is formulated such that a so-called *pre-evaluation* can be defined and shown to be a correct tool to prove contextual preorder and contextual equivalence of expressions in almost the same way as the simulation method would do it.
We will show that the two calculi L, L_S are equivalent. The calculus L_S does not use variable-binding chains for reduction steps, and permits a generalized copy-reduction that also may copy expression of the form $(c\ x_1 \ldots x_n)$, where x_i are variables. These expressions are called *cv-expression* in the following.

The rules of the calculus L_S are defined in figure 4, which can be applied in any context. We use labels indicating the normal order redex, where T means the top-term, S means a subterm reduction, V means visited. The labelng algorithm (unwind) starts with t^T, where no subexpression of t is labeled, and uses the following rules exhaustively. The labeling rules can be used in any context.

$$\begin{array}{ll}
(s\ t)^{S \vee T} & \to (s^S\ t)^V \\
(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ t)^T & \to (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ t^S)^V \\
(\texttt{letrec}\ x = s, \mathit{Env}\ \texttt{in}\ C[x^S]) & \to (\texttt{letrec}\ x = s^S, \mathit{Env}\ \texttt{in}\ C[x^V]) \\
(\texttt{letrec}\ x = s, y = C[x^S], \mathit{Env}\ \texttt{in}\ r) & \to (\texttt{letrec}\ x = s^S, y = C[x^V], \mathit{Env}\ \texttt{in}\ r) \\
 & \qquad \text{if } C \neq [.] \\
(\texttt{seq}\ s\ t)^{S \vee T} & \to (\texttt{seq}\ s^S\ t)^V \\
(\texttt{case}\ s\ \mathit{alts})^{S \vee T} & \to (\texttt{case}\ s^S\ \mathit{alts})^V
\end{array}$$

An L_S-*WHNF* is defined as an abstraction or a cv-expression, or an expression $(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ v)$, where v is an abstraction or a cv-expression.
It is easy to see that for every L_S-WHNF is also an L-WHNF, a nd that for every L-WHNF t, there is a reduction to a an L_S-WHNF: $t \xrightarrow{no,*} t'$ using only $(L_S, (abs))$, $(L_S, (lll))$, and $(L_S, (cp))$-reductions.

Theorem 5.1. *Let s be an expression. Then $s{\downarrow}_{L_S} \iff s{\downarrow}_L$.*

Proof. One direction follows from the standardization theorem 4.35 as follows: Let s be a term and let U_{L_S} be an L_S-evaluation of s. All reduction steps of L_S appear in L, either as reduction steps or as transformations (see below), with the only exception of L_S (cp-in) and (cp-e) reductions for the case when the copied value is a *cv*-expression. In this case the L_S reduction is represented in L as the following sequence (shown for (cp-in), completely analogous for (cp-e)):

$$\begin{array}{ll}
\texttt{letrec}\ x = (c\ x_1, \ldots, x_n), \mathit{Env}\ \texttt{in}\ C[x] & \xrightarrow{cpcx} \\
\texttt{letrec}\ x = (c\ y_1, \ldots, y_n), y_1 = x_1, \ldots, y_n = x_n, \mathit{Env}\ \texttt{in}\ C[(c\ y_1, \ldots, y_n)] & \xrightarrow{ve,*} \\
\texttt{letrec}\ x = (c\ x_1, \ldots, x_n), \mathit{Env}\ \texttt{in}\ C[(c\ x_1, \ldots, x_n)] &
\end{array}$$

(lbeta)	$((\lambda x.s)^S\ r) \to (\texttt{letrec}\ x = r\ \texttt{in}\ s)$
(cp-in)	$(\texttt{letrec}\ x = v^S, \mathit{Env}\ \ \texttt{in}\ C[x^V])$
	$\to (\texttt{letrec}\ x = v, \mathit{Env}\ \ \texttt{in}\ C[v])$
	where v is an abstraction or a cv-expression
(cp-e)	$(\texttt{letrec}\ x = v^S, \mathit{Env}, y = C[x^V]\ \texttt{in}\ r)$
	$\to (\texttt{letrec}\ x = v, \mathit{Env}, y = C[v]\ \texttt{in}\ r)$
	where v is an abstraction or a cv-expression
(abs)	$(c\ t_1 \dots t_n)^{S \vee T} \to (\texttt{letrec}\ x_1 = t_1, \dots, x_n = t_n\ \texttt{in}\ (c\ x_1 \dots x_n))$
	if $(c\ t_1 \dots t_n)$ is not a cv-expression
(llet-in)	$(\texttt{letrec}\ \mathit{Env}_1\ \texttt{in}\ (\texttt{letrec}\ \mathit{Env}_2\ \texttt{in}\ r)^S)$
	$\to (\texttt{letrec}\ \mathit{Env}_1, \mathit{Env}_2\ \texttt{in}\ r)$
(llet-e)	$(\texttt{letrec}\ \mathit{Env}_1, x = (\texttt{letrec}\ \mathit{Env}_2\ \texttt{in}\ s_x)^S\ \texttt{in}\ r)$
	$\to (\texttt{letrec}\ \mathit{Env}_1, \mathit{Env}_2, x = s_x\ \texttt{in}\ r)$
(lapp)	$((\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ t)^S\ s) \to (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (t\ s))$
(lcase)	$(\texttt{case}_T\ (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ t)^S\ \mathit{alts}) \to (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (\texttt{case}_T\ t\ \mathit{alts}))$
(seq-c)	$(\texttt{seq}\ v^S\ t) \to t$ if v is a value
(lseq)	$(\texttt{seq}\ (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ s)^S\ t) \to (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (\texttt{seq}\ s\ t))$
(case)	$(\texttt{case}\ (c\ t_1 \dots t_n)^S \dots ((c\ y_1 \dots y_n) \to s) \dots)$
	$\to\ (\texttt{letrec}\ y_1 = t_1, \dots, y_n = t_n\ \texttt{in}\ s)$
(choice-l)	$(\texttt{choice}\ s\ t)^{S \vee T} \to s$
(choice-r)	$(\texttt{choice}\ s\ t)^{S \vee T} \to t$

Fig. 4. Reduction rules of L_S

By the standardization theorem 4.35 all reduction and transformation steps in L preserve evaluation to an L-WHNF. Thus if $s{\downarrow}_{L_S}$ then $s{\downarrow}_L$.

For the other direction let U be a normal order L reduction for s. We show by induction on $\mathit{len}(U)$ that there exists a normal order L_S-reduction U_{L_S} for s.

For the base case let $\mathit{len}(U) = 0$. Then s is a L-WHNF, which can be reduced to an L_S-WHNF in some steps as observed above.
If the first reduction is an (lll)-reduction, then this is also a normal-order L_S-reduction.

Now let $s_1 \xrightarrow{no,L,a} s_2$ be the first reduction of U', where a is not an (lll)-reduction. If $a \in \{(\text{lbeta}), (\text{case-c}), (\text{seq-c}), (\text{choice})\}$ then $\xrightarrow{no,L,a}$ is also a normal order reduction for L_S. Using the induction hypothesis we derive the demanded normal-order L_S-reduction-sequence.

Otherwise, i.e. if the L-reduction exploits bindings over several variable-variable-bindings, the diagrams in figure 5 show how a normal-order L_S-reduction sequence can be derived for the first reduction. Then Proposition 4.36 shows that there is a normal-order reduction of length not greater than the length of a normal-order reduction of s', since the corresponding right and downward reduction sequences are covered by the proposition. □

$x = (c\ t_1\ t_2)$
`in case` $x\ \ldots$

$$\begin{tikzcd}
s_1 \arrow[r, "{no,L,case-e}"] \arrow[d, "{no,L_S,abs}"'] & s_2 \arrow[dd, "{S,cpcx,*}"] \\
\cdot \arrow[d, "{no,L_S,lll}"'] & \\
\cdot \arrow[d, "{no,L_S,cp,+}"'] & \cdot \arrow[ddl, "{S,ve,*}"] \\
\cdot \arrow[d, "{no,L_S,case}"'] & \\
s' &
\end{tikzcd}$$

$x = (c\ x_1\ x_2)$
`in case` $x\ \ldots$

$$\begin{tikzcd}
s_1 \arrow[r, "{no,L,case-e}"] \arrow[d, "{no,L_S,cp,+}"'] & s_2 \arrow[d, "{S,cpcx,*}"] \\
\cdot \arrow[d, "{no,L_S,case}"'] & \cdot \arrow[dl, "{S,ve,*}"] \\
s' &
\end{tikzcd}$$

$x = (c\ t_1\ t_2)$
`in seq` $x \ldots$

$$\begin{tikzcd}
s_1 \arrow[r, "{no,L,seq-e}"] \arrow[d, "{no,L_S,abs}"'] & s_2 \cdot \arrow[ddddl, "{S,cpcx,*}"] \\
\cdot \arrow[d, "{no,L_S,lll}"'] & \\
\cdot \arrow[d, "{no,L_S,cp,+}"'] & \\
\cdot \arrow[d, "{no,L_S,seq}"'] & \\
s' &
\end{tikzcd}$$

$x = (c\ x_1\ x_2)$
`in seq` $x\ \ldots$

$$\begin{tikzcd}
s_1 \arrow[r, "{no,L,seq-e}"] \arrow[d, "{no,L_S,cp,+}"'] & s_2 \arrow[ddl, "{S,cpcx,*}"] \\
\cdot \arrow[d, "{no,L_S,seq}"'] & \\
s' &
\end{tikzcd}$$

$x = (\lambda x.t)$
`in seq` $x\ \ldots$

$$\begin{tikzcd}
s_1 \arrow[r, "{no,L,seq-e}"] \arrow[d, "{no,L_S,cp,+}"'] & s_2 \arrow[ddl, "{S,cp\mathcal{S},+}"] \\
\cdot \arrow[d, "{no,L_S,seq}"'] & \\
s' &
\end{tikzcd}$$

$x = \lambda z.t, y = x$
`in` $(y\ s) \ldots$
where $a \in \{(lbeta), (seq)\}$

$$\begin{tikzcd}
s_1 \arrow[r, "{no,L,cp}"] \arrow[d, "{no,L_S,cp,*}"'] & s_2 \arrow[dl, "{S,cp\mathcal{S},*}"] \\
s' &
\end{tikzcd}$$

Fig. 5. Diagrams for converting L-reductions into L_S-reductions

Remark 5.2. We include the prototypical examples for all the diagrams in figure 5 in the same sequence as the diagrams:

`letrec` $x = (c\ t_1\ t_2), y = x$ `in case` $y\ (c\ y_1\ y_2) \to s$ $\xrightarrow{no,L,case-e}$
`letrec` $x = (c\ z_1\ z_2), z_1 = t_1, z_2 = t_2, y = x$
    `in letrec` $y_1 = z_1, y_2 = z_2$ `in` s $\xrightarrow{L,(cpcx)}$
`letrec` $x = (c\ u_1\ u_2), u_1 = z_1, u_2 = z_2, z_1 = t_1, z_2 = t_2, y = (c\ u_1\ u_2)$
    `in letrec` $y_1 = z_1, y_2 = z_2$ `in` s $\xrightarrow{L,(ve),*}$
`letrec` $x = (c\ z_1\ z_2), z_1 = t_1, z_2 = t_2, y = (c\ u_1\ u_2)$
    `in letrec` $y_1 = z_1, y_2 = z_2$ `in` s

$\texttt{letrec}\ x = (c\ t_1\ t_2), y = x\ \texttt{in case}\ y\ (c\ y_1\ y_2) \rightarrow s \quad \xrightarrow{no,L_S,abs}$

$\xrightarrow{no,L_S,\mathrm{llet}}$

$\texttt{letrec}\ x = (c\ z_1\ z_2), z_1 = t_1, z_2 = t_2, y = x\ \texttt{in case}\ y\ (c\ y_1\ y_2) \rightarrow s \quad \xrightarrow{no,L_S,cp}$

$\texttt{letrec}\ x = (c\ z_1\ z_2), z_1 = t_1, z_2 = t_2, y = (c\ z_1\ z_2)$

$\qquad \texttt{in case}\ y\ (cy_1\ y_2) \rightarrow s \quad \xrightarrow{no,L_S,cp}$

$\texttt{letrec}\ x = (c\ z_1\ z_2), z_1 = t_1, z_2 = t_2, y = (c\ z_1\ z_2)$

$\qquad \texttt{in case}\ (c\ z_1\ z_2)\ (c\ y_1\ y_2) \rightarrow s \quad \xrightarrow{no,L_S,case}$

$\texttt{letrec}\ x = (c\ z_1\ z_2), z_1 = t_1, z_2 = t_2, y = (c\ z_1\ z_2)$

$\qquad \texttt{in letrec}\ y_1 = z_1, y_2 = z_2\ \texttt{in}\ s$

$\texttt{letrec}\ x = (c\ x_1\ x_2), y = x\ \texttt{in case}\ y\ (c\ y_1\ y_2) \rightarrow s \quad \xrightarrow{no,L,case-e}$

$\texttt{letrec}\ x = (c\ z_1\ z_2), z_1 = x_1, z_2 = x_2, y = x$

$\qquad \texttt{in letrec}\ y_1 = z_1, y_2 = z_2\ \texttt{in}\ s \quad \xrightarrow{L,(cpcx)}$

$\texttt{letrec}\ x = (c\ u_1\ u_2), u_1 = z_1, u_2 = z_2, z_1 = x_1, z_2 = x_2, y = (c\ u_1\ u_2)$

$\qquad \texttt{in letrec}\ y_1 = z_1, y_2 = z_2\ \texttt{in}\ s \quad \xrightarrow{L,(ve),*}$

$\texttt{letrec}\ x = (c\ x_1\ x_2), y = (c\ x_1\ x_2)$

$\qquad \texttt{in letrec}\ y_1 = x_1, y_2 = x_2\ \texttt{in}\ s$

$\texttt{letrec}\ x = (c\ x_1\ x_2), y = x\ \texttt{in case}\ y\ (c\ y_1\ y_2) \rightarrow s \quad \xrightarrow{no,L_S,cp}$

$\texttt{letrec}\ x = (c\ x_1\ x_2), y = (c\ x_1\ x_2)$

$\qquad \texttt{in case}\ y\ (cy_1\ y_2) \rightarrow s \quad \xrightarrow{no,L_S,cp}$

$\texttt{letrec}\ x = (c\ x_1\ x_2), y = (c\ x_1\ x_2)$

$\qquad \texttt{in case}\ (c\ x_1\ x_2)\ (c\ y_1\ y_2) \rightarrow s \quad \xrightarrow{no,L_S,case}$

$\texttt{letrec}\ x = (c\ x_1\ x_2), y = (c\ x_1\ x_2)$

$\qquad \texttt{in letrec}\ y_1 = x_1, y_2 = x_2\ \texttt{in}\ s$

$\texttt{letrec}\ x = (c\ t_1\ t_2), y = x\ \texttt{in seq}\ y\ s \quad \xrightarrow{no,L,seq-e}$

$\texttt{letrec}\ x = (c\ t_1\ t_2), y = x\ \texttt{in}\ s \quad \xrightarrow{L,(cpcx)}$

$\texttt{letrec}\ x = (c\ x_1\ x_2), x_1 = t_1, x_2 = t_2, y = (c\ x_1\ x_2)\ \texttt{in}\ s$

$\texttt{letrec}\ x = (c\ t_1\ t_2), y = x\ \texttt{in seq}\ y\ s \quad \xrightarrow{no,L_S,abs}$

$\xrightarrow{no,L_S,\mathrm{llet}}$

$\texttt{letrec}\ x = (c\ x_1\ x_2), x_1 = t_1, x_2 = t_2, y = x$

$\qquad \texttt{in seq}\ y\ s \quad \xrightarrow{no,L_S,cp}$

$\texttt{letrec}\ x = (c\ x_1\ x_2), x_1 = t_1, x_2 = t_2, y = (c\ x_1\ x_2)$

$\qquad \texttt{in seq}\ y\ s \quad \xrightarrow{no,L_S,cp}$

$\texttt{letrec}\ x = (c\ x_1\ x_2), x_1 = t_1, x_2 = t_2, y = (c\ x_1\ x_2)$

$\qquad \texttt{in seq}\ (c\ x_1\ x_2)\ s \quad \xrightarrow{no,L_S,seq}$

$\texttt{letrec}\ x = (c\ x_1\ x_2), x_1 = t_1, x_2 = t_2, y = (c\ x_1\ x_2)\ \texttt{in}\ s$

$$\begin{array}{ll}
\texttt{letrec}\ x = (c\ x_1\ x_2), y = x\ \texttt{in seq}\ y\ s & \xrightarrow{no,L,seq-e} \\
\texttt{letrec}\ x = (c\ x_1\ x_2), y = x\ \texttt{in}\ s & \xrightarrow{cpS} \\
\texttt{letrec}\ x = (c\ x_1\ x_2), y = (c\ x_1\ x_2)\ \texttt{in}\ s & \\
\hline
\texttt{letrec}\ x = (c\ x_1\ x_2), y = x\ \texttt{in seq}\ y\ s & \xrightarrow{no,L_S,cp} \\
\texttt{letrec}\ x = (c\ x_1\ x_2), y = (c\ x_1\ x_2)\ \texttt{in seq}\ y\ s & \xrightarrow{no,L_S,cp} \\
\texttt{letrec}\ x = (c\ x_1\ x_2), y = (c\ x_1\ x_2)\ \texttt{in seq}\ (c\ x_1\ x_2)\ s & \xrightarrow{no,L_S,seq} \\
\texttt{letrec}\ x = (c\ x_1\ x_2), y = (c\ x_1\ x_2)\ \texttt{in}\ s & \\
\hline\hline
\end{array}$$

$$\begin{array}{ll}
\texttt{letrec}\ x = \lambda u.t, y = x\ \texttt{in seq}\ y\ s & \xrightarrow{no,L,seq-e} \\
\texttt{letrec}\ x = \lambda u.t, y = x\ \texttt{in}\ s & \xrightarrow{L,cpS} \\
\texttt{letrec}\ x = \lambda u.t, y = \lambda u.t\ \texttt{in}\ s & \\
\hline
\texttt{letrec}\ x = \lambda u.t, y = x\ \texttt{in seq}\ y\ s & \xrightarrow{no,L_S,cp} \\
\texttt{letrec}\ x = \lambda u.t, y = \lambda u.t\ \texttt{in seq}\ y\ s & \xrightarrow{no,L_S,cp} \\
\texttt{letrec}\ x = \lambda u.t, y = \lambda u.t\ \texttt{in seq}\ \lambda u.t\ s & \xrightarrow{no,L_S,seq} \\
\texttt{letrec}\ x = \lambda u.t, y = \lambda u.t\ \texttt{in}\ s & \\
\hline\hline
\end{array}$$

$$\begin{array}{ll}
\texttt{letrec}\ x = \lambda u.t, y = x\ \texttt{in}\ y\ s & \xrightarrow{no,L,cp} \\
\texttt{letrec}\ x = \lambda u.t, y = x\ \texttt{in}\ (\lambda u.t)\ s & \xrightarrow{no,L,lbeta} \\
\texttt{letrec}\ x = \lambda u.t, y = x\ \texttt{in letrec}\ \ u = s\ \texttt{in}\ t & \xrightarrow{L,cpS} \\
\texttt{letrec}\ x = \lambda u.t, y = \lambda u.t\ \texttt{in letrec}\ \ u = s\ \texttt{in}\ t & \\
\hline
\texttt{letrec}\ x = \lambda u.t, y = x\ \texttt{in}\ y\ s & \xrightarrow{no,L_S,cp} \\
\texttt{letrec}\ x = \lambda u.t, y = \lambda u.t\ \texttt{in}\ y\ s & \xrightarrow{no,L_S,cp} \\
\texttt{letrec}\ x = \lambda u.t, y = \lambda u.t\ \texttt{in}\ (\lambda u.t)\ s & \xrightarrow{no,L_S,lbeta} \\
\texttt{letrec}\ x = \lambda u.t, y = \lambda u.t\ \texttt{in letrec}\ u = s\ \texttt{in}\ t & \\
\hline\hline
\end{array}$$

Theorem 5.1 implies that the observational semantics of L and L_S completely agree:

Corollary 5.3. *Let s, t be expressions. Then $s \leq_{c,L} t \ \Leftrightarrow\ s \leq_{c,L_S} t$.*

Proof. This follows from Theorem 5.1 since for any context C it holds that $C[s]{\downarrow_L} \Leftrightarrow C[s]{\downarrow_{L_S}}$ as well as $C[t]{\downarrow_L} \Leftrightarrow C[t]{\downarrow_{L_S}}$. □

By Proposition 4.33, the expression Ω is the least element w.r.t. $\leq_c$, and for every closed expression s with $s{\Uparrow}$, the equation $s \sim_c \Omega$ holds, which holds by Corollary 5.3 also for the calculus L_S.

6 Pre-Evaluation of Expressions

In the following, we will again use the technical observation that during a normal-order reduction of t, we can trace the bindings $x_i = r_i$ of a closed subexpression

$r = (\texttt{letrec}\ x_1 = r_1, \ldots, x_n = r_n\ \texttt{in}\ s')$ of t, if r occurs on the surface of t. The application of this observation allows to draw several nice and important conclusions.
We will use evaluation in L_S to reduce closed expressions in all possible ways, where reduction takes place in surface contexts. The intention is to have a means to compare closed expressions by a set of results, even perhaps an infinite set. We use the additional constant $\circledcirc$ (called stop) in order to indicate stopped reductions. Its semantical value is $\perp$, but it is clearer if there is a notational distinction between them. One may also think of $\circledcirc$ as a synonym to Ω. This allows us to extend the notion of $\leq_c$ to include $\circledcirc$.

Definition 6.1. *A* pseudo-value *is an expression built from $\circledcirc$, constructors, and abstractions. A* answer *is an expression built from $\circledcirc$, constructors, and abstractions which is not the constant $\circledcirc$ itself, i.e. a pseudo-value is an answer or $\circledcirc$.*

We show the intention of the pre-evaluation by an example. The idea of pre-evaluation is to first obtain by L_S-normal order reduction all possible WHNFs, and then to apply normal-order reductions locally to the bindings. Since this does in general not terminate, we stop the reduction at any point and then fill the results into the in-expression: The bindings that are cv-expressions or abstractions are copied sufficiently often into the in-expression. Due to recursive bindings, this may also be a non-terminating process that has to stopped. Then we strip away the top letrec-environment and replace the occurrences of the bound variables by $\circledcirc$.

Example 6.2. The expression $(\texttt{letrec}\ x = \lambda y.\texttt{True}\ \texttt{in}\ x)$ has $\lambda y.\texttt{True}$ as resulting answer.
The expression $(\texttt{letrec}\ x = (\texttt{Cons True}\ x)\ \texttt{in}\ x)$ has the following resulting answers: $(\texttt{Cons}\ \circledcirc\ \circledcirc)$, $(\texttt{Cons True}\ \circledcirc)$, $(\texttt{Cons}\ \circledcirc\ (\texttt{Cons True}\ \circledcirc))$, $(\texttt{Cons True}\ (\texttt{Cons True}\ \circledcirc))$, $\ldots$.

The approximation reduction $\xrightarrow{A}$ is a slight extension of the L_S-reduction for the pre-evaluation and defined as follows:

Definition 6.3. *Let s be a closed expression. We define the approximation reduction $\xrightarrow{A}$as follows:*
$s \xrightarrow{A} v$ holds for some closed answer v iff there is a reduction from $(\texttt{letrec}\ x = s\ \texttt{in}\ x)$ to v using the following intermediate steps, in the given order:

1. *$(\texttt{letrec}\ x = s\ \texttt{in}\ x) \xrightarrow{*} s'$ using an L_S-evaluation to a WHNF s'. From s' we perform any number of L_S-reductions in application surface contexts (non-deterministically), where the target variables of (cp) are also in application surface contexts.*
2. *Perform any number of copy-reductions into the "in"-expression. Here the target variable of (cp) may be in any context C.*

3. *The last step is to remove the top-letrec-environment, and to replace all remaining let-bound variables in the "in"-expression by $\circledcirc$. The resulting expression is now either $\circledcirc$ or one of the desired answers v.*

The set of answers reachable from s by this procedure is defined as $ans(s)$.

Lemma 6.4. *Let s be closed expression and $v \in ans(s)$. Then $v \leq_c s$.*

Proof. This follows from the correctness of the transformations proved in the previous sections, from decreasingness of (choice) (see Proposition 4.30) and from the fact that $\bot \sim_c \Omega$ is the least element w.r.t. $\leq_c$ (see Proposition 4.33). □

Now we prove that sufficiently many answers are reached by these reduction possibilities.

Theorem 6.5. *Let R be a reduction context, s be a closed expression such that $R[s]\downarrow$. Then there is an answer v with $(\texttt{letrec}\ x = s\ \texttt{in}\ x) \xrightarrow{A} v$, such that $R[v]\downarrow$. Note that $s \sim_c (\texttt{letrec}\ x = s\ \texttt{in}\ x)$ (see Proposition 4.28).*

Proof. For the proof we always refer to the calculus L_S unless mentioned otherwise.
Let R be a reduction context and s be a closed expression. Let *Red* be a normal-order reduction of $R[(\texttt{letrec}\ x = s\ \texttt{in}\ x)] \xrightarrow{no} r_1 \xrightarrow{no} \ldots \xrightarrow{no} r_n$, where r_n is a WHNF, and n is the number of normal-order reductions. We use a similar labeling technique as in the proof of Proposition 4.33. In every expression of *Red*, the bindings inherited from $x = s$ can be identified in every r_i by labeling them with †. Thus we label letrec-bound variables and the bound expressions in surface positions that are derived from s. However, in contrast to the proof of Proposition 4.33, we do not label the in-expression, only the bindings. An important invariant is that for all †-labeled bindings $y_i = a_i$, and all free variables y in a_i, y is also a †-labeled variable, which follows by induction on the length of the reduction from the fact that s is closed. If a WHNF w of $R[(\texttt{letrec}\ x = s\ \texttt{in}\ x)]$ is reached, then from the WHNF we can gather all the labeled bindings in the top level letrec environment of w, and construct the expression $s' := (\texttt{letrec}\ Env\ \texttt{in}\ x)$, where we denote $x_1 = s_1, \ldots, x_m = s_m$ by *Env* and where $x_1 = x$ for convenience. Now we compute one possible answer v from s' as required by our claim as follows. We perform n of the following macro-copy-steps within the environment *Env* into the "in"-expression:
One step consists of replacing all occurrences of x_i by s_i in the "in"-expression (initially x) for all $x_i = s_i$ in *Env* s.t. s_i is an abstraction or a cv-expression. We do this in parallel for every letrec-bound variable, which is the same as applying the substitution σ formed from *Env*. This is repeated $n + 1$ times. The last step is to remove the top-environment, and to replace all letrec-bound variables in the in-expression by $\circledcirc$. This may produce either $\circledcirc$, or the desired answer v, and we have $s' \xrightarrow{A} v$ according to Definition 6.3. Since we assumed that a WHNF is reached, and s was in a reduction context before, it is not possible that only $\circledcirc$ is reached, since the initial variable x was in a reduction context and there must

be at least one normal-order copy into x. Thus at least one of the macro-copy steps will replace x by a constructor-expression or an abstraction.

Now we have to show that $R[v]\downarrow$. We start with the normal-order reduction *Red* of $R[(\texttt{letrec}\ x = s\ \texttt{in}\ x)]$. This reduction *Red* can be rearranged, such that all the reductions that are within the †-labeled *Env* are performed before all other reductions, i.e., $R[(\texttt{letrec}\ x = s\ \texttt{in}\ x)] \xrightarrow{*} R[s'] \xrightarrow{no,*} r_n$. Note that $R[(\texttt{letrec}\ x = s\ \texttt{in}\ x)] \xrightarrow{*} R[s']$ is in general not a normal-order-reduction, however, it is easy to see that the first part of the reduction of $(\texttt{letrec}\ x = s\ \texttt{in}\ x)$ is an L_S-normal-order reduction to a WHNF, since x is "demanded" first. The subsequent reductions remain in application surface positions. The reduction $R[s'] \xrightarrow{no,*} r_n$ is normal-order, and has length at most n. Now we argue why this is true: We view the reduction sequence *Red* as a mixture of reduction steps within †-labeled components, or reduction steps that modify the non-†-labeled components. All reductions are in surface contexts. Hence the †-reductions can be shifted to the left over non-†-reductions, since they are independent. The arguments are: the only interactions between the †-labeled expressions and the rest are (lll)-reductions, or (cp)-reductions of †-labeled values into other parts of the expression. Further invariants are: all †-labeled bindings in s' are at application surface positions, and it is not possible to copy non-†-labeled values into terms within a labeled binding.

Now we focus on the reduction sequence $R[s'] \xrightarrow{no,*} r_n$ of length at most n. We have to show that for $s' \xrightarrow{*} v$, we also have $R[v] \xrightarrow{no,*} u$, where u is a WHNF. The term v and its descendents can be represented using $\phi_{\geq k}$, which is defined as follows: $\phi_{\geq k}(r)$ denotes r modified by the following operations: first k applications of σ are performed (the substitution corresponding to the s-environment *Env*), then any number of (cp)-steps using *Env* and variables in r as target variables, and as a final step $[\circledcirc/x_i]$-replacements in r for all let-bound variables in *Env*. Now we have to show that $(\phi_{\geq n+1} R[s'])\downarrow$. This can be done by first computing forking diagrams, which are as follows:

$$\begin{tikzcd}
\cdot \arrow[r, "\phi_{\geq k}"] \arrow[d, "no{,}a"] & \cdot \arrow[d, dashed, "(\{(no{,}a)\cup(abs)\cup(lll)\cup(cp)\cup(cpbot)\cup(ve)\}{,}*)"] \\
\cdot \arrow[r, dashed, "\phi_{\geq k-1}"] & \cdot
\end{tikzcd}$$

We show the prototypical (non-trivial) examples for the diagram above:

$$\begin{tikzcd}
C[(x\ a)] \arrow[r, "\phi_{\geq k}"] \arrow[d, "no{,}cp"] & C[(v_i\ a)] \arrow[d, equal] \\
C[(s_i\ a)] \arrow[r, dashed, "\phi_{\geq k-1}"] & C[(v_i\ a)]
\end{tikzcd}$$

$$\begin{tikzcd}
C[(\texttt{letrec}\ y = c\ y_1\ y_2\ \texttt{in}\ D[y])] \arrow[r, "\phi_{\geq k}"] \arrow[d, "no{,}cp"] & C[(\texttt{letrec}\ y = c\ v_1\ v_2\ \texttt{in}\ D[y])] \arrow[d, dashed, "\{(abs)\cup(lll)\cup(no{,}cp)\cup(cp)\cup(cpbot)\cup(ve)\}{,}*"'] \\
C\begin{bmatrix}\texttt{letrec}\ y = c\ y_1\ y_2 \\ \texttt{in}\ D[c\ y_1\ y_2]\end{bmatrix} \arrow[r, dashed, "\phi_{\geq k-1}"] & C\begin{bmatrix}\texttt{letrec}\ y = c\ v_1\ v_2 \\ \texttt{in}\ D[c\ v_1\ v_2]\end{bmatrix}
\end{tikzcd}$$

Using the diagrams, it is easily shown by induction that, finally, we obtain a WHNF that is the result of a macro-copy reduction, using $\phi_{\geq i}$, where $i \geq 1$. The argument now is that the replaced positions do not contribute to the WHNF w, hence it remains a WHNF after applying $\phi_{\geq 1}$. This means there is a reduction sequence $R[v] \xrightarrow{*} w'$, where w' is a WHNF. Finally, the standardization theorem 4.35 shows that $R[v]\downarrow$.

□

7 Least Upper Bounds and Sets of Answers

In the following we use the approximation calculus.

7.1 On Contextual least Upper Bounds

In the following we use the calculus L_S.

Definition 7.1. *Let W be a set of expressions, and let t be an expression. Then t is a* lub *of W iff $\forall u \in W : u \leq_c t$, and for every s with $\forall u \in W : u \leq_c s$, it is $t \leq_c s$.*
The expression t is called a contextual lub (club) *of W, iff for all contexts C: $C[t]$ is a lub of $\{C[r] \mid r \in W\}$. The notation is $t \in club(W)$.*
An expression t is called a linear club (lclub) *of W, if the set W is a $\leq_c$-ascending chain of expressions. The notation is $t \in lclub(W)$. The set of all t such that $t \in lclub(A)$ for some $A \subseteq W$ is denoted as $sublclub(W)$.*

We write $C[W]$ for the set $\{C[r] \mid r \in W\}$ and a context C in the following.

Proposition 7.2. *The following are equivalent:*

1. *t is a club of W:*
2. *$\forall u \in W : u \leq_c t$ and for every context C: if $C[t]\downarrow$, then there is some $u \in W$, such that $C[u]\downarrow$.*

Proof. 1 $\implies$ 2:
Let t be a club of W, and let C be a context such that $C[t]\downarrow$, but for all $u \in W$: $C[u]\downarrow$ is false. If for some $u \in W : C[u]$ is open, or if $C[t]$ is open, then we instantiate all free variables in $C[t]$, $C[u]$ for $u \in W$ by Ω, say σ is this substitution, and obtain the same properties. But then Ω is the club of $\sigma(C[W])$, which is a contradiction to $\sigma(C[t])\downarrow$.
2 $\implies$ 1:
Let C, D be contexts. We will show that $D[t] = lub((D[W]))$. Let r be an expression with $\forall u \in W : D[u] \leq_c r$. The assumption implies that if $CD[t]\downarrow$, then there exists a $u \in W$ with $CD[u]\downarrow$. Since $D[u] \leq_c r$ we have also $C[r]\downarrow$. Hence $CD[t]\downarrow \Rightarrow C[r]\downarrow$. Since this holds for all contexts C, we have proved $D[t] \leq_c r$. This implies for all contexts $D : D[t] = lub(D[W])$, hence t is a club of W. □

Example 7.3. The following ascending chain $\lambda x_1.\Omega$, $\lambda x_1, x_2.\Omega, \ldots \lambda x_1, \ldots, x_n.\Omega$ has YK as lclub, which is equivalent to the value $\lambda x.(Y\ K)$.

Lemma 7.4. *Let s be a closed expression. Then $s \sim_c \Omega$ if and only if $ans(s) = \emptyset$.*

Proof. If $s \sim_c \Omega$ then obviously $ans(s) = \emptyset$. On the other hand, if $s \not\sim_c \Omega$, then $s\downarrow$, hence there is some WHNF w wth $s \xrightarrow{*} w$, and from a WHNF, we can construct at least one v with $s \xrightarrow{A} v$. □

Lemma 7.5. *Let t be a closed expression with $t \not\sim_c \Omega$. Then $t \in club(ans(t))$.*

Proof. For $v \in ans(t) \neq \emptyset$, we have $v \leq_c t$ by Lemma 6.4. Assume there is some r with $v \leq_c r$ for all $v \in ans(t)$. We have to show that $t \leq_c r$: Therefore let R be a reduction context with $R[t]\downarrow$. By Theorem 6.5, there is some answer $v \in ans(t)$, such that $R[v]\downarrow$. Hence $R[r]\downarrow$, since $v \leq_c r$. This means that $R[t]\downarrow \implies R[r]\downarrow$, hence by the context lemma we have $t \leq_c r$. □

This yields an immediate criterion for contextual preorder:

Corollary 7.6. *Let s, t be closed expressions. If for all $w \in ans(s)$, we also have $w \leq_c t$, then $s \leq_c t$.*

Now we are left with the task to devise a method for showing that $w \leq_c t$ for an answer w.

Definition 7.7. *Let W be a set of closed expressions. The set $cl_{lclub_\omega}(W)$ is defined as the least set that contains all its linear clubs of ascending chains, and contains W. If $sublclub(W) = W$, then we say the set W is closed w.r.t. linear clubs.*

Proposition 7.8. *Let w be a closed answer and t be a closed expression. If $w \leq_c r$ for some $r \in cl_{lclub_\omega}(ans(t))$, then $w \leq_c t$.*

Proof. Let r_i be an ascending chain with $r_i \leq_c t$ for all i, let r be the club of the chain r_i, and let $w \leq_c r$. Then $r \leq_c t$ by the definition of club. Transitivity of $\leq_c$ implies $w \leq_c t$. □

Another criterion is:

Theorem 7.9. *Let s, t be closed expressions. If for all $v \in ans(s)$ there is some $w \in sublclub(ans(t))$ with $v \leq_c w$, then $s \leq_c t$.*

Proof. This follows from Corollary 7.6 and Proposition 7.8. □

However, note that a simplistic subset-condition is insufficient to show $s \leq_c t$. We provide an example similar to one in [Man05], where, however no recursive let is available.

Example 7.10. Let

$$\begin{aligned} s &= \lambda x.Y\ K \\ f\ z &= \texttt{choice}\ z\ (\texttt{letrec}\ u = f\ z\ \texttt{in}\ \lambda x.u) \\ t &= f\ \Omega \end{aligned}$$

The function f can be given explictely using standard methods as $f = Y\ (\lambda g.\lambda z.\texttt{choice}\ \Omega\ (\texttt{letrec}\ u = g\ z\ \texttt{in}\ \lambda x.u))$
Then for every $v \in ans(t)$, we have $v <_c s$. However, it is easy to see that $s \sim_c t$, since $\lambda x.Y\ K$ is the club of the ascending chain of values in $ans(t)$.

Example 7.11. Let s, t be expressions defined as

$$\begin{aligned} s &= repeat\ \texttt{True} \\ repeat &= Y\ (\lambda r.\lambda x.\texttt{Cons}\ x\ (r\ x)) \\ t &= Y\ (\lambda a.\texttt{choice}\ \Omega\ (\texttt{Cons True}\ a) \end{aligned}$$

Then s and t have comparable sets of answers: all approximations of the infinite list ($\texttt{Cons True}$ ($\texttt{Cons True}$ $\ldots$)).

We also require some tools for comparing answers:
The properties of the contextual preorder show that for constructor expressions, we have

Proposition 7.12. *$(c\ s_1 \ldots s_n) \leq_c (c\ t_1 \ldots t_n)$ is equivalent to $s_i \leq_c t_i$ for all i.*

which together with $\Omega \leq_c t$ for all t is a strong criterion for answers consisting only of constructors, variables, and $\circledcirc$.

8 Criteria for Abstractions

Besides the trivial method to compare two abstractions $\lambda x.s$ and $\lambda x.t$ by α-equivalence, perhaps combined with other correct transformations, we give a stronger condition for $\lambda x.s \leq_c \lambda x.t$ that is based on applying the abstractions to all possible pseudo-value arguments instead of using the criterium for all contexts.
First we have to extend the answer-method to open expressions.

Lemma 8.1. *[Context lemma for Closing Reduction Contexts] Let s, t be expressions. Then $s \leq_c t$ iff for all reduction contexts R: if $R[s], R[t]$ are closed and $R[s]\downarrow$, then also $R[t]\downarrow$,*

Proof. We only have to show the if-direction. Let R be a reduction context such that $R[s]\downarrow$. Let $R[s] \to r_1 \to \ldots \to r_n$ be a normal-order reduction where r_n is a WHNF. It is no restriction to assume that all free variables of s and also of t are captured by R: If not, we can add enough bindings $x = y$ to the top level environment of R, which is a correct transformation since (ve) and (gc) are correct transformations. If R doesn't have a top level letrec-environment, then we add one consisting of these bindings. The modifications do not change the property of being a reduction context. Then the reduction rules and the definition of normal-order imply that for any substitution σ into the free variables of $R[s]$, the reduction $\sigma(R[s]) \to \sigma(r_1) \to \ldots \to \sigma(r_n)$ is also a normal-order reduction to a WHNF. We choose σ such that all free variables are mapped to Ω. The context $\sigma(R')$ is the desired reduction context. Since R' captures al variables of s, t, the expressions $R'[s], R'[t]$ are closed and $R'[s]\downarrow$, hence by assumptions also $R'[t]\downarrow$, Easy inspection shows that this can only be the case, if also $R[t]\downarrow$, since Ω can never be in a reduction context in the normal reduction corresponding to $R'[t]\downarrow$. Using the context lemma, we can conclude that $s \leq_c t$. $\square$

A *pseudo-value environment Env* is an enviroment where every bound term is a (closed) pseudo-value.

Proposition 8.2. *Let s, t be two expressions. Then $s \leq_c t$ iff for all pseudo-value environments Env: if* (`letrec` *Env* `in` s), (`letrec` *Env* `in` t) *are closed then* (`letrec` *Env* `in` s) $\leq_c$ (`letrec` *Env* `in` t).

Proof. The only-if direction is obvious.
In order to show the other direction, we will use Lemma 8.1. Let R be a reduction context such that $R[s], R[t]$ are closed and such that $R[s]\downarrow$. It is no restriction to assume that $R[\cdot]$ is of the form (`letrec` Env_1, Env_2 `in` $R'[\cdot]$), where Env_1 binds all the variables in $FV(s,t)$, and (`letrec` Env_1 `in` $[\cdot]$) is closed. Since $s' :=$ (`letrec` Env_1, Env_2 `in` $R'[s]$)$\downarrow$, there is a normal-order reduction *Red* of s'. In the same way as in the proof of Theorem 6.5, we can commute reductions, and obtain a pre-evaluated environment Env_1', such that also $s'' :=$ (`letrec` Env_1', Env_2 `in` $R'[s]$)$\downarrow$. The environment Env_1' will be further modified into an environment Env_1'' as follows: Every binding

$x = r$, where r is not an abstraction and not a cv-expression is changed into $x = \circledcirc$. Again we have $s^{(3)} := (\texttt{letrec}\ \mathit{Env}_1'', \mathit{Env}_2\ \texttt{in}\ R'[s])\downarrow$, since the $\circledcirc$-bindings do not influence the normal-order reduction. Let n be the length of a normal-order reduction of $s^{(3)}$. Then we further modify Env_1'' into $\mathit{Env}_1^{(3)}$ by applying the substitution σ corresponding to Env_1'' at least n times to the environment, and then replacing all remaining occurrences of variables by $\circledcirc$. Similar as in the proof of Theorem 6.5, we argue that $(\texttt{letrec}\ \mathit{Env}_1^{(3)}, \mathit{Env}_2\ \texttt{in}\ R'[s])\downarrow$. Using the knowledge about correct transformations, it can be proved using induction that $(\texttt{letrec}\ \mathit{Env}_1^{(3)}, \mathit{Env}_2\ \texttt{in}\ R'[s]) \sim_c (\texttt{letrec}\ \mathit{Env}_1^{(3)}, \mathit{Env}_2\ \texttt{in}\ R'[(\texttt{letrec}\ \mathit{Env}_1^{(3)}\ \texttt{in}\ s)])$, hence $(\texttt{letrec}\ \mathit{Env}_1^{(3)}, \mathit{Env}_2\ \texttt{in}\ R'[(\texttt{letrec}\ \mathit{Env}_1^{(3)}\ \texttt{in}\ s)])\downarrow$.
Now we argue the reverse way for t: by the assumption, we have $(\texttt{letrec}\ \mathit{Env}_1^{(3)}, \mathit{Env}_2\ \texttt{in}\ R'[(\texttt{letrec}\ \mathit{Env}_1^{(3)}\ \texttt{in}\ s)]) \leq_c (\texttt{letrec}\ \mathit{Env}_1^{(3)}, \mathit{Env}_2\ \texttt{in}\ R'[(\texttt{letrec}\ \mathit{Env}_1^{(3)}\ \texttt{in}\ t)])$, hence $(\texttt{letrec}\ \mathit{Env}_1^{(3)}, \mathit{Env}_2\ \texttt{in}\ R'[(\texttt{letrec}\ \mathit{Env}_1^{(3)}\ \texttt{in}\ t)])\downarrow$. The same argument as above shows that also $(\texttt{letrec}\ \mathit{Env}_1^{(3)}, \mathit{Env}_2\ \texttt{in}\ R'[t])\downarrow$. Since $\circledcirc \sim_c \bot$ is the $\leq_c$-least element, and (cp) does not change the $\sim_c$ equivalence class, we also have $(\texttt{letrec}\ \mathit{Env}_1'', \mathit{Env}_2\ \texttt{in}\ R'[t])\downarrow$. Since $(\texttt{letrec}\ \mathit{Env}_1'', \mathit{Env}_2\ \texttt{in}\ R'[t])$ can be reached from $(\texttt{letrec}\ \mathit{Env}_1, \mathit{Env}_2\ \texttt{in}\ R'[t])$ by reductions that only decrease by $\leq_c$ due to results in Sections 3, we finally have $(\texttt{letrec}\ \mathit{Env}_1, \mathit{Env}_2\ \texttt{in}\ R'[t])\downarrow$. Since the reduction context R was arbitrary, we can apply Lemma 8.1 and obtain that $s \leq_c t$

□

Theorem 8.3. *$\lambda x.s \leq \lambda x.t$ iff for all closed pseudo-values v: $(\lambda x.s)\ v \leq (\lambda x.t)\ v$.*

Proof. Follows from Proposition 8.2, since $(\lambda x.s)\ v \sim_c (\texttt{letrec}\ x = v\ \texttt{in}\ s)$ □

Corollary 8.4. *Let v, w be answers with $v \leq_c w$. Then there are the following two cases:*

1. *$v = c\ s_1 \dots s_n$, $v = c\ t_1 \dots t_n$, and $s_i \leq_c t_i$ for all i.*
2. *$v = \lambda x.v'$, $w = \lambda x.w'$ and Theorem 8.3 is applicable.*

9 Finite Simulation Method and Examples

As proved in the last section, we have several criteria to prove $s \leq_c t$ for closed expressions s, t.

1. If $\mathit{ans}(s) \subseteq \mathit{ans}(t)$, then $s \leq_c t$.
2. If for every $v \in \mathit{ans}(s)$, there is some $w \in \mathit{ans}(t)$ with $v \leq_c w$, then $s \leq_c t$.
3. If for every $v \in \mathit{ans}(s)$, there is some $w \in \mathit{ans}(t)$ with $v \leq_c w$ or some $w \in \mathit{sublclub}(\mathit{ans}(t))$ with $v \leq_c w$, then $s \leq_c t$.

4. if $s = c\ s_1 \dots s_n$, $t = c\ t_1 \dots t_n$, and $s_i \leq_c t_i$ for all i, then $s \leq_c t$.
5. if $s = \lambda x.s'$, $t = \lambda x.t'$, and for all pseudo-values v: $s\ v \leq_c t\ v$, then $s \leq_c t$.

The following (non-effective) procedure is a prototype of "finite simulation" for testing two closed expressions s, t whether they are in a relation $s \leq_c t$:

1. Compute the answer-sets $ans(s)$ and $ans(t)$.
2. For every value $v \in ans(s)$, find a value $w \in ans(t)$ such that $v \leq_c w$.
3. For the $v \leq_c w$-test, use the following tests, recursively:
   (a) If $v = (c\ s_1 \dots s_m)$, $w = (c\ t_1 \dots t_m)$, make sure that $v_i \leq_c w_i$ for all i.
   (b) If $v = \lambda x.s'$, $w = \lambda x.t'$, make sure that for all pseudo-values a:
   $(v\ a) \leq_c (w\ a)$, again using this procedure.

Under the assumption of finiteness of answer-sets, of boundedness of computation depth and decidability of all involved tests, the procedure is effective. Proving $s \sim t$ for expressions s, t can be done by checking $s \leq_c t$ and $t \leq_c s$.

Example 9.1. As an example let

$$\begin{aligned} s &= (\texttt{letrec}\ x = (\texttt{choice}\ \Omega\ (\texttt{Cons}\ 1\ x))\ \texttt{in}\ x) \\ t &= \mathit{repeat}\ 1 \end{aligned}$$

Then s can be reduced to the answers $(\texttt{Cons}\ 1(\texttt{Cons}\ 1 \dots (\texttt{Cons}\ 1\ \Omega)))$, and t can be reduced to the same answers, where we use the equivalence of $\circledcirc$ and Ω. This implies that $s \sim_c t$.

Example 9.2. This example shows that the finite simluation can distinguish expressions that differ only by sharing. Let $s := (\texttt{letrec}\ x = \texttt{choice True False in}\ \lambda y.x)$ and let $t = \lambda y.(\texttt{letrec}\ x = \texttt{choice True False in}\ x)$. These expressions are contextually different, using the context $C[\cdot] := (\texttt{letrec}\ z = [\cdot]\ \texttt{in if}\ (z\ \bot)\ \texttt{then}\ (\texttt{if}\ (z\ \bot)\ \texttt{then True else}\ \bot)\ \texttt{else True})$. The answer-sets are as follows: $ans(t) = \{t\}$, and $ans(s) = \{\lambda y.\texttt{True}, \lambda y.\texttt{False}\}$, which are clearly different.

Example 9.3. Our method allows to show the algebraic laws for `choice`:
For all closed expressions s, r, t, we have

$$\begin{array}{ll} \texttt{choice}\ s\ s & \sim_c s \\ \texttt{choice}\ s\ t & \sim_c \texttt{choice}\ t\ s \\ \texttt{choice}\ (\texttt{choice}\ s\ t)\ r & \sim_c \texttt{choice}\ s\ (\texttt{choice}\ t\ r) \end{array}$$

This follows from the criteria in Corollary 7.6, since the set of answers is the same on the left and right hand side. It is also possible to show that the identities for `choice` hold for all expressions, and hence the identities can be used everywhere as program transformation:
For all expressions s, r, t, the identities above hold.
This can be shown as follows using Proposition 8.2: For every pseudo-value environment Env that closes s and t, we consider $(\texttt{letrec}\ Env\ \texttt{in choice}\ s\ t)$

and (`letrec` *Env* `in` `choice` $t\ s$). The computation of values starts by a normal-order reduction, and obviously, the first step in the normal-order reduction is the choice-reduction. But then the right and left hand side have the same set of answers, hence they are equivalent. The same can be done for the other identities.

10 Conclusion

We have shown that in a call-by-need lambda calclus with letrec, where the proof method of Howe fail to prove correctness of co-inductive simulation, the correctness of finite simulation can be established as a tool with almost the same practical power. It is based on computing all possible answers that can be derived from closed expressions and then comparing the answers. The local approach to compare constructor expressions and abstractions is more or less the same as for similarity.
Further research is to adapt and extend the methods to an appropriately defined simulation, perhaps a proof of the open problem of correctness of simulation in [SSSS04] could be solved, and to investigate an extension of the tools and methods to a combination of may- and must-convergence.

11 Acknowledgements

We thank David Sabel for reading several versions of the paper and for helpful comments.

References

AB02. Zena M. Ariola and Stefan Blom. Skew confluence and the lambda calculus with letrec. *Annals of Pure and Applied Logic*, 117:95–168, 2002.

Abr90. Samson Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

AK96. Z. M. Ariola and Jan Willem Klop. Equational term graph rewriting. *Fundamentae Informaticae*, 26(3,4):207–240, 1996.

AK97. Zena M. Ariola and Jan Willem Klop. Lambda calculus with explicit recursion. *Inform. and Comput.*, 139(2):154–233, 1997.

BN98. Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

Gor99. Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Theoret. Comput. Sci.*, 228(1-2):5–47, October 1999.

Han96. Michael Hanus. A unified computation model for functional and logic programming. In *POPL 97*, pages 80–93. ACM, 1996.

How89. D. Howe. Equality in lazy computation systems. In *4th IEEE Symp. on Logic in Computer Science*, pages 198–203, 1989.

How96. D. Howe. Proving congruence of bisimulation in functional programming languages. *Inform. and Comput.*, 124(2):103–112, 1996.

KSS98. Arne Kutzner and Manfred Schmidt-Schauß. A nondeterministic call-by-need lambda calculus. In *International Conference on Functional Programming 1998*, pages 324–335. ACM Press, 1998.

Mac02. Elena Machkasova. *Computational Soundness of Non-Confluent Calculi with Applications to Modules and Linking.* PhD thesis, Boston University, 2002.

Mac07. Elena Machkasova. Computational soundness of a call by name calculus of recursively-scoped records. In *7th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2007)*, Electron. Notes Theor. Comput. Sci. ENTCS, 2007.

Man04. Matthias Mann. Congruence of bisimulation in a non-deterministic call-by-need lambda calculus. In *Prel. Proc. of the Workshop on Structural Operational Semantics, SOS '04, (London, 2004)*, volume NS-04-1 of *BRICS Notes Series*, pages 20–38, August 2004.

Man05. Matthias Mann. *A Non-Deterministic Call-By-Need Lambda Calculus: Proving Similarity a Precongruence by an Extension of Howe's Method to Sharing.* PhD thesis, Dept. of Computer Science and Mathematics, J.W.Goethe-Universität, Frankfurt, Germany, 2005.

MSC99. Andrew K. D. Moran, David Sands, and Magnus Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, volume 1594 of *Lecture Notes in Comput. Sci.*, pages 85–102. Springer-Verlag, 1999.

MT00. Elena Machkasova and Franklyn A. Turbak. A calculus for link-time compilation. In *ESOP'2000*, volume 1782 of *LNCS*, pages 260–274, 2000.

Pey03. Simon Peyton Jones. *Haskell 98 language and libraries: the Revised Report.* Cambridge University Press, 2003. `www.haskell.org`.

PGF96. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. 23th Principles of Programming Languages*, 1996.

Plo75. Gordon D. Plotkin. Call-by-name, call-by-value, and the lambda-calculus. *Theoret. Comput. Sci.*, 1:125–159, 1975.

SSM07. Manfred Schmidt-Schauß and Matthias Mann. On equivalences and standardization in a non-deterministic call-by-need lambda calculus. Frank report 31, Inst. f. Informatik, J.W.Goethe-University, Frankfurt, August 2007.

SSS06. David Sabel and Manfred Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. Frank report 24, Inst. f. Informatik, J.W.Goethe-University, Frankfurt, January 2006. submitted for publication.

SSS07. David Sabel and Manfred Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Math. Structures Comput. Sci.*, 2007. accepted for publication.

SSSS04. Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. On the safety of Nöcker's strictness analysis. Frank report 19, Inst. f. Informatik, J.W.Goethe-University, Frankfurt, 2004.

SSSS08. Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, pages 00–00, 2008. accepted for publication.

WPK03. J. B. Wells, Detlef Plump, and Fairouz Kamareddine. Diagrams for meaning preservation. In *RTA*, volume 2706 of *LNCS*, pages 88 –106, 2003.