Simulation in the Call-by-Need Lambda-Calculus with letrec

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

[1] Dept. Informatik und Mathematik, Inst. Informatik, J.W. Goethe-University,
PoBox 11 19 32, D-60054 Frankfurt, Germany,
{schauss,sabel}@ki.informatik.uni-frankfurt.de
[2] Division of Science and Mathematics,
University of Minnesota, Morris, MN 56267-2134, U.S.A
elenam@morris.umn.edu

Technical Report Frank-40

Research group for Artificial Intelligence and Software Technology
Institut für Informatik,
Fachbereich Informatik und Mathematik,
Johann Wolfgang Goethe-Universität,
Postfach 11 19 32, D-60054 Frankfurt, Germany

April 6, 2010

Abstract. This paper shows the equivalence of applicative similarity and contextual approximation, and hence also of bisimilarity and contextual equivalence, in the deterministic call-by-need lambda calculus with letrec. Bisimilarity simplifies equivalence proofs in the calculus and opens a way for more convenient correctness proofs for program transformations. Although this property may be a natural one to expect, to the best of our knowledge, this paper is the first one providing a proof. The proof technique is to transfer the contextual approximation into Abramsky's lazy lambda calculus by a fully abstract and surjective translation. This also shows that the natural embedding of Abramsky's lazy lambda calculus into the call-by-need lambda calculus with letrec is an isomorphism between the respective term-models. We show that the equivalence property proven in this paper transfers to a call-by-need letrec calculus developed by Ariola and Felleisen.

1 Introduction

Non-strict programming languages such as the core-language of Haskell can be modeled using call-by-need lambda calculi. Contextual semantics, based on an operational semantics, describes behavior of expressions in all possible contexts and can model the semantics of different variants of these calculi. Applicative

bisimulation is a restricted form of contextual equivalence: if two closed expressions behave the same on all arguments, then they are bisimilar. It allows more convenient proofs of e.g. correctness of program transformations. Abramsky & Ong showed that applicative bisimulation is the same as contextual equivalence in a specific simple lazy lambda calculus [Abr90,AO93], and Howe [How89,How96] proved that in classes of calculi applicative bisimulation is the same as contextual equivalence. This leads to the expectation that some form of applicative bisimulation may be used for calculi with Haskell's cyclic let(rec). Howe's method is applicable to calculi with non-recursive let even in the presence of non-determinism [MSS10]. However, in the case of (cyclic) letrec and non-determinism the method fails, as a recent counterexample shows [SSMS09]. This raises a question: which call-by-need calculi with letrec permit applicative bisimilarity as a tool for proving contextual equality.

We show in this paper that for the minimal extension of Abramsky's lazy lambda calculus with letrec which implements sharing and explicit recursion, the equivalence of contextual equivalence and applicative bisimulation indeed holds. The technique used is via two translations: W from a call-by-need letrec-calculus into a full call-by-name letrec calculus using infinite trees as justification for the correctness (i.e. full abstraction), and N translating the letrec expressions away using a family of fixpoint combinators. Full abstraction of the translation, an analysis of applicative contexts, and a variant of behavioral similarity then show that the applicative similarity can be transferred between the calculi and that the embedding of the lazy lambda calculus into the call-by-need calculus is an isomorphism of the respective term models.

In [Jef94] there is an investigation into the semantics of a lambda-calculus that permits cyclic graphs, and where a fully abstract denotational semantics is described. However, the calculus is different from our calculi in its expressiveness since it permits strictness annotations and a parallel convergence test, where the latter is required for the full abstraction property of the denotational model. Expressiveness of programming languages was investigated e.g. in [Fel91] and the usage of syntactic methods was formulated as a research program there, with non-recursive let as the paradigmatic example. Our isomorphism-theorem 6.9 shows that this approach is extensible to a cyclic let.

Related work on calculi with recursive bindings includes the following foundational papers. An early paper that proposes cyclic let-bindings (as graphs) is [AK94], where reduction and confluence properties are discussed. [AFM$^+$95,AF97,MOW98] present call-by-need lambda calculi with non-recursive let and a let-less formulation of call-by-need reduction. For a calculus with non-recursive let it is shown in [MOW98] that call-by-name and call-by-need evaluation induce the same observational equivalences. Call-by-need lambda calculi with a recursive let that closely correspond to our calculus L_{need} are also presented in [AFM$^+$95,AF97,AB02]. In [AB02] it is shown that there exist infinite normal forms and that the calculus satisfies a form of confluence. In this paper we show that the letrec calculus of [AF97] is equivalent to L_{need} w.r.t. convergence and contextual equivalence (see Theorem 7.1) and that bisimulation

for the letrec calculus of [AF97] is equivalent to contextual equivalence. This supports our experience and view that contextual equivalence is a more central notion than a specific standard reduction.

Outline: In Sect. 3 we introduce the two letrec-calculi and recall results for Abramsky's lazy lambda calculus. In Sect. 4 and 5 the translations W and N are introduced and the full-abstraction results are obtained. In Sect. 6 we show that bisimulation and contextual equivalence are the same in the call-by-need calculus with letrec. In Sect. 7 we show that our result is transferable to the letrec-calculus of [AF97]. Finally, we conclude in Sect. 8.

2 Common Notions and Notations for Calculi

Before we explain the specific calculi, some common notions are introduced. A calculus definition consists of its syntax together with its operational semantics which defines the evaluation of programs and the implied equivalence of expressions.

Definition 2.1. *An untyped deterministic* calculus $\mathcal{D}$ *is a four-tuple* $(\mathcal{E}, \mathcal{C}, \rightarrow, \mathcal{W})$*, where* $\mathcal{E}$ *are expressions,* $\mathcal{C} : \mathcal{E} \rightarrow \mathcal{E}$ *is a set of functions (which usually represents contexts),* $\rightarrow$ *is a small-step reduction relation (usually the normal-order reduction), which is a partial function on expressions, and* $\mathcal{W} \subset \mathcal{E}$ *is a set of* values *of the calculus.*

For $C \in \mathcal{C}$ *and an expression* s*, the functional application is denoted as* $C[s]$*. For contexts, this is the replacement of the hole of* C *by* s*. We also assume that the identity function* Id *is contained in* $\mathcal{C}$ *with* $Id[s] = s$ *for all expressions* s*.*

The transitive closure *of* $\rightarrow$ *is denoted as* $\xrightarrow{+}$ *and the* transitive and reflexive closure *of* $\rightarrow$ *is denoted as* $\xrightarrow{*}$*. Given an expression* t*, a sequence* $t \rightarrow t_1 \rightarrow \ldots \rightarrow t_n$ *is called a* reduction sequence*; it is called an* evaluation *if* t_n *is a value, i.e.* $t_n \in W$*. Then we say* s converges *and denote this as* $s{\downarrow}t_n$ *or as* $s{\downarrow}$ *if* t_n *is not important. If there is no* t_n *s.t.* $s{\downarrow}t_n$ *then* s diverges*, denoted as* $s{\Uparrow}$*. When dealing with multiple calculi, we often use the calculus name to mark its expressions and relations, e.g.* $\xrightarrow{\mathcal{D}}$ *denotes a reduction relation in* $\mathcal{D}$*.*

Contextual approximation and equivalence can be defined in a general way:

Definition 2.2. *Let* $D = (\mathcal{E}, \mathcal{C}, \rightarrow, \mathcal{W})$ *be a calculus and* s, t *be* D*-expressions.* Contextual approximation $\leq_D$ *and* contextual equivalence $\sim_D$ *are defined as:*

$$s \leq_D t \text{ iff } \forall C \in \mathcal{C} : \quad C[s]{\downarrow_D} \Rightarrow C[t]{\downarrow_D}$$
$$s \sim_D t \text{ iff } s \leq_D t \wedge t \leq_D s$$

Note that $\leq_D$ *is a precongruence and that* $\sim_D$ *is a congruence.*

We are interested in translations between calculi that are faithful w.r.t. the corresponding contextual preorders. Recall that we developed such translations between calculi with contextual equivalences in [SSNSS08,SSNSS09]: A translation $\tau : (\mathcal{E}_1, \mathcal{C}_1, \rightarrow_1, \mathcal{W}_1) \rightarrow (\mathcal{E}_2, \mathcal{C}_2, \rightarrow_2, \mathcal{W}_2)$ is a mapping $\tau_E : \mathcal{E}_1 \rightarrow \mathcal{E}_2$ and a mapping $\tau_C : \mathcal{C}_1 \rightarrow \mathcal{C}_2$ such that $\tau_C(Id_1) = Id_2$. The following notions are defined:

– τ is *compositional* iff $\tau(C[e]) = \tau(C)[\tau(e)]$ for all C, e.
– τ is *convergence equivalent* iff $e{\downarrow_1} \iff \tau(e){\downarrow_2}$ for all e.
– τ is *adequate* iff for all $e, e' \in \mathcal{E}_1$: $\tau(e) \sim_2 \tau(e') \implies e \sim_1 e'$.
– τ is *fully abstract* iff for all $e, e' \in \mathcal{E}_1$: $e \sim_1 e' \iff \tau(e) \sim_2 \tau(e')$.

From [SSNSS08,SSNSS09] it is known that a compositional and convergence equivalent translation is adequate.

3 Three Calculi

In this section we present the calculi that we use in the paper: the two calculi L_{need} and L_{name} with letrec, which have the same syntax, but differ in their reduction strategies, and Abramsky's "lazy lambda calculus", which is a pure lambda calculus with a call-by-name reduction that has abstractions as successful results.

3.1 The Call-by-Need Calculus L_{need}

We begin with the call-by-need lambda calculus L_{need} which is exactly the call-by-need calculus of [SS07]. The set $\mathcal{E}$ of L_{need}-expressions is as follows where x, x_i are variables:

$$s_i, s, t \in \mathcal{E} ::= x \mid (s\ t) \mid (\lambda x.s) \mid (\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t)$$

We assign the names *application*, *abstraction*, or `letrec`-*expression* to the expressions $(s\ t)$, $(\lambda x.s)$, $(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t)$, respectively. A group of `letrec` bindings is abbreviated as Env.

We assume that variables x_i in `letrec`-bindings are all distinct, that letrec-expressions are identified up to reordering of binding-components, and that, for convenience, there is at least one binding. `letrec`-bindings are recursive, i.e., the scope of x_j in $(\texttt{letrec}\ x_1 = s_1, \ldots, x_{n-1} = s_{n-1}\ \texttt{in}\ s_n)$ are all expressions s_i with $1 \leq i \leq n$. Free and bound variables in expressions and α-renamings are defined as usual. The set of free variables in t is denoted as $FV(t)$. 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 α-rename bound variables in the result if necessary.

A *context* C is an expression from L_{need} extended by a symbol $[\cdot]$, the *hole*, such that $[\cdot]$ occurs exactly once (as subexpression) in C. A formal definition is:

Definition 3.1. Contexts $\mathcal{C}$ *are defined by the following grammar:*

$$\begin{aligned} C \in \mathcal{C} ::= {} & [\cdot] \mid (C\ s) \mid (s\ C) \mid (\lambda\ x.C) \mid\ (\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ C) \\ & \mid\ (\texttt{letrec}\ Env, x = C\ \texttt{in}\ s) \end{aligned}$$

(lbeta)	$C[((\lambda x.s)^S\ r)] \to C[(\texttt{letrec}\ x = r\ \texttt{in}\ s)]$
(cp-in)	$(\texttt{letrec}\ x = s^S, Env\ \ \texttt{in}\ C[x^V]) \to (\texttt{letrec}\ x = s, Env\ \ \texttt{in}\ C[s])$ where s is an abstraction or a variable
(cp-e)	$(\texttt{letrec}\ x = s^S, Env, y = C[x^V]\ \texttt{in}\ r) \to (\texttt{letrec}\ x = s, Env, y = C[s]\ \texttt{in}\ r)$ where s is an abstraction or a variable
(llet-in)	$(\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ r)^S) \to (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ r)$
(llet-e)	$(\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s_x)^S\ \texttt{in}\ r) \to (\texttt{letrec}\ Env_1, Env_2, x = s_x\ \texttt{in}\ r)$
(lapp)	$C[((\texttt{letrec}\ Env\ \texttt{in}\ t)^S\ s)] \to C[(\texttt{letrec}\ Env\ \texttt{in}\ (t\ s))]$

Fig. 1. Reduction rules of L_{need}

Given a term t and a context C, we write $C[t]$ for the L_{need}-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 3.2. *The* reduction rules *for the calculus and language L_{need} are defined in Fig. 1, where the labels S, V are used for the exact definition of the normal-order reduction below. Several reduction rules are denoted by their name prefix, e.g. the union of (llet-in) and (llet-e) is called (llet). The union of (llet) and (lapp) is called (lll).*

For the definition of the normal order reduction strategy of the calculus L_{need} we use the labeling algorithm in Figure 2, which detects the position to which a reduction rule is applied according to the normal order. It uses the following labels: S (subterm), T (top term), V (visited). We use $\vee$ when a rule allows two options for a label, e.g. $s^{S \vee T}$ stands for s labeled with S or T. A labeling rule $l \to r$ is applicable to a (labeled) expression s if s matches l with the labels given by l where s may have more labels than l if not otherwise stated. The labeling algorithm has as input an expression s and then exhaustively applies the rules in Fig. 2 to s^T, where no other subexpression in s is labeled. The label T is used to prevent the labeling algorithm from visiting `letrec`-environments that are not at the top of the expression. The labeling algorithm either terminates with *fail* or with success, where in general the direct superterm of the S-marked subexpression indicates a potential normal-order redex. The use of such a labeling algorithm corresponds to the search of a redex in term graphs where it is usually called unwinding.

Example 3.3. For the expression $\texttt{letrec}\ x = x\ \texttt{in}\ x$ the labeling does not fail:

$$(\texttt{letrec}\ x = x\ \texttt{in}\ x)^T \to (\texttt{letrec}\ x = x\ \texttt{in}\ x^S)^V \to (\texttt{letrec}\ x = x^S\ \texttt{in}\ x^V)^V$$

But for the expressions $\texttt{letrec}\ x = (y\ x), y = (x\ y)\ \texttt{in}\ x$ and $\texttt{letrec}\ x = (x\ \lambda u.u)\ \texttt{in}\ x$ the labeling fails.

$$
\begin{array}{ll}
(\texttt{letrec}\ Env\ \texttt{in}\ t)^T & \to (\texttt{letrec}\ Env\ \texttt{in}\ t^S)^V \\
C[(s\ t)^{S \vee T}] & \to C[(s^S\ t)^V] \\
(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C[x^S]) & \to (\texttt{letrec}\ x = s^S, Env\ \texttt{in}\ C[x^V]) \\
(\texttt{letrec}\ x = s, y = C[x^S], Env\ \texttt{in}\ t) & \to (\texttt{letrec}\ x = s^S, y = C[x^V], Env\ \texttt{in}\ t) \\
 & \quad \text{if } s \text{ was not labeled and if } C[x] \neq x \\
(\texttt{letrec}\ x = s^V, y = C[x^S], Env\ \texttt{in}\ t) & \to \textit{fail} \text{ if } C[x] \neq x \\
(\texttt{letrec}\ x = C[x^S]^V, Env\ \texttt{in}\ t) & \to \textit{fail} \text{ if } C[x] \neq x
\end{array}
$$

Fig. 2. Labeling algorithm for L_{need}

Definition 3.4 (Normal Order Reduction of L_{need}). *Let t be an expression. Then a single normal order reduction step $\xrightarrow{need}$ is defined as follows: first the labeling algorithm is applied to t. If the labeling algorithm terminates successfully, then one of the rules in Figure 1 is applied, if possible, where the labels S, V must match the labels in the expression t (again t may have more labels). The* normal order redex *is defined as the left-hand side of the applied reduction rule. The notation for a normal-order reduction that applies the rule a is $\xrightarrow{need,a}$, e.g. $\xrightarrow{need,lapp}$ applies the rule $(lapp)$.*

Definition 3.5. *A* reduction context R_{need} *is any context, such that its hole is labeled with S or T by the labeling algorithm.*

Note that the normal order redex as well as the normal order reduction is unique. A *weak head normal form in L_{need} (L_{need}-WHNF)* is either an abstraction $\lambda x.s$, or an expression $(\texttt{letrec}\ Env\ \texttt{in}\ \lambda x.s)$. The notions of convergence, divergence and contextual approximation are as defined in Sect. 2. Note that black holes, i.e. expressions with cyclic dependencies in a normal order reduction context, diverge, e.g. $\texttt{letrec}\ x = x\ \texttt{in}\ x$. Other expressions which diverge are open expressions where a free variable appears (perhaps after several reductions) in reduction position. A specific representative of diverging expressions is $\Omega := (\lambda z.(z\ z))\ (\lambda x.(x\ x))$, i.e. $\Omega \Uparrow_{need}$.

Example 3.6. We consider the expression $t_1 := \texttt{letrec}\ x = (y\ \lambda u.u), y = \lambda z.z\ \texttt{in}\ x$. The labeling algorithm applied to t_1 yields $(\texttt{letrec}\ x = (y^V\ \lambda u.u)^V, y = (\lambda z.z)^S\ \texttt{in}\ x^V)^V$. The only reduction rule that matches this labeling is the reduction rule (cp-e), i.e. $t_1 \xrightarrow{need} (\texttt{letrec}\ x = ((\lambda z'.z')\ \lambda u.u), y = (\lambda z.z)\ \texttt{in}\ x) = t_2$. The labeling of t_2 is $(\texttt{letrec}\ x = ((\lambda z'.z')^S\ \lambda u.u)^V, y = (\lambda z.z)\ \texttt{in}\ x^V)^V$, which makes the reduction (lbeta) applicable, i.e. $t_2 \xrightarrow{need} (\texttt{letrec}\ x = (\texttt{letrec}\ z' = \lambda u.u\ \texttt{in}\ z'), y = (\lambda z.z)\ \texttt{in}\ x) = t_3$. The labeling of t_3 is $(\texttt{letrec}\ x = (\texttt{letrec}\ z' = \lambda u.u\ \texttt{in}\ z')^S, y = (\lambda z.z)\ \texttt{in}\ x^V)^V$. Thus an (llet-e)-reduction is applicable to t_2, i.e. $t_3 \xrightarrow{L_{need}} (\texttt{letrec}\ x = z', z' = \lambda u.u, y = (\lambda z.z)\ \texttt{in}\ x) = t_4$. Application of the labeling algorithm to t_4 yields: $(\texttt{letrec}\ x = z'^S, z' = \lambda u.u, y = (\lambda z.z)\ \texttt{in}\ x^V)^V$. Thus the normal order reduction is a (cp-in)-reduction, i.e. $t_4 \xrightarrow{L_{need}} (\texttt{letrec}\ x = z', z' = \lambda u.u, y = (\lambda z.z)\ \texttt{in}\ z') = t_5$ The la-

beling of t_5 is $(\texttt{letrec}\ x = z', z' = \lambda u.u^S, y = (\lambda z.z)\ \texttt{in}\ z'^V)^V$. Again a (cp-e) reduction is applicable, i.e. $t_5 \to (\texttt{letrec}\ x = z', z' = \lambda u.u, y = (\lambda z.z)\ \texttt{in}\ \lambda u'.u') = t_6$ The labeling algorithm applied to t_6 yields $(\texttt{letrec}\ x = z', z' = \lambda u.u, y = (\lambda z.z)\ \texttt{in}\ \lambda u'.u'^S)^V$, but no reduction is applicable to t_6, since t_6 is a WHNF.

An Alternative Definition of Normal Order Reduction Reduction contexts of L_{need} can be syntactically defined by the following grammar

$$\begin{aligned} R_{need} \in \mathcal{R}_{need} := {} & A \mid \texttt{letrec}\ \mathit{Env}\ \texttt{in}\ A \\ & \mid \texttt{letrec}\ x_1 = A_1, x_2 = A_2[x_1], \ldots, x_n = A_n[x_{n-1}], \mathit{Env}\ \texttt{in}\ A[x_n] \end{aligned}$$

where x_i are variables, $A_2, \ldots, A_n$ are not the empty context and A, A_i are $\mathcal{A}$-contexts defined as $A \in \mathcal{A} ::= [\cdot] \mid (A\ s)$ where s is an expression.

Normal order reduction can be alternatively defined (without labels) as follows:

(lbeta) $R_{need}[(\lambda x.s)\ r] \to R_{need}[\texttt{letrec}\ x = r\ \texttt{in}\ s]$

(cp-in) $\texttt{letrec}\ y = s, \mathit{Env}\ \texttt{in}\ A[y] \to \texttt{letrec}\ y = s, \mathit{Env}\ \texttt{in}\ A[s]$
where s is an abstraction or a variable

(cp-e) $\texttt{letrec}\ y_1 = s, y_2 = A_2[y_1], \ldots, y_n = A_n[y_{n-1}], \mathit{Env}\ \texttt{in}\ A[y_n]$
$\to \texttt{letrec}\ y_1 = s, y_2 = A_2[s], \ldots, y_n = A_n[y_{n-1}], \mathit{Env}\ \texttt{in}\ A[y_n]$
where s is an abstraction or a variable, and $A_2, \ldots, A_n$ are non-empty $\mathcal{A}$-contexts

(llet-in) $(\texttt{letrec}\ \mathit{Env}_1\ \texttt{in}\ (\texttt{letrec}\ \mathit{Env}_2\ \texttt{in}\ r)) \to (\texttt{letrec}\ \mathit{Env}_1, \mathit{Env}_2\ \texttt{in}\ r)$

(llet-e) $\texttt{letrec}\ y_1 = (\texttt{letrec}\ \mathit{Env}_1\ \texttt{in}\ r), y_2 = A_2[y_1], \ldots, y_n = A_n[y_{n-1}], \mathit{Env}_2\ \texttt{in}\ A[y_n]$
$\to \texttt{letrec}\ y_1 = r, \mathit{Env}_1, y_2 = A_2[y_1], \ldots, y_n = A_n[y_{n-1}], \mathit{Env}\ \texttt{in}\ A[y_n]$
where $A_2, \ldots, A_n$ are non-empty $\mathcal{A}$-contexts

(lapp) $R_{need}[((\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ r)\ t)] \to R_{need}[(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (r\ t))]$

3.2 The Call-by-Name Calculus L_{name}

Now we define a call-by-name calculus on the L_{need}-syntax. The syntax of the calculus L_{name} is the same as that of L_{need}, but the reduction rules are different. This calculus L_{name} has a different call-by-name-reduction than the one in [SS07], since that calculus treats only beta-redexes as call-by-name, but uses a sharing variant for (cp).

The reduction contexts R_{name} are contexts of the form $L[A]$ where the context classes $\mathcal{A}$ and $\mathcal{L}$ are defined by $L \in \mathcal{L} ::= [\cdot] \mid \texttt{letrec}\ \mathit{Env}\ \texttt{in}\ L$; $A \in \mathcal{A} ::= [\cdot] \mid (A\ s)$ where s is any expression. Normal order reduction $\xrightarrow{name}$ is defined by the following three rules:

$$\begin{array}{lll} \text{(lapp)} & R_{name}[(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ t)\ s] & \to R_{name}[\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (t\ s)] \\ \text{(beta)} & R_{name}[((\lambda x.s)\ t)] & \to R_{name}[s[t/x]] \\ \text{(cp)} & L[\texttt{letrec}\ \mathit{Env},\ x = s\ \texttt{in}\ R_{name}[x]] & \to L[\texttt{letrec}\ \mathit{Env},\ x = s\ \texttt{in}\ R_{name}[s]] \end{array}$$

Note that $\xrightarrow{name}$ is unique. An L_{name}-WHNF is defined as an expression of the form $L[\lambda x.s]$. We write $s{\downarrow_{name}}$ iff there is a normal-order reduction to a L_{name}-WHNF, i.e. iff $s \xrightarrow{name,*} L[\lambda x.s']$.

3.3 The Lazy Lambda Calculus

In this subsection we give a short description of the lazy lambda calculus [Abr90], denoted with L_{lazy}, which is a call-by-name lambda calculus. The set $\mathcal{E}$ of L_{lazy}-expressions is that of the usual (untyped) lambda calculus: $s, s_i, t \in \mathcal{E} ::= x \mid (s_1\ s_2) \mid (\lambda x.s)$ where e, e_i are expressions, and x means a variable. The set $\mathcal{W}$ of *values* are the L_{lazy}-abstractions. The reduction contexts $\mathcal{R}_{lazy}$ are defined by $R_{lazy} \in \mathcal{R}_{lazy} := [\cdot] \mid (R_{lazy}\ s)$ where s is any L_{lazy}-expression. A $\xrightarrow{lazy}$-reduction is defined by the rule: (beta) $\quad R_{lazy}[((\lambda x.s)\ t)] \to R_{lazy}[s[t/x]]$. The $\xrightarrow{lazy}$-reduction is unique.

We repeat the definitions and the required properties of L_{lazy}, where proofs can be found in [How89,How96,Abr90,AO93]. For basic definitions and confluence see e.g. [Bar84]. Since this calculus is well-studied and some properties are folklore, there are different and alternative proofs of the properties below.We give a sketch of how these proofs can be constructed in the Appendix B. We require these properties in other sections and as properties of the target of translations, which allows us to lift the properties to the calculi L_{name} and L_{need}.

Definition 3.7 (Simulation in L_{lazy}). *Let η be a binary relation on closed L_{lazy}-expressions. Then $s\ [\eta]_{lazy}\ t$ holds iff $s{\downarrow}\lambda x.s'$ implies $\big(\ t{\downarrow}\lambda x.t'$ and for all closed L_{lazy}-expressions r the relation $s'[r/x]\ \eta\ t'[r/x]$ holds$\big)$. The relation $\leq_{b,lazy}$ is defined as the greatest fixpoint of the operator $[\cdot]_{lazy}$.*

For a relation η on closed expressions, let the open extension η^o be defined as $s\ \eta^o\ t$ iff for all closing substitutions σ: $\sigma(s)\ \eta\ \sigma(t)$. Note that by the theorem below, this can be shown to be equivalent to: for all closing substitutions σ that replace variables by closed abstractions or Ω: $\sigma(s)\ \eta\ \sigma(t)$. As an example $\leq^o_{b,lazy}$ is the open extension of $\leq_{b,lazy}$.

There are several variants of behaviorally and contextually defined relations in L_{lazy}, that are all equivalent to contextual approximation. We omit the proof here, but it can be found in Appendix B.

Theorem 3.8. *In L_{lazy}, all the following relations are equivalent to contextual approximation $\leq_{lazy}$:*

1. $\leq^o_{b,lazy}$.
2. *The relation $\leq_{lazy,1}$ where $s \leq_{lazy,1} t$ iff for all closing contexts C: $C[s]{\downarrow} \implies C[t]{\downarrow}$.*
3. *The relation $\leq_{lazy,2}$, defined as: $s \leq_{lazy,2} t$ iff for all closed contexts C and all closing substitutions: $C[\sigma(s)]{\downarrow} \implies C[\sigma(t)]{\downarrow}$.*
4. *The relation $\leq^o_{b,lazy,1}$ where $\leq_{b,lazy,1}$ is defined using the Kleene-construction: $\leq_{b,lazy,1} = \bigcap_{i\geq 0} \leq'_{b,i}$, where $\leq'_{b,0}$ is the relation $\mathcal{E}\times\mathcal{E}$, and $\leq'_{b,i+1} := [\leq'_{b,i}]_{lazy}$ for all i.*
5. *The relation $\leq^o_{b,lazy,2}$ where $\leq_{b,lazy,2}$ is defined as: $s \leq_{b,lazy,2} t$ iff for all $n \geq 0$ and all closed expressions $r_i, i = 1, \ldots, n$: $s\ r_1 \ldots r_n{\downarrow} \implies t\ r_1 \ldots r_n{\downarrow}$.*
6. *The relation $\leq^o_{b,lazy,3}$, where $\leq_{b,lazy,3}$ is defined as: $s \leq_{b,lazy,3} t$ iff for all $n \geq 0$ and all $r_i, i = 1, \ldots, n$, where r_i may be a closed abstraction or Ω: $s\ r_1 \ldots r_n{\downarrow} \implies t\ r_1 \ldots r_n{\downarrow}$.*

7. *The relation* $\leq^o_{b,lazy,4}$, *where* $\leq_{b,lazy,4}$ *is the greatest fixpoint of the operator* $[\cdot]_{lazy,a\Omega}$ *on closed expressions. By definition* $s\ [\eta]_{lazy,a\Omega}\ t$ *holds iff* $s{\downarrow}\lambda x.s'$ *implies* $t{\downarrow}\lambda x.t'$ *and for all closed* L_{lazy}*-abstractions* r *and* $r = \Omega$*, the relation* $s'[r/x]\ \eta\ t'[r/x]$ *holds.*

Beta-reduction is a correct program transformation in L_{lazy}:

Theorem 3.9. *Let* s, t *be* L_{lazy}*-expressions. If* $s \xrightarrow{beta} t$*, then* $s\ \sim_{lazy}\ t$*. For all* L_{lazy}*-expressions* s, t*:* $\Omega \leq_{lazy}\ s$*. If* s, t *are closed and* $s{\Uparrow}$ *and* $t{\Uparrow}$*, then* $s\ \sim_{lazy}\ t$*.*

Also the following can easily be derived from Theorem 3.8 and Theorem 3.9.

Proposition 3.10. *For open* L_{lazy}*-expressions* s, t*, where all free variables of* s, t *are in* $\{x_1, \ldots, x_n\}$*:* $s \leq_{lazy} t \iff \lambda x_1, \ldots x_n.s \leq_{lazy} \lambda x_1, \ldots x_n.t$

Proposition 3.11. *Given any two closed* L_{lazy}*-expressions* s, t*: for all closed* L_{lazy}*-abstractions* r *and also for* $r = \Omega$ $s\ r \leq_{lazy} t\ r \iff s \leq_{lazy} t$*.*

Proof. The if-direction follows from the congruence property. The only-if direction follows from Theorem 3.8.

4 The Translation $W : L_{need} \to L_{name}$

The translation $W : L_{need} \to L_{name}$ is defined as the identity on expressions and contexts, but the convergence predicates are changed. We will prove that contextual equivalence based on L_{need}-evaluation and contextual equivalence based on L_{name}-evaluation are equivalent. We will use infinite trees to connect both evaluation strategies. Note that [SS07] already shows that infinite tree convergence is equivalent to call-by-need convergence. Thus, we mainly treat call-by-name evaluation in this section.

We recall the definition of an infinite tree from [SS07], and describe the set of trees as a calculus in the sense of Section 2 called L_{tree}: The set of infinite trees $\mathcal{T}$ is co-inductively defined using the grammar $T \in \mathcal{T} ::= x \mid (T_1\ T_2) \mid \lambda x.T \mid \bot$ where x is a variable, T, T_1, T_2 are infinite trees, $\bot$ is a (special) constant. Contexts are trees with exactly one occurrence of a hole (as a subexpression).

Definition 4.1. *Tree reduction contexts* $\mathcal{R}$ *for (infinite) trees are inductively defined by* $\mathcal{R} ::= [\cdot] \mid (\mathcal{R}\ T)$*, where* T *stands for an infinite tree. The only reduction on trees is:*

$$(betaTr) \qquad ((\lambda x.s)\ r) \to s[r/x]$$

If the reduction rule is applied in an $\mathcal{R}$*-context, it is a* normal order reduction on trees $\xrightarrow{tree}$*. Values are trees of the form* $\lambda x.T$*, i.e. abstractions.*

Now we define a translation IT from L_{name}-expressions into L_{tree}-expressions.

We use Dewey notation, i.e. strings over $\{1, 2\}$, as positions of infinite trees, where numbers are separated by a period. Here 1 refers to the left and 2 to the right subtree of an application, and 1 to the body of an abstraction. The empty string is denoted as ε. For an infinite tree T its *label at position* p (written as $T\lfloor_p$) is defined as usual, i.e. $(T_1\ T_2)\lfloor_{1.p} = T_1\lfloor_p$, $(T_1\ T_2)\lfloor_{2.p} = T_2\lfloor_p$, $(\lambda x.T)\lfloor_\varepsilon = \lambda x$, $(T_1\ T_2)\lfloor_\varepsilon = app$, $x\lfloor_\varepsilon = x$, and $\bot\lfloor_\varepsilon = \bot$. The subtree of T at position p is $T|_p$.

Definition 4.2. *Given an expression t, the infinite tree $IT(t)$ of t is defined by the labels at valid positions, where the positions and the labels of $IT(t)$ for every position are computed by the following algorithm, using the notation $C[t'\lfloor_p]$ if the algorithm searches the label at position p and is currently at the subexpression t'. Given the expression t and a position p, if and only if the following rules ($\mapsto$) (where C, C_i are L_{name}-contexts, s, t are L_{name}-expressions) exhaustively applied to $t\lfloor_p$ end with a label $l \in \{\lambda x, app, x, \bot\}$, then p is a position of $IT(t)$ and $IT(t)\lfloor_p = l$.*

The final steps in the label computation are as follows:

$$
\begin{array}{lll}
C[(\lambda x.s)\lfloor_\varepsilon] & \mapsto & \lambda x \\
C[(s\ t)\lfloor_\varepsilon] & \mapsto & app \\
C[x\lfloor_\varepsilon] & \mapsto & x \qquad \textit{if } x \textit{ is a free or a lambda-bound variable} \\
C[\texttt{letrec}\ x = C[x\lfloor_\varepsilon], Env\ \texttt{in}\ s] & & \mapsto \bot \\
C[\texttt{letrec}\ x_1 = C_1[y_1], \ldots, x_n = C_n[x_1\lfloor_\varepsilon], Env\ \texttt{in}\ s] & & \mapsto \bot
\end{array}
$$

For the general cases, we proceed as follows:

$$
\begin{array}{lll}
1.\ C[(\lambda x.s)\lfloor_{1.p}] & \mapsto & C[\lambda x.(s\lfloor_p)] \\
2.\ C[(s\ t)\lfloor_{1.p}] & \mapsto & C[(s\lfloor_p\ t)] \\
3.\ C[(s\ t)\lfloor_{2.p}] & \mapsto & C[(s\ t\lfloor_p)] \\
4.\ C[(\texttt{letrec}\ Env\ \texttt{in}\ r)\lfloor_p] & \mapsto & C[(\texttt{letrec}\ Env\ \texttt{in}\ r\lfloor_p)] \\
5.\ C_1[(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C_2[x\lfloor_p])] & \mapsto & C_1[(\texttt{letrec}\ x = s\lfloor_p, Env\ \texttt{in}\ C_2[x])] \\
6.\ C_1[\texttt{letrec}\ x = s, y = C_2[x\lfloor_p], Env\ \texttt{in}\ r] & \mapsto & C_1[\texttt{letrec}\ x = s\lfloor_p, y = C_2[x], Env\ \texttt{in}\ r]
\end{array}
$$

In all cases not mentioned above, the result is undefined, and hence the position p is not a position of the tree.

Lemma 4.3. *Let $s, t \in L_{name}$. Then $s \xrightarrow{name,cp} t$ or $s \xrightarrow{name,lapp} t$ implies $IT(s) = IT(t)$.*

Proof. For (cp) let $s = C_1[\texttt{letrec}\ x = s, Env\ \texttt{in}\ C_2[x]]$ and $t = C_1[\texttt{letrec}\ x = s, Env\ \texttt{in}\ C_2[s]]$. Then for $IT(s)$ and $IT(t)$ the only change may happen at the position that corresponds to x in $C_2[x]$, but as the computation of the labels shows, the labels remain unchanged.

For (lapp) let $s = C[(\texttt{letrec}\ Env\ \texttt{in}\ s')\ t']$ and $t = C[\texttt{letrec}\ Env\ \texttt{in}\ (s'\ t')]$. Then it is again easy to observe that every label of every position is identical for $IT(s)$ and $IT(t)$.

Lemma 4.4. *Let* $s_1 := R_{name}[(\lambda x.s)\ t] \xrightarrow{name,beta} R_{name}[s[t/x]] =: s_2$. *Then* $IT(s_1) \xrightarrow{tree} IT(s_2)$.

Proof. The redex $((\lambda x.s)\ t)$ is mapped by IT to a unique tree position within a tree reduction context in $IT(s_1)$. The computation IT transforms $((\lambda x.s)\ t)$ into a subtree $\sigma((\lambda x.s)\ t)$, where σ is a substitution replacing variables by infinite trees. The tree reduction replaces $\sigma((\lambda x.s)\ t)$ by $\sigma(s)[\sigma(t)/x]$, hence the lemma holds.

Proposition 4.5. *Let* s *be an expression with* $s{\downarrow_{name}}$. *Then* $IT(s){\downarrow_{tree}}$.

Proof. This follows by induction on the length of a normal order reduction of s. The base case holds, since $IT(L[(\lambda x.s)])$ is always a value tree. For the induction step we consider the first reduction of s, say $s \to s'$. The induction hypothesis shows $IT(s'){\downarrow_{tree}}$. If the reduction $s \to s'$ is a ($name$,lapp) or ($name$,cp) reduction, then Lemma 4.3 implies $IT(s){\downarrow_{tree}}$. If $s \xrightarrow{name,beta} s'$, then Lemma 4.4 shows $IT(s) \xrightarrow{tree} IT(s')$ and thus $IT(s){\downarrow_{tree}}$.

Now we show the other direction:

Lemma 4.6. *Let* s *be an expression such that* $IT(s) = \mathcal{R}[T]$, *where* $\mathcal{R}$ *is a tree reduction context and* $T \neq \bot$. *Then there is an expression* s' *such that* $s \xrightarrow{name,(lapp)\vee(cp),*} s'$, $IT(s') = IT(s)$, $s' = R[s'']$, $IT(L[s'']) = T$, *where* $R = L[A[\cdot]]$ *is a reduction context for some* $\mathcal{L}$*-context* L *and some* $\mathcal{A}$*-context* A, s'' *is a free variable, an abstraction or an application iff* T *is a free variable, an abstraction or an application, respectively, and the position* p *of the hole in* $\mathcal{R}$ *is also the position of the hole in* $A[\cdot]$.

Proof. The tree T may be an abstraction, an application, or a free variable in $R[T]$. Let p be the position of the hole of $\mathcal{R}$. We will show by induction on the label-computation for p in s that there is a reduction $s \xrightarrow{name,(lapp)\vee(cp),*} s'$, where s' as claimed in the lemma.
We consider the label-computation for p to explain the induction measure, where we use the rule numbers of Definition 4.2. Let q be such that the label computation for p is of the form 4^*q and q does not start with 4. The measure for induction is a tuple (a, b), where a is the length of q, and $b \geq 0$ is the maximal number with $q = 2^b q'$. The base case is (a, a): Then the label computation is of the form 2^* and indicates that s is of the form $L[A[s'']]$ and satisfies the claim of the lemma. For the induction step we have to check several cases:

1. The label computation is of the form $4^*2^+4\ldots$. Then a normal-order (lapp) can be applied to s resulting in s_1. The label-computation for p w.r.t. s_1 is of the same length, and only applications of 2 and 4 are interchanged, hence the second component of the measure is strictly decreased.
2. The label computation is of the form $4^*2^*5\ldots$. Then a normal-order (cp) can be applied to s resulting in s_1. The length q is strictly decreased by 1, and perhaps one 6.-step is changed into a 5.-step. Hence the measure is strictly reduced. □

Lemma 4.7. *Let s be an expression with $IT(s) \xrightarrow{tree} T$. Then there is some s' with $s \xrightarrow{name,*} s'$ and $IT(s') = T$.*

Proof. If $IT(s) \xrightarrow{tree} T$, then $IT(s) = \mathcal{R}[(\lambda x.t_1)\ t_2]$ where $\mathcal{R}$ is a reduction context and $T = \mathcal{R}[t_1[t_2/x]]$. Let p be the position of the hole of $\mathcal{R}$ in $IT(s)$. We first apply Lemma 4.6 to s and the tree context $\mathcal{R}[([\cdot]\ t_2)]$ and thus obtain a reduction $s \xrightarrow{name,*} s'$, such that $IT(s) = IT(s')$ and $s' = R[r]$ where $R = L[A[\cdot]]$ is a reduction context and $IT(L[r]) = (\lambda x.t_1)$, and r is an abstraction. It is obvious that $IT(s')|_{p.2} = t_2$ and that $R = L[A'[[\cdot]\ r_2]]$. Thus $s' = L[A'[((\lambda x.r_1)\ r_2)]] \xrightarrow{name,beta} L[A'[r_1[r_2/x]] = s''$. Now one can verify that $IT(s'') = T$ must hold.

Proposition 4.8. *Let s be an expression with $IT(s)\downarrow_{tree}$. Then $s\downarrow_{name}$.*

Proof. We use induction on the length k of a tree reduction $IT(s) \xrightarrow{tree,k} T$, where T is a value tree. For the base case it is easy to verify that if $IT(s)$ is a value tree, then $s \xrightarrow{name,cp,*} L[\lambda x.s']$ for some $\mathcal{L}$-context and some s'. I.e. $s \downarrow_{name}$. The induction step follows by Lemma 4.7.

Corollary 4.9. *For all L_{name}-expressions s: $s\downarrow_{name}$ if, and only if $IT(s)\downarrow_{tree}$.*

Theorem 4.10. $\leq_{name} = \leq_{need}$

Proof. We have shown that L_{name}-convergence is equivalent to infinite tree convergence. In [SS07] it was shown that L_{need}-convergence is equivalent to infinite tree convergence. Hence, L_{name}-convergence and L_{need}-convergence are equivalent, which also implies that both contextual preorders and also the contextual equivalences are identical.

Corollary 4.11. *W is convergence equivalent and fully abstract.*

5 Translation $N : L_{name} \longrightarrow L_{lazy}$

We use multi-fixpoint combinators as defined in [Gol05] to translate letrec-expressions into equivalent ones without a `letrec`. The translated expressions belong to L_{lazy}.

Definition 5.1. *Given $n > 1$, a family of n fixpoint combinators Y_i^n for $i = 1, \ldots, n$ can be defined as follows:*

$$\begin{aligned} Y_i^n := \lambda f_1, \ldots, f_n.(\ & (\lambda x_1, \ldots, x_n.f_i\ (x_1\ x_1\ \ldots x_n)\ \ldots\ (x_n\ x_1\ \ldots x_n)) \\ & (\lambda x_1, \ldots, x_n.f_1\ (x_1\ x_1\ \ldots x_n)\ \ldots\ (x_n\ x_1\ \ldots x_n)) \\ & \ldots \\ & (\lambda x_1, \ldots, x_n.f_n\ (x_1\ x_1\ \ldots x_n)\ \ldots\ (x_n\ x_1\ \ldots x_n))) \end{aligned}$$

The idea of the translation is to replace ($\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ r$) by $r[S_1/x_1, \ldots, S_n/x_n]$ where $S_i := Y_i^n\ F_1 \ldots F_n$ and $F_i := \lambda x_1, \ldots, x_n.s_i$.

In this way the fixpoint combinators implement the generalized fixpoint property: $Y_i^n\ F_1 \ldots F_n \sim F_i\ (Y_1^n\ F_1 \ldots F_n) \ldots (Y_n^n\ F_1 \ldots F_n)$. However, our translation uses modified expressions, as shown below.

Consider the expression $Y_i^n F_1 \ldots F_n$. Expanding the notations, we get $((\lambda f_1, \ldots, f_n.(X_i\ \ X_1\ \ \ldots\ \ X_n))\ \ F_1\ \ \ldots\ \ F_n)$ where $X_i = \lambda x_1 \ldots x_n.(f_i\ (x_1\ x_1\ \ \ldots\ \ x_n)\ \ \ldots\ \ (x_n\ x_1\ \ \ldots\ \ x_n))$. Reducing further:

$$(\lambda f_1, \ldots, f_n.(X_i\ X_1\ \ \ldots\ \ X_n))\ F_1\ \ \ldots\ \ F_n \xrightarrow{\beta,*} (X_i'\ X_1'\ \ \ldots\ \ X_n'),$$
$$\text{where } X_i' = \lambda x_1 \ldots x_n.(F_i\ (x_1\ x_1\ \ \ldots\ \ x_n) \ldots (x_n\ x_1\ \ \ldots x_n))$$

We take the latter expression as the definition of the multi-fixpoint translation, where we avoid substitutions and instead generate β-redexes.

Definition 5.2. *The translation $N :: L_{name} \to L_{lazy}$ is recursively defined as:*

- $N(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ r) = ((\lambda x_1. \ldots x_n.(N(r)))\ U_1\ \ \ldots\ \ U_n)$
  where $U_i = (\lambda x_1, \ldots, x_n.x_i\ x_1 \ldots x_n)\ X_1'\ \ \ldots\ \ X_n'$,
  $X_i' = \lambda x_1 \ldots x_n.F_i(x_1 x_1 \ldots x_n) \ldots (x_n x_1 \ldots x_n)$,
  $F_i = \lambda x_1, \ldots, x_n.N(s_i)$.
- $N(s_1\ s_2) = (N(s_1)\ N(s_2))$
- $N(\lambda x.s) = \lambda x.N(s)$
- $N(x) = x$.

We extend N to contexts by treating the hole as a constant, i.e. $N([\cdot]) = [\cdot]$.

Convergence equivalence of the translation N follows by inspecting the relation between L_{name}- and the translated L_{lazy}-reductions. The full proof can be found in Appendix C, Proposition C.6.

Proposition 5.3. *N is convergence equivalent, i.e. $\forall t \in L_{name}$: $t{\downarrow_{name}} \iff N(t){\downarrow_{lazy}}$.*

Lemma 5.4. *The translation N is compositional, i.e. for all expressions t and all contexts C: $N(C[t]) = N(C)[N(t)]$.*

Proof. This easily follows by structural induction on the definition.

Proposition 5.5. *For all $s, t \in L_{name}$: $N(s) \leq_{lazy} N(t) \implies s \leq_{name} t$, i.e. N is adequate.*

Proof. Since N is convergence equivalent (Proposition 5.3) and compositional by Lemma 5.4, we derive that N is adequate (see [SSNSS08] and Section 2).

Lemma 5.6. *For* `letrec`*-free expressions s, t of L_{name} the following holds: $s, t \in L_{lazy}$ and $s \leq_{name} t \implies s \leq_{lazy} t$.*

Proof. Clearly every `letrec`-free expression of L_{name} is also an L_{lazy} expression. Let s, t be `letrec`-free such that $s \leq_{name} t$. Let C be an L_{lazy}-context such that $C[s]{\downarrow_{lazy}}$, i.e. $C[s] \xrightarrow{lazy,k} \lambda x.s'$. By comparing the reduction strategies in L_{name} and L_{lazy}, we obtain that $C[s] \xrightarrow{name,k} \lambda x.s'$ (by the identical reduction sequence), since $C[s]$ is `letrec`-free. Thus, $C[s]{\downarrow_{name}}$ and also $C[t]{\downarrow_{name}}$, i.e. there is a normal order reduction in L_{name} for $C[t]$ to a WHNF. Since $C[t]$ is `letrec`-free, we can perform the identical reduction in L_{lazy} and obtain $C[t]{\downarrow_{lazy}}$.

The language L_{lazy} is embedded into L_{name} (and also L_{need}) by the identity embedding $\iota(s) = s$. In the following proposition we show that every L_{need}-WHNF (and also every L_{name}-WHNF) is contextually equivalent to an abstraction:

Proposition 5.7. *For all $s \in L_{name}$: $s \sim_{name} \iota(N(s))$. If s is an L_{need}-WHNF and $N(s){\downarrow_{lazy}} v$ where v is an abstraction, then $s \sim_{need} \iota(v)$.*

Proof. We first show that for all expressions $s \in L_{name}$: $s \sim_{name} \iota(N(s))$. Since N is the identity mapping on `letrec`-free expressions of L_{name} and $N(s)$ is `letrec`-free, we have $N(\iota(N(s))) = N(s)$. Hence adequacy of N (Proposition 5.5) implies $s \sim_{name} \iota(N(s))$. Theorem 3.9 shows $N(s) \sim_{lazy} v$ and Proposition 5.5 show that $\iota(v) \sim_{name} \iota(N(s)) \sim_{name} s$. Finally, Theorem 4.10 shows the claim.

Proposition 5.8. *For all $s, t \in L_{name}$: $s \leq_{name} t \implies N(s) \leq_{lazy} N(t)$.*

Proof. For this proof we treat L_{lazy} expressions as L_{name} expressions. Let $s, t \in L_{name}$ and $s \leq_{name} t$. By Proposition 5.7: $N(s) \sim_{name} s \leq_{name} t \sim_{name} N(t)$ and thus $N(s) \leq_{name} N(t)$. Since $N(s)$ and $N(t)$ are `letrec`-free, we can apply Lemma 5.6 and thus have $N(s) \leq_{lazy} N(t)$.

Now we put all parts together, where $(N \circ W)(s)$ means $N(W(s))$:

Theorem 5.9. *N and $N \circ W$ are fully-abstract, i.e. for all L_{need}-expressions s, t: $s \leq_{need} t \iff N(W(s)) \leq_{lazy} N(W(t))$.*

6 On Simulation in L_{need}

First we show that finite simulation (see [SSM08]) is correct for L_{need}:

Proposition 6.1. *Let s, t be closed expressions in L_{need}. The following holds: $\big($For all closed abstractions r and for $r = \Omega$: $s\ r \leq_{need} t\ r\big) \iff s \leq_{need} t$.*

Proof. The $\Leftarrow$ direction is trivial. We show the nontrivial part. Assume that for all closed abstractions r and for $r = \Omega$: $s\ r \leq_{need} t\ r$. Then we transfer the problem to L_{lazy} as follows: $N(s)$ and $N(t)$ are closed expressions in L_{lazy}. Since the translation N is surjective, every closed L_{lazy}-expression is in the image of N. Thus for every closed L_{lazy}-expression r' that is an abstraction or Ω, there is some L_{need}-expression r, such that $N(r) = r'$. We have $N(s)\ r'{\downarrow} \implies N(t)\ r'{\downarrow}$, since $N(s\ r) = (N(s)\ N(r))$, and since N is fully abstract. We can apply Proposition 3.11 and obtain $N(s) \leq_{lazy} N(t)$. Now Theorem 5.9 shows $s \leq_{need} t$.

Now we show that the co-inductive definition of an applicative simulation results in a relation equivalent to contextual preorder. We show the following helpful lemma:

Lemma 6.2. *For all closed expressions s and r and L_{need}-WHNFs w: $(s\ r)\downarrow w \iff \exists v : s\downarrow v \wedge (v\ r)\downarrow w$.*

Proof. In order to prove "$\Rightarrow$" let $(s\ r)\downarrow w$. There are two cases, which can be verified by induction on the length k of a reduction sequence $(s\ r) \xrightarrow{need,k} w$: $(s\ r) \xrightarrow{need,*} ((\lambda x.s')\ r) \xrightarrow{need,*} w$, where $s \xrightarrow{need,*} (\lambda x.s')$, and the claim holds. The other case is $(s\ r) \xrightarrow{need,*} (\texttt{letrec}\ Env\ \texttt{in}\ ((\lambda x.s')\ r)) \xrightarrow{need,*} w$, where $s \xrightarrow{need,*} (\texttt{letrec}\ Env\ \texttt{in}\ (\lambda x.s'))$. In this case $((\texttt{letrec}\ Env\ \texttt{in}\ (\lambda x.s'))\ r) \xrightarrow{need,(lapp)} (\texttt{letrec}\ Env\ \texttt{in}\ ((\lambda x.s')\ r)) \xrightarrow{need,*} w$, and thus the claim is proven. The "$\Leftarrow$"-direction can be proven in a similar way using induction on the length of reduction sequences.

Definition 6.3. *We define in L_{need} a simulation $\leq_{b,need}$ as follows:*
Let s, t be closed expressions and η be a binary relation on closed expressions. Then $s\ [\eta]_{need}\ t$ holds iff $s\downarrow_{need} v$ implies that $t\downarrow_{need} w$, and for all closed letrec-free abstractions r and for $r = \Omega$: $(v\ r)\ \eta\ (w\ r)$.

The relation $\leq_{b,need}$ is defined to be the greatest fixpoint of $[\cdot]_{need}$ within binary relations on closed expressions. Its open extension is denoted with $\leq^o_{b,need}$.

Proposition 6.4. *In L_{need}, for closed s, t the statement $s \leq_{b,need} t$ is equivalent to the following condition for s, t:*
$\forall n \geq 0$, and for all $r_i, i = 1, \ldots, n$ that may be closed letrec-free abstractions or Ω: $(s\ r_1 \ldots r_n)\downarrow_{need} \implies (t\ r_1 \ldots r_n)\downarrow_{need}$.

Proof. Lemma 6.2 makes Theorem A.11 (see Appendix A) applicable for the tests $([\cdot]\ r)$ where r is a closed letrec-free abstraction or Ω.

Now we can prove that the simulation relation $\leq_{b,need}$ is equivalent to the contextual preorder on closed expressions:

Theorem 6.5. *For closed expressions s, t: $s \leq_{b,need} t \iff s \leq_{need} t$.*

Proof. Let $\leq_{need,0}$ the restriction of $\leq_{need}$ to closed expressions. It is easy to verify that $\leq_{need,0} \subseteq [\leq_{need,0}]_{need}$ and thus for closed expressions s, t: $s \leq_{need} t \implies s \leq_{b,need} t$. For the other direction let $s \leq_{b,need} t$. The criterion in Proposition 6.4 then implies that for all $n \geq 0 : s\ r_1\ \ldots\ r_n \downarrow_{need} \implies t\ r_1\ \ldots\ r_n \downarrow_{need}$, where r_i are closed letrec-free abstractions or Ω. Full-abstraction of $N \circ W$ (see Theorem 5.9) implies that $N(W(s\ r_1\ \ldots\ r_n)) \downarrow_{lazy} \implies N(W(t\ r_1\ \ldots\ r_n)) \downarrow_{lazy}$. Since N and W translate applications into applications, this also shows that $N(W(s))\ N(W(r_1))\ \ldots\ N(W(r_n)) \downarrow_{lazy} \implies N(W(t))\ N(W(r_1))\ \ldots\ N(W(r_n)) \downarrow_{lazy}$. Moreover, since every L_{lazy}-abstractions is an $N \circ W$-image of a letrec-free abstraction, we also conclude that $N(W(s)) \leq_{b,lazy,3} N(W(t))$. Now Theorem 3.8 and full abstraction of $N \circ W$ finally show $s \leq_{need} t$.

Using the characterization in Proposition 6.4, it is possible to prove non-trivial equations, as shown in the example below.

Example 6.6. We consider two fixpoint combinators Y_1 and Y_2, where Y_1 is defined non-recursively, while Y_2 uses recursion. The definitions are: $Y_1 := \lambda f.((\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x)))$, $\quad Y_2 := \texttt{letrec}\ \mathit{fix} = \lambda f.f\ (\mathit{fix}\ f)\ \texttt{in}\ \mathit{fix}$.

Using Proposition 6.4 we can easily derive that $Y_1\ K \sim_{need} Y_2\ K$ where $K := \lambda a.(\lambda b.a)$. This follows since $(Y_1\ K\ r_1 \dots\ r_n)$ converges for all n. The obtained WHNF is equivalent (some `letrec`-bindings are garbage collected, and some variable-to-variable chains are eliminated) to $(\texttt{letrec}\ w = (x\ x), k = (\lambda a.(\lambda b.a)), x = (\lambda y.(k(yy)))\ \texttt{in}\ \lambda u.w)$. Normal-order reduction of $(Y_2\ K\ r_1 \dots r_n)$ also always converges, where the WHNF is equivalent to the expression $(\texttt{letrec}\ w = (\mathit{fix}\ k), \mathit{fix} = (\lambda f.(f(\mathit{fix}\ f))), k = (\lambda a.(\lambda b.a))\ \texttt{in}\ (\lambda u.w))$. Thus $Y_1\ K \sim_{need} Y_2\ K$ and both expressions are greatest elements w.r.t. $\leq_{need}$.

For open expressions, we can lift the properties from L_{lazy}, which also follows from full abstraction of $N \circ W$ and from Lemma 3.10.

Lemma 6.7. *Let s, t be any expressions, and let the free variables of s, t be in $\{x_1, \dots, x_n\}$. Then $s \leq_{need} t \iff \lambda x_1, \dots, x_n.s \leq_{need} \lambda x_1, \dots, x_n.t$*

The results above imply the following theorem:

Main Theorem 6.8 $\leq_{need} = \leq^o_{b,need}$.

The main theorem implies that our embedding of the call-by-need letrec calculus into Abramsky's lazy lambda calculus is isomorphic w.r.t. the corresponding term models, i.e.:

Theorem 6.9. *The identical embedding $\iota : \mathcal{E}_{lazy} \to \mathcal{E}_{need}$ leads to an isomorphism between the term-models: Let the preorder, the quotients modulo $\sim_{lazy}$ and $\sim_{need}$, and the lifting of ι be marked with an overbar. Then $\bar{\iota} : \overline{\mathcal{E}_{lazy}} \to \overline{\mathcal{E}_{need}}$ is a bijection, and for all $s_1, s_2 \in \overline{\mathcal{E}_{lazy}} : s_1\ \overline{\leq_{lazy}}\ s_2 \iff \bar{\iota}(s_1)\ \overline{\leq_{need}}\ \bar{\iota}(s_2)$.*

7 The Call-by-Need Lambda Calculus of Ariola & Felleisen

For the sake of completeness we show that our results are transferable to the call-by-need lambda calculus with letrec of [AF97]. The syntax is identical to the calculus L_{need}, but the standard reduction strategy of [AF97] differs from our normal order reduction. In particular [AF97] do not provide a standard reduction strategy but an equational system from which we will derive a standard reduction.

We will show that the normal order reduction and the standard reduction corresponding to the equational system of [AF97] are interchangeable and thus define the same notion of contextual equivalence. As a further result we show that bisimilarity can also be based on the strategy according to [AF97] and coincides with contextual equivalence.

We recall the standard reduction strategy of [AF97]. We will denote the notions related to Ariola & Felleisen's calculus with a prefix or mark "AF", if necessary. First we introduce AF-evaluation contexts R_{AF} that play a role similar to our reduction contexts:

$$R_{AF} ::= [\cdot] \mid (R_{AF}\ s) \mid \texttt{letrec}\ Env\ \texttt{in}\ R_{AF} \mid \texttt{letrec}\ Env, x = R_{AF}\ \texttt{in}\ R_{AF}[x] \\ \mid \texttt{letrec}\ x_1 = R_{AF}, x_2 = R_{AF}[x_1], \ldots x_n = R_{AF}[x_{n-1}], Env\ \texttt{in}\ R_{AF}[x_n]$$

In Figure 3 the standard reductions (abbreviated as AF-reduction) of [AF97, Section 8] are shown where L is an $\mathcal{L}$-context as introduced in Sect. 3.2 and $R_{AF,i}, R'_{AF}, R''_{AF}$ are R_{AF}-contexts. The calculus of [AF97] uses the notion of a black hole which represents a cyclic dependency of the form $\texttt{letrec}\ x_1 = R_{AF}[x_n], x_2 = R_{AF}[x_1], \ldots x_n = R_{AF}[x_1]$. In contrast to [AF97], we do not consider a black hole to be an answer and therefore do not copy it in (deref) rules. This reflects the authors' intention, as shown by a similar copy restriction in [AK94].

(β_{need})	$R_{AF}[(\lambda x.s)\ r] \to R_{AF}[(\texttt{letrec}\ x = r\ \texttt{in}\ s)]$
(lift)	$R_{AF}[(\texttt{letrec}\ Env\ \texttt{in}\ L[\lambda x.s])\ r] \to R_{AF}[\texttt{letrec}\ Env\ \texttt{in}\ (L[\lambda x.s]\ r)]$
(deref)	$R_{AF,1}[\texttt{letrec}\ Env, x = \lambda y.s\ \texttt{in}\ R_{AF,2}[x]]$
	$\to R_{AF,1}[\texttt{letrec}\ Env, x = \lambda y.s\ \texttt{in}\ R_{AF,2}[\lambda y.s]]$
(deref_{env})	$R'_{AF}[\texttt{letrec}\ x_1 = \lambda y.s, x_2 = R_{AF,2}[x_1], \ldots, x_n = R_{AF,n}[x_{n-1}], Env\ \texttt{in}\ R''_{AF}[x_n]]$
	$\to R'_{AF}[\texttt{letrec}\ x_1 = \lambda y.s,$
	$\qquad x_2 = R_{AF,2}[\lambda y.s], \ldots, x_n = R_{AF,n}[x_{n-1}], Env\ \texttt{in}\ R''_{AF}[x_n]]$
(assoc)	$R_{AF,1}[\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ L[\lambda x.s])\ \texttt{in}\ R_{AF,2}[x]]$
	$\to R_{AF,1}[\texttt{letrec}\ Env_1, Env_2, x = L[\lambda x.s]\ \texttt{in}\ R_{AF,2}[x]]$
(assoc_{env})	$R'_{AF}[\texttt{letrec}\ x_1 = (\texttt{letrec}\ Env_2\ \texttt{in}\ L[\lambda x.s]),$
	$\qquad x_2 = R_{AF,2}[x_1], \ldots, x_n = R_{AF,n}[x_{n-1}], Env_1\ \texttt{in}\ R''_{AF}[x_n]]$
	$\to R'_{AF}[\texttt{letrec}\ Env_2, x_1 = L[\lambda x.s],$
	$\qquad x_2 = R_{AF,2}[x_1], \ldots, x_n = R_{AF,n}[x_{n-1}], Env_1\ \texttt{in}\ R''_{AF}[x_n]]$

Fig. 3. Reduction rules defining $\xrightarrow{AF}$

AF-answers are terms of the form $L[\lambda x.s]$. We write $s \xrightarrow{AF} t$, iff s is transformed into t by one of the rules in Fig. 3. If $s \xrightarrow{AF,*} v$ where v an AF-answer, then we write $s \downarrow_{AF} v$ or $s \downarrow_{AF}$, resp. if the answer v is not of interest. For the corresponding contextual approximation and equivalence we use $\leq_{AF}$ and $\sim_{AF}$ as symbols.

Compared to the reduction strategy in L_{need}, the AF-reduction performs the let-shiftings (lapp), (llet-in), (llet-e) as late as possible. A difference from L_{need} is that sometimes reduction steps must be performed in deeply nested lets. For instance, in $\texttt{letrec}\ x = (\texttt{letrec}\ y = \lambda z.z\ \texttt{in}\ (\lambda u.z)(\lambda uu))\ \texttt{in}\ x$ the L_{need} reduction will apply (llet-e) immediately, whereas AF will reduce $(\lambda u.z)(\lambda uu)$ first, and only then apply (assoc).

Theorem 7.1. $\downarrow_{need} = \downarrow_{AF}$, $\leq_{need} = \leq_{AF}$ *and* $\sim_{need} = \sim_{AF}$.

Proof. The proof of the first claim can be found in appendix E, Proposition E.5 and the other claims follows easily from the first.

Definition 7.2 (AF-simulation). *Let s, t be closed expressions and η be a binary relation on closed expressions. Then $s\ [\eta]_{AF}\ t$ holds iff $s{\downarrow_{AF}}v$ implies that $t{\downarrow_{AF}}w$, where v and w are answers, and for all closed letrec-free abstractions r and for $r = \Omega$: $(v\ r)\ \eta\ (w\ r)$. The relation $\leq_{b,AF}$ is defined to be the greatest fixpoint of $[\cdot]_{AF}$ within the binary relations on closed expressions. Its open extension is denoted with $\leq^o_{b,AF}$.*

It remains to show that $\leq^o_{b,AF} = \leq_{AF}$. As a first step we derive an alternative characterization of $\leq_{b,AF}$.

Proposition 7.3. *For closed $s, t \in L_{need}$ the relation $s \leq_{b,AF} t$ is equivalent to: $\forall n \geq 0$, and for all $r_i, i = 1, \ldots, n$ that may be letrec-free abstractions or Ω: $(s\ r_1 \ldots r_n){\downarrow_{AF}} \implies (t\ r_1 \ldots r_n){\downarrow_{AF}}$.*

Proof. For all closed expressions s, r it holds: $(s\ \ r)\ {\downarrow_{AF}}\ \ w \iff \exists$AF-answer $v : s \downarrow_{AF} v \wedge (v\ r) \downarrow_{AF}$. This follows since AF-reduction first evaluates the argument in function position of an application to an AF-answer before the second argument is evaluated. Now Theorem A.11 shows the claim.

Proposition 7.4. $\leq_{b,need} = \leq_{b,AF}$

Proof. Since $\downarrow_{need} = \downarrow_{AF}$ the previous proposition and Proposition 6.4 show the claim.

From Theorem 6.5 we already know that $\leq_{b,need}$ is equivalent to $\leq_{need}$ on closed expressions. Thus $\leq_{b,AF}$ is identical to $\leq_{need}$ on closed expressions. This easily extends to the open extension of $\leq_{b,AF}$. Thus we have:

Theorem 7.5. $\leq_{AF}\ = \leq^o_{b,AF}$

8 Conclusion

In this paper we show that co-inductive bisimulation, in the style of Howe, is equivalent to contextual equivalence in a deterministic call-by-need calculus with letrec (i.e. let with cyclic bindings). As a further work one may extend the proof to a call-by-need letrec calculus with case, constructors, and `seq`, but not to nondeterminism, since counterexamples exist that show that contextual equivalence cannot be characterized by the usual notion of bisimulation.

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.

AF97. Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Funct. Programming*, 7(3):265–301, 1997.

AFM+95. Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *POPL'95*, pages 233–246, San Francisco, California, 1995. ACM Press.

AK94. Z. M. Ariola and Jan Willem Klop. Cyclic Lambda Graph Rewriting. In *Proc. IEEE LICS*, pages 416–425. IEEE Press, 1994.

AO93. Samson Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. *Inf. Comput.*, 105(2):159–267, 1993.

Bar84. H. P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics.* North-Holland, Amsterdam, New York, 1984.

Fel91. Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1–3):35–75, December 1991.

Gol05. Mayer Goldberg. A variadic extension of Curry's fixed-point combinator. *Higher-Order and Symbolic Computation*, 18(3-4):371–388, 2005.

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.

Jef94. Alan Jeffrey. A fully abstract semantics for concurrent graph reduction. In *Proc. IEEE LICS*, pages 82–91, 1994.

MOW98. John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *J. Funct. Programming*, 8:275–317, 1998.

MSS10. Matthias Mann and Manfred Schmidt-Schauß. Similarity implies equivalence in a class of non-deterministic call-by-need lambda calculi. *Information and Computation*, 208(3):276 – 291, 2010.

SS07. Manfred Schmidt-Schauß. Correctness of copy in calculi with letrec. In *Term Rewriting and Applications (RTA-18)*, volume 4533 of *LNCS*, pages 329–343. Springer, 2007.

SSM08. Manfred Schmidt-Schauß and Elena Machkasova. A finite simulation method in a non-deterministic call-by-need calculus with letrec, constructors and case. In *Proc. of RTA 2008*, number 5117 in LNCS, pages 321–335. Springer-Verlag, 2008.

SSMS09. Manfred Schmidt-Schauß, Elena Machkasova, and David Sabel. Counterexamples to simulation in non-deterministic call-by-need lambda-calculi with letrec. Frank report 38, Inst. f. Informatik, Goethe-University, Frankfurt, 2009.

SSNSS08. Manfred Schmidt-Schauß, Joachim Niehren, Jan Schwinghammer, and David Sabel. Adequacy of compositional translations for observational semantics. In *5th IFIP TCS 2008*, volume 273 of *IFIP*, pages 521–535. Springer, 2008.

SSNSS09. Manfred Schmidt-Schauß, Joachim Niehren, Jan Schwinghammer, and David Sabel. Adequacy of compositional translations for observational semantics. Frank report 33, Inst. f. Informatik, Goethe-University, Frankfurt, 2009.

A Characterizations of Similarity in Deterministic Calculi

In this section we prove in a general way that for deterministic calculi (DC, see Def. 2.1), the applicative similarity that is usually defined as the greatest fixpoint of an operator on relations, is equivalent to the inductive definition using Kleene's fixpoint theorem. Moreover, we show that applicative similarity can be equivalently defined as $s \leq_b t$, iff for all $n > 0$ and closed expressions $r_i, i = 1, \ldots, n$, the implication $(s\ r_1 \ldots r_n)\downarrow \implies (t\ r_1 \ldots r_n)\downarrow$ holds, provided the calculus is convergence-admissible, which means that for all r: $(s\ r)\downarrow v \iff \exists v' : s\downarrow v' \wedge (v'\ r)\downarrow v$.

Definition A.1. *Let $D = (\mathcal{E}, \mathcal{C}, \rightarrow, \mathcal{W})$ be an untyped deterministic calculus and let $\mathcal{Q} \subseteq \mathcal{C}$ be a set of functions on expressions (i.e. $\forall Q \in \mathcal{Q} : Q :: \mathcal{E} \rightarrow \mathcal{E}$). Then the* $\mathcal{Q}$-experiment operator $[\cdot]_{\mathcal{Q}} :: (\mathcal{E} \times \mathcal{E}) \rightarrow (\mathcal{E} \times \mathcal{E})$*: is defined as follows*

$$e_1\, [\eta]_{\mathcal{Q}}\, e_2 \text{ iff } e_1 \downarrow v_1 \implies (e_2 \downarrow v_2 \wedge \forall Q \in \mathcal{Q} : Q(v_1)\, \eta\, Q(v_2))$$

We define $\mathcal{Q}$-similarity $\leq_{b,\mathcal{Q}}$ *as the greatest fixed point of* $[\cdot]_{\mathcal{Q}}$

In the following we implicitly assume $(\mathcal{E}, \mathcal{C}, \rightarrow, \mathcal{W})$ to be an untyped deterministic calculus and $\mathcal{Q} \subseteq \mathcal{C}$ be a set of functions on expressions.

Lemma A.2. *For all expressions $e_1, e_2 \in \mathcal{E}$ the following holds: $e_1 \leq_{b,\mathcal{Q}} e_2$ if, and only if $e_1 \downarrow v_1 \implies (e_2 \downarrow v_2 \wedge \forall Q \in \mathcal{Q} : Q(v_1) \leq_{b,\mathcal{Q}} Q(v_2))$.*

Proof. Since $\leq_{b,\mathcal{Q}}$ is a fixed point of $[\cdot]_{\mathcal{Q}}$, we have $\leq_{b,\mathcal{Q}} = [\leq_{b,\mathcal{Q}}]_{\mathcal{Q}}$. This equation is equivalent to the claim of the lemma.

Now we show that the operator $[\cdot]_{\mathcal{Q}}$ is monotonous and lower-continuous, and thus we can apply Kleene's fixpoint theorem to derive an alternative characterization of $\leq_{b,\mathcal{Q}}$.

Lemma A.3. *The operator $[\cdot]_{\mathcal{Q}}$ is monotonous w.r.t. set inclusion, i.e. for all binary relations η_1, η_2 on expressions $\eta_1 \subseteq \eta_2 \implies [\eta_1] \subseteq [\eta_2]$.*

Proof. Let $\eta_1 \subseteq \eta_2$ and $e_1\, [\eta_1]_{\mathcal{Q}}\, e_2$. The latter assumption implies that $e_1 \downarrow v_1 \implies (e_2 \downarrow v_2 \wedge \forall Q \in \mathcal{Q} : Q(v_1)\, \eta_1\, Q(v_2))$ holds. From $\eta_1 \subseteq \eta_2$ we have $e_1 \downarrow v_1 \implies (e_2 \downarrow v_2 \wedge \forall Q \in \mathcal{Q} : Q(v_1)\, \eta_2\, Q(v_2))$. Thus, $e_1\, [\eta_2]_{\mathcal{Q}}\, e_2$.

For infinite chains of sets $S_1, S_2 \ldots$, we define the greatest lower bound w.r.t. set-inclusion ordering as $\mathrm{glb}(\mathrm{S}_1, \mathrm{S}_2, \ldots) = \bigcap_{\mathrm{i}=1}^{\infty} \mathrm{S}_\mathrm{i}$.

Proposition A.4. $[\cdot]_{\mathcal{Q}}$ *is lower-continuous w.r.t. countably infinite descending chains $C = \eta_1 \supseteq \eta_2 \supseteq \ldots$, i.e.* $\mathrm{glb}([\mathrm{C}]_{\mathcal{Q}}) = [\mathrm{glb}(\mathrm{C})]_{\mathcal{Q}}$ *where $[C]_{\mathcal{Q}}$ is the infinite descending chain $[\eta_1]_{\mathcal{Q}} \supseteq [\eta_2]_{\mathcal{Q}} \supseteq \ldots$.*

Proof. "$\supseteq$": Since $\mathrm{glb(C)} = \bigcap_{\mathrm{i=1}}^{\infty} \eta_{\mathrm{i}}$, we have for all i: $\mathrm{glb(C)} \subseteq \eta_{\mathrm{i}}$. Applying monotonicity of $[\cdot]_{\mathcal{Q}}$ yields $[\mathrm{glb(C)}]_{\mathcal{Q}} \subseteq [\eta_{\mathrm{i}}]_{\mathcal{Q}}$ for all i. This implies $[\mathrm{glb(C)}]_{\mathcal{Q}} \subseteq \bigcap_{\mathrm{i=1}}^{\infty} [\eta_{\mathrm{i}}]_{\mathcal{Q}}$, i.e. $[\mathrm{glb(C)}]_{\mathcal{Q}} \subseteq \mathrm{glb}([\mathrm{C}]_{\mathcal{Q}})$.

"$\subseteq$": Let $(e_1, e_2) \in \mathrm{glb}([\mathrm{C}]_{\mathcal{Q}})$, i.e. for all i: $(e_1, e_2) \in [\eta_i]_{\mathcal{Q}}$. Unfolding the definition of $[\cdot]_{\mathcal{Q}}$ gives: $\forall i : e_1 \downarrow v_1 \implies (e_2 \downarrow v_2 \wedge \forall Q \in \mathcal{Q} : Q(v_1)\, \eta_i\, Q(v_2))$. Now we can move the universal quantifier for i inside the formula: $e_1 \downarrow v_1 \implies (e_2 \downarrow v_2 \wedge \forall Q \in \mathcal{Q} : \forall \mathrm{i}: Q(v_1)\, \eta_i\, Q(v_2))$. This is equivalent to $e_1 \downarrow v_1 \implies (e_2 \downarrow v_2 \wedge \forall Q \in \mathcal{Q} : Q(v_1)\, (\bigcap_{i=1}^{\infty} \eta_i)\, Q(v_2))$ or $e_1 \downarrow v_1 \implies (e_2 \downarrow v_2 \wedge \forall Q \in \mathcal{Q} : (Q(v_1), Q(v_2)) \in \mathrm{glb(C)})$ and thus $(e_1, e_2) \in [\mathrm{glb(C)}]_{\mathcal{Q}}$.

Definition A.5. *Let $\leq_{b,\mathcal{Q},i}$ for $i \in \mathbb{N}_0$ be defined as follows:*

$$\leq_{b,\mathcal{Q},0} = \mathcal{E} \times \mathcal{E} \quad \textit{and} \quad \leq_{b,i} = [\leq_{b,\mathcal{Q},i-1}]_{\mathcal{Q}}, \textit{ if } i > 0$$

Theorem A.6. $\leq_{b,\mathcal{Q}} = \bigcap_{i=1}^{\infty} \leq_{b,\mathcal{Q},i}$

Proof. This follows by Kleene's fixpoint theorem, since $[\cdot]_{\mathcal{Q}}$ is monotonous and lower-continuous, and since $\leq_{b,\mathcal{Q},i+1} \subseteq \leq_{b,\mathcal{Q},i}$ for all $i \geq 0$.

Note that this representation of $\leq_{b,\mathcal{Q}}$ allows *inductive* proofs to show similarity.

A.1 $\mathcal{Q}$-Similarity for Convergence-Admissible DC

In this section we show that under certain conditions $\mathcal{Q}$-similarity is identical to $\leq_{\mathcal{Q}}$ which is defined below and can be interpreted as a observational preorder, which uses the functions of $\mathcal{Q}$ as observers:

Definition A.7. *Let $(\mathcal{E}, \mathcal{C}, \rightarrow, \mathcal{W})$ be an untyped deterministic calculus, and $\mathcal{Q} \subseteq \mathcal{C}$. Then the relation $\leq_{\mathcal{Q}}$ is defined as follows:*

$$e_1 \leq_{\mathcal{Q}} e_2 \textit{ iff } \forall n \geq 0, Q_i \in \mathcal{Q} : Q_1(Q_2(\ldots(Q_n(e_1)))) \downarrow \implies Q_1(Q_2(\ldots(Q_n(e_2)))) \downarrow$$

Our characterization result will only apply if the underlying calculus is convergence-admissible w.r.t. $\mathcal{Q}$:

Definition A.8. *An untyped deterministic calculus $(\mathcal{E}, \mathcal{C}, \rightarrow, \mathcal{W})$ is* convergence-admissible w.r.t. $\mathcal{Q}$ *if, and only if $\forall Q \in \mathcal{Q}, e \in \mathcal{E} : Q(e) \downarrow v \iff \exists v' : e \downarrow v' \wedge Q(v') \downarrow v$*

We show some helpful properties of $\leq_{\mathcal{Q}}$:

Lemma A.9. *Let $(\mathcal{E}, \mathcal{C}, \rightarrow, \mathcal{W})$ be convergence-admissible w.r.t. $\mathcal{Q}$. Then the following holds:*

- $e_1 \leq_{\mathcal{Q}} e_2 \implies Q(e_1) \leq_{\mathcal{Q}} Q(e_2)$ *for all* $Q \in \mathcal{Q}$

– $e_1 \leq_{\mathcal{Q}} e_2, e_1 \downarrow v,$ *and* $e_2 \downarrow w \implies v \leq_{\mathcal{Q}} w$

Proof. The first part is easy to verify.

For the second part let $e_1 \leq_{\mathcal{Q}} e_2$, and $e_1 \downarrow v$, $e_2 \downarrow w$ hold. Assume that $Q_1(\ldots(Q_n(v))) \downarrow v'$ for some $n \geq 0$ where all $Q_i \in \mathcal{Q}$. Convergence-admissibility implies $Q_1(\ldots(Q_n(e_1))) \downarrow v'$. Now $e_1 \leq_{\mathcal{Q}} e_2$ implies $Q_1(\ldots(Q_n(e_2))) \downarrow w'$. Finally, convergence-admissibility (applied multiple times) shows that $e_2 \downarrow w$ and $Q_1(\ldots(Q_n(w))) \downarrow w'$ holds.

We prove that $\leq_{b,\mathcal{Q}}$ respects functions $Q \in \mathcal{Q}$ provided the underlying DC is convergence-admissible w.r.t. $\mathcal{Q}$:

Lemma A.10. *Let $(\mathcal{E}, \mathcal{C}, \rightarrow, \mathcal{W})$ be convergence-admissible w.r.t. $\mathcal{Q}$. Then for all $e_1, e_2 \in E : e_1 \leq_{b,\mathcal{Q}} e_2 \implies Q(e_1) \leq_{b,\mathcal{Q}} Q(e_2)$ for all $Q \in \mathcal{Q}$*

Proof. Let $e_1 \leq_{b,\mathcal{Q}} e_2$, $Q_0 \in \mathcal{Q}$, and $Q_0(e_1) \downarrow v_1$. By convergence admissibility $e_1 \downarrow v_1'$ holds and $Q_0(v_1') \downarrow v_1$. Since $e_1 \leq_{b,\mathcal{Q}} e_2$ this implies $e_2 \downarrow v_2'$ and for all $Q \in \mathcal{Q} : Q(v_1') \leq_{b,\mathcal{Q}} Q(v_2')$. Hence, from $Q_0(v_1') \downarrow v_1$ we derive $Q_0(v_2') \downarrow v_2$. By convergence admissibility this also implies $Q_0(e_2) \downarrow v_2$.

It remains to show for all $Q \in \mathcal{Q}$: $Q(v_1) \leq_{b,\mathcal{Q}} Q(v_2)$: Since $Q_0(v_1') \downarrow v_1$ and $Q_0(v_2') \downarrow v_2$, applying Lemma A.2 to $Q_0(v_1') \leq_{b,\mathcal{Q}} Q_0(v_2')$ implies $Q(v_1) \leq_{b,\mathcal{Q}} Q(v_2)$ for all $Q \in \mathcal{Q}$.

We now prove that $\leq_{\mathcal{Q}}$ and $\mathcal{Q}$-similarity coincide for convergence-admissible DC:

Theorem A.11. *Let $(\mathcal{E}, \mathcal{C}, \rightarrow, \mathcal{W})$ be convergence-admissible w.r.t. $\mathcal{Q}$. Then $\leq_{\mathcal{Q}} = \leq_{b,\mathcal{Q}}$ holds.*

Proof. "$\subseteq$": Let $e_1 \leq_{\mathcal{Q}} e_2$. We use Theorem A.6 and show $e_1 \leq_{b,\mathcal{Q},i} e_2$ for all i. We use induction on i. The base case ($i = 0$) obviously holds. Let $i > 0$ and let $e_1 \downarrow v_1$. Then $e_1 \leq_{\mathcal{Q}} e_2$ implies $e_2 \downarrow v_2$. Thus, it is sufficient to show that $Q(v_1) \leq_{b,\mathcal{Q},i-1} Q(v_2)$ for all $Q \in \mathcal{Q}$: As induction hypothesis we use that $\leq_{\mathcal{Q}} \subseteq \leq_{b,\mathcal{Q},i-1}$ holds. Using Lemma A.9 twice and $e_1 \leq_{\mathcal{Q}} e_2$, we have $Q(v_1) \leq_{\mathcal{Q}} Q(v_2)$. The induction hypothesis shows that $Q(v_1) \leq_{b,\mathcal{Q},i-1} Q(v_2)$.

"$\supseteq$": Let $e_1 \leq_{b,\mathcal{Q}} e_2$. By induction on the number n of observers we show $\forall n, Q_i \in \mathcal{Q} : Q_1(\ldots(Q_n(e_1))) \downarrow \implies Q_1(\ldots(Q_n(e_2))) \downarrow$. The base case follows from $e_1 \leq_{b,\mathcal{Q}} e_2$. For the induction step we use the following induction hypothesis: $e_1' \leq_{b,\mathcal{Q}} e_2' \implies \forall j < n, Q_i \in \mathcal{Q} : Q_1(\ldots(Q_j(e_1'))) \downarrow \implies Q_1(\ldots(Q_j(e_2'))) \downarrow$ for all e_1', e_2'. Let $Q_1(\ldots(Q_n(e_1))) \downarrow$. From Lemma A.10 we have $e_1'' \leq_{b,\mathcal{Q}} e_2''$, where $e_i'' = Q_n(e_i)$. Now the induction hypothesis shows that $Q_1(\ldots(Q_{n-1}(e_1''))) \downarrow \implies Q_1(\ldots(Q_{n-1}(e_2''))) \downarrow$ and thus $Q_1(\ldots(Q_n(e_2))) \downarrow$.

B The Lazy Lambda Calculus: Citations and Sketch of Proofs

In this section we prove Theorem 3.8 by using the results of the appendix A and the proved equivalences in [How89,How96,Abr90]. For basic definitions and for proofs of confluence of the unrestricted reduction see [Bar84].

Theorem B.1. *In L_{lazy}, all the following relations are identical:*

1. $\leq_{lazy}$.
2. $\leq^o_{b,lazy}$.
3. *The relation $\leq_{lazy,1}$, defined as: $s \leq_{lazy,1} t$ iff for all closing contexts C: $C[s]\downarrow \implies C[t]\downarrow$.*
4. *The relation $\leq_{lazy,2}$, defined as: $s \leq_{lazy,2} t$ iff for all closed contexts C and all closing substitutions: $C[\sigma(s)]\downarrow \implies C[\sigma(t)]\downarrow$.*
5. *The relation $\leq_{lazy,3}$, defined as: $s \leq_{lazy,3} t$ iff for all multi-contexts $M[\cdot,\ldots,\cdot]$ and all substitutions: $M[\sigma(s),\ldots,\sigma(s)]\downarrow \implies M[\sigma(t),\ldots,\sigma(t)]\downarrow$.*
6. *The relation $\leq_{lazy,4}$, defined as: $s \leq_{lazy,4} t$ iff for all contexts $C[\cdot]$ and all substitutions: $C[\sigma(s)]\downarrow \implies C[\sigma(t)]\downarrow$.*
7. *The relation $\leq^o_{b,lazy,1}$ where $\leq_{b,lazy,1}$ is defined using the Kleene-construction: I.e. $\leq_{b,lazy,1} = \bigcap_{i\geq 0} \leq'_{b,i}$, where $\leq'_{b,0}$ is the full relation, and $\leq'_{b,i+1} := [\leq'_{b,i}]_{lazy}$ for all i.*
8. *The relation $\leq^o_{b,lazy,2}$ where $\leq_{b,lazy,2}$ is defined as: $s \leq_{b,lazy,2} t$ iff for all $n \geq 0$ and all closed expressions $r_i, i = 1,\ldots,n$: $s\ r_1 \ldots r_n\downarrow \implies t\ r_1 \ldots r_n\downarrow$.*
9. *The relation $\leq^o_{b,lazy,3}$, where $\leq_{b,lazy,3}$ is defined as: $s \leq_{lazy,3} t$ iff for all $n \geq 0$ and all $r_i, i = 1,\ldots,n$, where r_i may be a closed abstraction or Ω: $s\ r_1 \ldots r_n\downarrow \implies t\ r_1 \ldots r_n\downarrow$.*
10. *The relation $\leq^o_{b,lazy,4}$, where $\leq_{b,lazy,4}$ is the greatest fixpoint of the operator $[\cdot]_{lazy,a\Omega}$ on closed expressions: for all $n \geq 0$ and all $r_i, i = 1,\ldots,n$, where r_i may be a closed abstraction or Ω: Then $s\ [\eta]_{lazy,a\Omega}\ t$ holds iff $s\downarrow\lambda x.s'$ implies $t\downarrow\lambda x.t'$ and for all closed L_{lazy}-abstractions r and $r = \Omega$, the relation $s'[r/x]\ \eta\ t'[r/x]$ holds. The relation $\leq_{b,lazy,4}$ is defined as the greatest fixpoint of the operator $[\cdot]_{lazy,a\Omega}$.*

Proof. – (2) $\iff$ (3): This was proved in [How89,How96].

– (2) $\iff$ (4): This was proved in [Abr90]. This also implies that (β) is correct for $\sim_{lazy,2}$ and by the previous item it also correct for $\sim_{lazy,1}$ (where $\sim_{lazy_i} = \leq_{lazy,i} \cap \geq_{lazy,i}$).

– (1) $\iff$ (3): The "$\Rightarrow$"-direction is obvious. For the other direction let $s_1 \leq_{lazy,1} s_2$ and let C be a context such that $FV(C[s_1]) \cup FV(C[s_2]) \neq \emptyset = \{x_1,\ldots,x_n\}$ and let $C[s_1]\downarrow$, i.e. $C[s_1] \xrightarrow{lazy,*} \lambda x.t_1$. Let $C' = (\lambda x_1,\ldots,x_n.C)\ \underbrace{\Omega \ldots \Omega}_{n\text{-times}}$. Then $s_i \xrightarrow{lazy,\beta} s'_i = C[s_i][\Omega/x_1,\ldots,\Omega/x_n]$ for $i = 1,2$. It is easy to verify that the reduction for $C[s_1]$ can also be performed for s'_i, since every reduction in the sequence $C[s_1] \xrightarrow{lazy,*} \lambda x.t_1$ cannot be of the form $R[x_i]$ with R being a reduction context. Thus $s_i\downarrow$. Since $C'[s_i]$ must be closed for $i = 1,2$, the precondition implies $C'[s_2]\downarrow$ and also $s'_2\downarrow$. Wlog. let $s'_2 \xrightarrow{lazy,*} \lambda y.t_2$. It is easy to verify that every term in this sequence cannot be of the form $R[\Omega]$ where R is a reduction context, since otherwise the reduction would not terminate (since $R[\Omega] \xrightarrow{lazy,+} R[\Omega]$). This implies that we can replace the Ω-expression by the free variables, i.e. that $C[s_2]\downarrow$. Note that this also shows by the previous items that (β) is correct for $\sim_{lazy}$.

- (6) $\iff$ (1) One direction is trivial. For the other direction let $s \leq_{lazy} t$ and let C be a context, σ be a substitution, such that $C[\sigma(s)] \downarrow$. Let $\sigma = \{x_1 \to s_1, \ldots x_n \to s_n\}$ and let $C' = C[(\lambda x_1, \ldots, x_n.[\cdot])\ s_1\ \ldots\ s_n]$. Then $C'[s] \xrightarrow{\beta,n} C[\sigma(s)]$. Since (β)-reduction is correct for $\sim_{lazy}$, we have $C'[s] \downarrow$. Applying $s \leq_{lazy} t$ yields $C'[t] \downarrow$. Since $C'[t] \xrightarrow{\beta,n} C[\sigma(t)]$ and (β) is correct for $\sim_{lazy}$, we have $C[\sigma(t)] \downarrow$.
- (5) $\iff$ (6): Obviously, $s \leq_{lazy,3} t \implies s \leq_{lazy,4} t$. We show the other direction by induction on n – the number of holes in M – that for all expressions s, t: $s \leq_{lazy,4} t$ implies $M[\sigma(s), \ldots \sigma(s)] \downarrow \implies M[\sigma(t), \ldots \sigma(t)] \downarrow$. The base cases for $n = 0, 1$ are obvious. For the induction step assume that M has $n > 1$ holes. Let $M' = M[\sigma(s), \cdot_2, \ldots, \cdot_n]$ and $M'' = M[\sigma(t), \cdot_2, \ldots, \cdot_n]$ Then obviously $M'[\sigma(s), \ldots, \sigma(s)] = M[\sigma(s), \ldots, \sigma(s)]$ and thus $M'[\sigma(s), \ldots, \sigma(s)] \downarrow$. For $C = M[\cdot_1, \sigma(s), \ldots, \sigma(s)]$ we have $C[\sigma(s)] = M'[\sigma(s), \ldots, \sigma(s)]$ and $C[\sigma(t)] = M''[\sigma(s), \ldots, \sigma(s)]$ Since $C[\sigma(s)] \downarrow$, the relation $s \leq_{lazy,4} t$ implies that $C[\sigma(t)] \downarrow$ and thus $M''[\sigma(s), \ldots, \sigma(s)] \downarrow$. Now the induction hypothesis shows that $M''[\sigma(t), \ldots, \sigma(t)] \downarrow$, since the number of holes of M'' is strictly smaller than n. Since $M''[\sigma(t), \ldots, \sigma(t)] = M[\sigma(t), \ldots, \sigma(t)]$ we have $M[\sigma(t), \ldots, \sigma(t)] \downarrow$.
- (7) $\iff$ (2): This was proven in [Abr90] and also follows from Theorem A.6, since the lazy lambda calculus is a DC we can choose $\mathcal{Q} = \{[\cdot]\ t \mid t \text{ is a closed expression}\}$ and since $(\lambda x.s)\ t \xrightarrow{lazy} s[t/x]$.
- (8) $\iff$ (2): This follows for the relations on closed expressions by Theorem A.11, since the DC for L_{lazy} described before is convergence admissible. It also holds for the extensions to open expressions, since the construction for the open extension is identical for both relations.
- (9) $\iff$ (10): This follows for the relations on closed expressions by Theorem A.11, since the DC for L_{lazy} with $\mathcal{Q} = \{[\cdot]\ t \mid t \text{ is an abstraction or } \Omega\}$ is convergence admissible. It also holds for the extensions to open expressions, since the construction for the open extension is identical for both relations.
- (8) $\iff$ (9): One direction is trivial. For the other direction let $s \leq^o_{b,lazy,3} t$ and let $\sigma(s)\ r_1\ \ldots\ r_n \downarrow$ for some closing substitution σ and closed expressions $r_1, \ldots, r_n$. Since (β) is correct for $\sim_{lazy}$ for every expression r_i holds $r_i \sim_{lazy} r'_i$ where r'_i is either a closed abstraction or Ω. Thus $(\sigma(s)\ r_1\ \ldots\ r_n) \sim_{lazy} (\sigma(s)\ r'_1\ \ldots\ r'_n)$, which implies $(\sigma(s)\ r'_1\ \ldots\ r'_n) \downarrow$. Now $s \leq^o_{b,lazy,3} t$ shows $(\sigma(t)\ r'_1\ \ldots\ r'_n) \downarrow$. Applying contextual equivalence again shows $(\sigma(t)\ r_1\ \ldots\ r_n) \downarrow$, i.e. $s \leq^o_{b,lazy,2} t$. □

C Convergence Equivalence of N

The proof of convergence equivalence of the translation N may be performed directly, but it would be complicated due to the additional (β)-reductions required in L_{lazy}. For this technical reason we provide a second translation N', which requires a special treatment for the translation of contexts and uses a substitution function σ:

Definition C.1. *The translation* $N' :: L_{name} \to L_{lazy}$ *for expressions is recursively defined as:*

- $N'(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ r) = \sigma(N'(r))$ *where*

$$\begin{aligned} \sigma &= \{x_1 \mapsto U_1, \ldots x_n \mapsto U_n\} \\ U_i &= (X'_i X'_1 \ldots X'_n), \\ X'_i &= \lambda x_1 \ldots x_n . F_i(x_1 x_1 \ldots x_n) \ldots (x_n x_1 \ldots x_n), \\ F_i &= \lambda x_1, \ldots, x_n . N'(s_i). \end{aligned}$$

- $N'(s_1\ s_2) = (N'(s_1)\ N'(s_2))$
- $N'(\lambda x.s) = \lambda x.N'(s)$
- $N'(x) = x$.

The extension of N' *to contexts requires that contexts are represented using an additional substitution, i.e. a context translates into a pair* (D, σ) *acting as a function on expressions. Filling the hole of a context* (D, σ) *by an expression* t *is by definition* $(D, \sigma)(t) = D[\sigma(t)]$. *The translation is defined as*

$N'(C) = (C', \sigma)$,	*if the hole of* C *is not inside the right hand side of any* `letrec`*-binding.* C' *and* σ *are calculated by applying* N' *to* C*: for calculating* C' *the hole of* C *is treated as a constant, and* σ *is the substitution affecting the hole of* C'.
$N'(C) = (N(C), \sigma_\emptyset)$,	*if the hole of* C *is inside the right hand side of some* `letrec`*-binding,* $N(C)$ *treats the hole as a constant and* $\sigma_\emptyset$ *is the empty substitution*

This translation guarantees that the hole of contexts is not duplicated by the translation. Note that using the translation N' for all contexts (without using N) would lead to scoping problems when the context hole becomes a part of an expression U_i.

Lemma C.2. *The translation* N *is equivalent to* N' *on expressions, i.e. for all* L_{name}*-expressions* t *the equivalence* $N(t) \sim_{lazy} N'(t)$ *holds.*

Proof. This follows from the definitions and correctness of beta-reduction in L_{lazy} by Theorem 3.9.

Now we first prove that the translation N' is convergence-equivalent. Due to Lemma C.2 this will also imply that N is convergence-equivalent. All reduction contexts in L_{name} translate into reduction contexts in L_{lazy}, since removing the case of `letrec` from the definition of a reduction context in L_{name} results in the reduction context definition in L_{lazy}. However, this can not be reversed, since a `letrec`expression applied to an expression is a (lapp) redex and does not allow the marking algorithm to descend into the expression inside a `letrec`. The lemma below gives a more precise characterization of this relation:

Lemma C.3. *If C is a reduction context in L_{name}, then $N'(C) = C'[\sigma(\cdot)]$, where C' is a reduction context in L_{lazy} and σ is a substitution.*

If C is a reduction context in L_{lazy}, and $N'(C') = (C, \sigma)$ for some substitution σ and some context C' in L_{name}, then C' is a $\mathcal{W}$-context, which are defined as $\mathcal{W} = L[\mathcal{W}] \mid A[\mathcal{W}] \mid [\cdot]$ ($\mathcal{L}$- and $\mathcal{A}$-contexts are defined as before in Sect. 3.2).

Proof. The first claim can be shown by structural induction on C. It holds, since applications are translated into applications and `letrec`-expressions are translated into substitutions.

The other part can be shown by induction on the number of translations steps. It is easy to observe that the definition of a reduction context in L_{name} does not descend into `letrec`-expressions below applications. For instance, in ((`letrec` *Env* `in` $((\lambda x.s)\ t))\ r$) the reduction contexts are $[\cdot]$ and $([\cdot]\ r)$ and the redex is (lapp), i.e. the reduction context does not reach $((\lambda x.s)\ t)$. In general, applications in such cases appear in contexts of the form $\mathcal{W}$, as defined in the lemma. By examining the expression definition we observe that these (lapp)-redexes are the only cases where non-reduction contexts may be translated into reduction contexts.

We inspect how WHNFs and values of both calculi are related w.r.t.l N':

Lemma C.4. *Let s be a L_{name}-expression. Then s is a WHNF in L_{name} iff $N'(s)$ is a WHNF in L_{lazy}.*

Proof. If $s = L[\lambda x.t]$ then $N'(s) = \lambda x.\sigma(N'(t))$ is a WHNF. For the other direction we assume that $N'(s)$ is an abstraction in L_{lazy}. Then s cannot have a beta redex or a (lapp) redex since both of these cases translate into an application. To show that s cannot have a (cp)-redex, we consider the translation of (L[`letrec` *Env* `in` $R[x]$]): $N'(L[$`letrec` *Env* `in` $R[x]])$ results in $R'[\sigma(N'(x))]$ where $N'(L[$`letrec` *Env* `in` $R]) = (R', \sigma)$, $N'(x) = x$, and σ maps x into an application. R' is a reduction context by Lemma C.3, and since $\sigma(N'(x))$ is an application, the expression $N'(s)$ has a beta-redex in a reduction context.

As the last case we need to verify that if s is irreducible but not a L_{name}-WHNF then $N'(s)$ cannot be a WHNF. The only case needed to be inspected is $s = R[x]$ where x is a free variable, but then $N'(s) = R'[x]$ where R' is a L_{lazy}-reduction context and x is free in $N'(s)$, i.e. $N'(s)$ is not a WHNF. Therefore it cannot be the case that $N'(s)$ is a WHNF when s is not.

In the remaining part of this section we use reduction diagrams to show that N' reflects and preserves convergence.

Transferring L_{name}-reductions into L_{lazy}-reductions

In this section we analyze how normal order reduction in L_{name} can be transferred into L_{lazy} via N'. We illustrate this by using reduction diagrams. For $s_1 \xrightarrow{name} s_2$ we analyze how the reduction transfers to $N'(s_1)$. The cases are on the rule used in $s_1 \xrightarrow{name} s_2$:

– (β) Let $s = R[(\lambda x.t)r]$ be an expression in L_{name}, where R is a reduction context. We observe that in L_{name}: $s \xrightarrow{name} s' = R[t[r/x]]$. Let $N'(R[\cdot]) = (R', \sigma)$. Then the translations for s and s' are as follows:

$$N'(s) = R'[\sigma(N'((\lambda x.t)\ r))] = R'[(\lambda x.\sigma(N'(t)))\ \sigma(N'(r))]$$
$$N'(s') = N'(R[t[r/x]]) = R'[\sigma(N'(t[r/x]))] = R'[\sigma(N'(t))[\sigma(N'(r))/x]]$$

Since R' is a reduction context in L_{lazy}, this shows $N'(s) \xrightarrow{lazy} N'(s')$. Thus we have the following diagram:

$$\begin{array}{ccc} \cdot & \xrightarrow{N'} & \cdot \\ {\scriptstyle name,\beta}\downarrow & & \downarrow{\scriptstyle lazy,\beta} \\ \cdot & \xrightarrow{N'} & \cdot \end{array}$$

– *(cp)* Consider the (cp) reduction. Without loss of generality we assume that x_1 is the variable that gets substituted:

$$t = L[\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ R[x_1]] \xrightarrow{name,cp}$$
$$t' = L[\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ R[s_1]]$$

Let $N'(L) = ([\cdot], \sigma_L)$, $N'(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ [\cdot]) = ([\cdot], \sigma_{Env})$, and $N'(R) = (R', \sigma_R)$ where R' is a reduction context. Then

$$\begin{aligned} N'(t) &= \sigma_L(\sigma_{Env}(R'[\sigma_R(x_1)])) = \sigma_L(\sigma_{Env}(R'))[\sigma_L(\sigma_{Env}(\sigma_R(x_1)))] \\ &= \sigma_L(\sigma_{Env}(R'))[\sigma_L(\sigma_{Env}(x_1))] \end{aligned}$$

where the last step follows, since x_1 cannot be substituted by σ_R, and

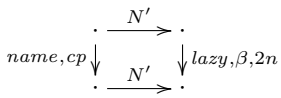
$$N'(t') = \sigma_L(\sigma_{Env}(R'))[\sigma_L(\sigma_{Env}(N'(s_1)))]$$

where it is again necessary to observe that $\sigma_R(s_1) = s_1$ must hold. The context $R'' = \sigma_L(\sigma_{Env}(R'))$ must be a reduction context, since R' is a reduction context. This means that we need to show that $R''[\sigma_L(\sigma_{Env}(x_1))] \xrightarrow{lazy,*} R''[\sigma_L(\sigma_{Env}(N'(s_1)))]$ holds.
By definition of the translation N' (Definition C.1) $\sigma_L(\sigma_{Env}(x_1)) = U_1 = (X_1'X_1' \ldots X_n')$, where $X_i' = \lambda x_1 \ldots x_n.F_i(x_1x_1 \ldots x_n) \ldots (x_nx_1 \ldots x_n)$, and $F_i = \lambda x_1, \ldots, x_n.\sigma_L(N'(s_i))$, i.e., $N'(t) = R''[U_1]$.
Performing the applications, we transform U_1 in $2n$ steps as

$$\begin{aligned} & (\lambda x_1, \ldots, x_n.(F_1(x_1x_1 \ldots x_n) \ldots (x_nx_1 \ldots x_n))\ X_1'\ \ldots\ X_n' \\ \xrightarrow{\beta,n}\ & F_1\ (X_1'X_1' \ldots X_n')\ \ldots\ (X_n'X_1' \ldots X_n') \\ =\ & (\lambda x_1, \ldots, x_n.\sigma_L(N'(s_1))\ (X_1'X_1' \ldots X_n')\ \ldots\ (X_n'X_1' \ldots X_n') \\ \xrightarrow{\beta,n}\ & \sigma_L(N'(s_1))[U_1/x_1, \ldots, U_n/x_n]. \end{aligned}$$

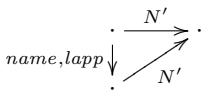
Obviously, for all reduction contexts in L_{lazy} we have: $s \xrightarrow{no} t$ implies $R[s] \xrightarrow{no} R[t]$. Hence $N'(t) \xrightarrow{2n,lazy,\beta} R''[\sigma_L(N'(s_1))[U_1/x_1, \ldots, U_n/x_n]]$

holds. Since $x_1, \ldots, x_n$ cannot occur free in L, the last expression is the same as $R''[\sigma_L(\sigma_{Env}(N'(s_1)))]$. Thus the diagram is as follows:



where n is the number of bindings in the `letrec`-subexpression where the copied binding is.

– *(lapp)* The diagram for this case is:



This is because the argument in the (lapp) reduction does not depend on the `letrec` environment:

$$R[(\texttt{letrec}\ Env\ \texttt{in}\ s)\ t] \xrightarrow{name} R[(\texttt{letrec}\ Env\ \texttt{in}\ (s\ t))]$$

Here free variables of t do not depend on Env so the translation of t does not change by adding Env. I.e., for $N'(R) = (R', \sigma_R)$ and $N'(\texttt{letrec}\ Env\ \texttt{in}\ [\cdot]) = ([\cdot], \sigma_{Env})$ we have $N'(R[(\texttt{letrec}\ Env\ \texttt{in}\ s)t]) = R'[\sigma_R(\sigma_{Env}(N'(s))\ \ N'(t))] = R'[\sigma_R(\sigma_{Env}(N'(s)\ \ N'(t))] = N'(R[(\texttt{letrec}\ Env\ \texttt{in}\ (s\ t)]).$

Transferring L_{lazy}-reductions into L_{name}-reductions

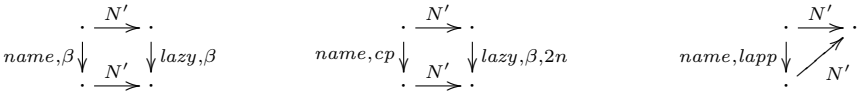
We will now analyze how normal order reductions for $N'(s)$ can be transferred into normal order reductions for s in L_{name}.

Let s be an L_{name} expression and $N'(s) \xrightarrow{lazy} s'$. We split the argument into three cases based on whether or not a normal order reduction is applicable to s:

– If $s \xrightarrow{(name)} t'$, then we can use the already developed diagrams, since normal-order reduction in both calculi is unique.
– s is irreducible and a WHNF. This case cannot happen, since then $N'(s)$ would also be a WHNF (see Lemma C.4) and thus irreducible.
– s is irreducible but not a WHNF. Then s must be of the form $R[x]$ where x is a free variable in s. For the translation $N'(s)$ this would result in $R'[\sigma_R(x)]$ where $N'(R) = (R', \sigma_R)$, R' is a reduction context in L_{lazy}. Since σ_R cannot substitute x (x is free), we have $N'(s) = R'[x]$ which is impossible, since $N'(s)$ is reducible by the assumption $N'(s) \xrightarrow{lazy} s'$.

We summarize the diagrams in the following lemma:

Lemma C.5. *Normal-order reductions in L_{name} can be transferred into reductions in L_{lazy}, and vice versa, by the following diagrams:*



Proposition C.6. *N' and N are convergence equivalent, i.e. for all L_{name}-expressions t: $t{\downarrow_{name}} \iff N'(t){\downarrow_{lazy}}$ ($t{\downarrow_{name}} \iff N(t){\downarrow_{lazy}}$, resp.).*

Proof. We first prove convergence equivalence of N': Suppose $t{\downarrow_{name}}$. Let $t \xrightarrow{name,k} s$ where s is a WHNF. We show that there exists an L_{lazy}-WHNF s' such that $N'(t) \xrightarrow{lazy,*} s'$ by induction on k. The base case follows from Lemma C.4. The induction step follows by applying a diagram from Lemma C.5 and then using the induction hypothesis.

For the other direction we assume that $N'(t){\downarrow_{lazy}}$, i.e. there exists a WHNF $s' \in L_{lazy}$ s.t. $N'(t) \xrightarrow{lazy,k} s'$. By induction on k we show that there exists a L_{name}-WHNF s such that $t \xrightarrow{name,*} s$. The base case is covered by Lemma C.4. The induction step uses the diagrams. Here it is necessary to observe that the (lapp)-diagram cannot be applied infinitely often without being interleaved with other reductions. This is obvious, since there are no infinite sequences of (lapp)-reductions.

It remains to show convergence equivalence of N: Let $s{\downarrow_{name}}$ then $N'(s){\downarrow_{lazy}}$, since N' is convergence equivalent. Lemma C.2 implies $N'(s) \sim_{lazy} N(s)$ and thus $N(s){\downarrow_{lazy}}$ must hold. For the other direction Lemma C.2 shows that $N(s){\downarrow_{lazy}}$ implies $N'(s){\downarrow_{lazy}}$. Using convergence equivalence of N' yields $s{\downarrow_{name}}$.

D Correctness of Reduction Rules in L_{need}

A program transformation T is a binary relation on expressions. A transformation T is called *correct*, if it is included in the contextual equality, i.e. $T \subseteq \sim_c$. Note that a correct program transformation can be applied in any context since contextual equivalence is a congruence. In this section we show that all calculus reductions of L_{need} preserve contextual equivalence.

Proposition D.1. *The transformation (lbeta), (llet-in), and (lapp) are correct program transformations for L_{need}.*

Proof. Let s, t be closed L_{need}-expressions such that $s \xrightarrow{a} t$ where a is a (lbeta), (llet-in), or (lapp) reduction. Recall that $\leq_{b,need}$ is defined in Definition 6.3. We show that $S := (\{(s,t),(t,s)\} \cup \leq_{b,need}) \subseteq \leq_{b,need}$: by proving that S is $[\,]_{need}$-dense, i.e. $S \subseteq [S]_{need}$. For $(s_1,t_1) \in \leq_{b,need}$ obviously the following holds: $(s_1,t_1) \in [S]_{need}$, since $\leq_{b,need}$ is the greatest fixpoint of $[\,]_{need}$. Let $s \xrightarrow{a} t$, and $s \downarrow_{need} v$. Since $s \xrightarrow{a} t$ is the first step of the unique normal order reduction of s, we have $t \downarrow_{need} v$. Let r be a closed abstraction or Ω. Obviously $((v\ r),(v\ r)) \in \leq_{b,need} \subseteq S$. Now assume that $t \downarrow_{need} v$. Since $s \xrightarrow{a} t$ is a normal order reduction, we can reason as before and thus have S is $[\,]_{need}$-dense. This shows $s \leq_{b,need} t$ and $t \leq_{b,need} s$. Since s, t were chosen arbitrarily this extends to open terms in the usual way, and thus we have that (lbeta), (llet-in) and (lapp) are correct program transformations.

Note that we cannot reason in the same way for (llet-e), (cp-in), and (cp-e), since they are not necessarily normal order reductions of L_{need} even if they are applied in the empty context. E.g. the transformation $\texttt{letrec}\ x = (\texttt{letrec}\ y = \lambda w.w\ \texttt{in}\ y)\ \texttt{in}((\lambda u.u)\ x) \to \texttt{letrec}\ x = y, y = \lambda w.w\ \texttt{in}((\lambda u.u)\ x)$ is a (llet-e)-transformation, but not a normal order reduction.

Proposition D.2. *The transformation (llet-e) is a correct program transformation.*

Proof. Let s, t be closed L_{need}-expressions such that $s \xrightarrow{llet-e} t$. We use Proposition 6.4 to show that $s \sim_{b,need} t$ holds. First we show $s \leq_{b,need} t$: With $\xrightarrow{i,llet-e}$ we denote *internal* (llet-e)-reductions, i.e. (llet-e)-reductions that are not normal order. Let r_i, $i = 1, \ldots, n$, $n \geq 0$ be closed abstractions or Ω. Let $s' = s\ r_1\ \ldots\ r_n \downarrow_{need}$. If s' is a WHNF, then $t' = t\ r_1\ \ldots\ r_n$ must be a WHNF, too. Now assume that s' is not a WHNF. We split the proof into two cases:

- $n = 0$ Then a case analysis shows that the following diagrams always holds (where the normal order reduction cannot be a (lapp)-reduction):

$$\begin{array}{ccc} s & \xrightarrow{i,llet-e} & t \\ {\scriptstyle need}\downarrow & & \downarrow{\scriptstyle need} \\ s_0 & \xrightarrow[i \vee n,llet-e]{} & t_0 \end{array}$$

- $n > 0$: Then the following diagram holds:

$$\begin{array}{ccc} s & \xrightarrow{i,llet-e} & t \\ {\scriptstyle need,lapp}\downarrow & & \downarrow{\scriptstyle need,lapp} \\ s_0 & \xrightarrow[i \vee n,llet-e]{} & t_0 \end{array}$$

Now we can perform an induction on the length of the reduction sequences $s' \xrightarrow{need,*} v$ where v is a WHNF, to show that $t' \downarrow_{need}$: The base case is already covered. For the other cases we apply the overlapping diagrams to construct a successful normal order reduction sequence for t'. Either the bottom reduction in the diagram is $\xrightarrow{i,llet-e}$ and we can use the induction hypothesis, or it is a $\xrightarrow{n,llet-e}$-reduction, and the reduction sequences are joined.

Now we show $t \leq_{b,need} s$: Let $t' = t\ r_1\ \ldots\ r_n \downarrow_{need}$ where r_i are closed abstractions or Ω. If t' is a WHNF, then $s' = s\ r_1\ \ldots\ r_n$ must be a WHNF, too. If $s \to t$ is a normal order reduction, then the claim follows. For the other cases we can construct a successful reduction sequence for s' where the overlappings are as before.

After extending this argument to open expressions in the obvious manner the claim follows.

Proposition D.3. *The transformations (cp-in) and (cp-e) are correct.*

Proof. This has been proved in [SS07] for a more general rule, which allows to copy arbitrary expressions.

We summarize that all calculus reductions of L_{need} are correct:

Theorem D.4. *The reduction rules (lbeta), (cp-in), (cp-e), (llet-e), (llet-in), and (lapp) are correct program transformations in L_{need}.*

E The Call-by-Need Lambda Calculus of Ariola & Felleisen

In this section we analyze the call-by-need lambda calculus with letrec of [AF97]. We first prove that AF-convergence and L_{need}-convergence coincide. It is easy to verify that AF-answers reduce to L_{need}-WHNFs:

Lemma E.1. *If s is an AF-answer, then $s \xrightarrow{need,llet-in,*} s'$ where s' is a L_{need}-WHNF.*

Lemma E.2. *If $s \xrightarrow{AF} t$, then $s \sim_{need} t$*

Proof. This follows from Theorem D.4, since all AF-reductions are also correct program transformations in L_{need} which is shown by the following table:

AF-reduction	*need*-transformation (usual non-standard)
β_{need}	(lbeta)
(lift)	(lapp)
(deref)	(cp-in)
(deref_{env})	(cp-e)
(assoc)	(llet-e)
(assoc_{env})	(llet-e)

Proposition E.3. $s \downarrow_{AF} \implies s \downarrow_{need}$

Proof. Let $s_0 \downarrow_{AF}$, i.e. $s_0 \xrightarrow{AF} s_1 \xrightarrow{AF} \ldots \xrightarrow{AF} s_n$, where s_n is an AF-answer. We use induction on n. If $n = 0$, then Lemma E.1 shows the claim. For the induction step we assume as induction hypothesis, that $s_1 \downarrow_{need}$ holds. Since every AF-reduction preserves contextual equivalence in L_{need}, we have $s_0 \sim_{need} s_1$ and thus $s_0 \downarrow_{need}$ must hold.

Lemma E.4. *Let $s \xrightarrow{need} t$ and $t \downarrow_{AF}$. Then $s \downarrow_{AF}$ holds.*

Proof. Let $s \xrightarrow{need} t \xrightarrow{AF,n} v$ where v is an AF-answer and $n \geq 0$. By induction on n we show that $s \downarrow_{AF}$. For the base case t is AF-answer. By inspecting all L_{need}-reduction one can verify, that either s must be also be an AF-answer, or $s \xrightarrow{AF} t$. Thus we have $s \downarrow_{AF}$. For the induction step let $t \xrightarrow{AF} t' \xrightarrow{AF,n-1} v$.

We need to consider overlappings of AF- and normal order reductions of the following form:

$$\begin{array}{ccc} s & \xrightarrow{need} & t \\ & & \downarrow_{AF} \\ & & t' \end{array}$$

If the reduction $s \xrightarrow{need} t$ is also an AF-reduction then we are finished since then obviously $s \downarrow_{AF}$. Hence we only treat the other cases where we use as induction hypothesis that for all r with $r \xrightarrow{need} t'$ we have $r \downarrow_{AF}$.

- $s \xrightarrow{need,lbeta} t$. Then $s \xrightarrow{\beta_{need}} t$ and thus the case trivially holds.
- $s \xrightarrow{need,lapp} t$. Suppose that this reduction is not an AF-reduction. Then $s = A[((\texttt{letrec}\ Env\ \texttt{in}\ r_1)\ r_2)]$ and $t = A[\texttt{letrec}\ Env\ \texttt{in}\ (r_1\ r_2)]$. An AF-reduction may modify the term t inside Env, inside r_1 or for a copy-operation the `letrec`-expression with the target in Env or r_1. Nevertheless the same reduction is applicable to s, i.e. the terms can be joined as follows:

$$\begin{array}{ccc} s & \xrightarrow{need,lapp} & t \\ {\scriptstyle AF}\downarrow & & \downarrow{\scriptstyle AF} \\ s' & \underset{need,lapp}{- - \succ} & t' \end{array}$$

  The induction hypothesis shows $s' \downarrow_{AF}$ and thus $s \downarrow_{AF}$.
- $s \xrightarrow{need,llet-in} t$ or $s \xrightarrow{need,llet-e} t$. Suppose that this is not an AF-reduction. Then again the AF-reduction for t can also be performed for s such that there results are related by a normal order llet-reduction. Note that it may happen that a $deref$ or a $assoc$ reduction for t is a $deref_{env}$ or a $assoc_{env}$ reduction for s.

$$\begin{array}{ccc} s & \xrightarrow{need,llet} & t \\ {\scriptstyle AF}\downarrow & & \downarrow{\scriptstyle AF} \\ s' & \underset{need,llet}{- - \succ} & t' \end{array}$$

  Again the induction hypothesis shows $s' \downarrow_{AF}$ and thus $s \downarrow_{AF}$.
- $s \xrightarrow{need,cp} t$. If an abstraction is copied, then $s \xrightarrow{AF} t$ by a $(deref)$ or $(deref_{env})$ reduction. Thus this case is straightforward. If a variable is copied, then this reduction cannot be an AF-reduction. Since the AF strategy follows variable-to-variable chains, every AF-reduction performed for t can also be performed for s. In this case both terms result in the same expression. For instance, let $s = \texttt{letrec}\ x = \lambda z.r, y = x\ \texttt{in}\ y$. Then $s \xrightarrow{need,cp} t = \texttt{letrec}\ x = \lambda z.r, y = x\ \texttt{in}\ x$, $t \xrightarrow{AF} t' = \texttt{letrec}\ x = \lambda z.r, y = x\ \texttt{in}\ \lambda z.r$, and $s \xrightarrow{AF} t'$. In summary this gives two situations:

$$\begin{array}{ccc} s & \xrightarrow{need,cp} & t \\ {\scriptstyle AF}\downarrow & & \downarrow{\scriptstyle AF} \\ s' & \underset{need,cp}{- - \succ} & t' \end{array} \qquad \begin{array}{ccc} s & \xrightarrow{need,cp} & t \\ & {\scriptstyle AF}\searrow & \downarrow{\scriptstyle AF} \\ & & t' \end{array}$$

For the first diagram we apply the induction hypothesis to s' and then derive $s \downarrow_{AF}$. The second case is obvious.

Proposition E.5. $s \downarrow_{need} \Longrightarrow s \downarrow_{AF}$.

Proof. Let $s \xrightarrow{need,n} v$ where v is a WHNF and $n \geq 0$. We use induction on n to show $s \downarrow_{AF}$. For the base case s is an L_{need}-WHNF. Then s is an AF-answer, too, i.e. $s \downarrow_{AF}$. For the induction step let $s \xrightarrow{need} t \xrightarrow{need,n-1} v$. As induction hypothesis we use $t \downarrow_{AF}$, i.e. there exists a reduction sequence $t \xrightarrow{AF,m} w$, where w is an AF answer and $m \geq 0$. I.e. this can be depicted as follows:

$$\begin{array}{ccccc} s & \xrightarrow{need} & t & \xrightarrow{need,n-1} & v \\ & & \downarrow{\scriptstyle AF,m} & & \\ & & w & & \end{array}$$

Now we can apply Lemma E.4 to $s \xrightarrow{need} t \xrightarrow{AF,m} w$ and have $s \downarrow_{AF}$.

Theorem E.6. $\leq_{need} = \leq_{AF}$ *and* $\sim_{need} = \sim_{AF}$.

Proof. This follows since L_{need}-convergence and AF-convergence are equivalent predicates.