Simulation in the Call-by-Need Lambda-Calculus with Letrec, Case, Constructors, and Seq

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-49

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

August 22, 2013[3]

Abstract. This paper shows equivalence of several versions of applicative similarity and contextual approximation, and hence also of applicative bisimilarity and contextual equivalence, in LR, the deterministic call-by-need lambda calculus with letrec extended by data constructors, case-expressions and Haskell's seq-operator. LR models an untyped version of the core language of Haskell. The use of bisimilarities simplifies equivalence proofs in calculi and opens a way for more convenient correctness proofs for program transformations.
The proof is by a fully abstract and surjective transfer into a call-by-name calculus, which is an extension of Abramsky's lazy lambda calculus. In the latter calculus equivalence of our similarities and contextual approximation can be shown by Howe's method. Using an inductive variant of similarity we then transfer similarity back to the calculus LR.
The translation from the call-by-need letrec calculus into the extended call-by-name lambda calculus is the composition of two translations. The first translation replaces the call-by-need strategy by a call-by-name strategy and its correctness is shown by exploiting infinite trees which emerge by unfolding the letrec expressions. The second translation encodes letrec-expressions by using multi-fixpoint combinators and its correctness is shown syntactically by comparing reductions of both calculi.
A further result of this paper is an isomorphism between the mentioned calculi, which is also an identity on letrec-free expressions.

1 Introduction

Motivation

Non-strict functional programming languages, such as the core-language of Haskell [Pey03], can be modeled using extended call-by-need lambda calculi.

The operational semantics of such a programming language defines how programs are evaluated and how the value of a program is obtained. Based on the operational semantics, the notion of *contextual equivalence* (see *e.g.* [Mor68,Plo75]) is a natural notion of program equivalence which follows Leibniz's law to identify the indiscernibles, that is two programs are equal iff their observable (termination) behavior is indistinguishable even if the programs are used as a subprogram of any other program (i.e. if the programs are plugged into

[3] This document is a revised version of an earlier version originally published on the web in July 2012.

any arbitrary context). For pure functional programs it suffices to observe whether or not the evaluation of a program terminates with a value (i.e. whether the program *converges*). Contextual equivalence has several advantages: Any reasonable notion of program equivalence should be a congruence which distinguishes obvious different values, *e.g.* different constants are distinguished, and functions (abstractions) are distinguished from constants. Contextual equivalence satisfies these requirements and is usually the coarsest of such congruences. Another (general) advantage is that once expressions, contexts, an evaluation, and a set of values are defined in a calculus, its definition of contextual equivalence can be derived, and thus this approach can be used for a broad class of program calculi.

On the other hand, due to the quantification over all program contexts, verifying equivalence of two programs *w.r.t.* contextual equivalence is often a difficult task. Nevertheless such proofs are required to ensure the *correctness of program transformations* where the correctness notion means that contextual equivalence is preserved by the transformation. Correctness of program transformations is indispensable for the correctness of compilers, but program transformations also play an important role in several other fields, *e.g.* in code refactoring to improve the design of programs, or in software verification to simplify expressions and thus to provide proofs or tests.

Bisimulation is another notion of program equivalence which was first invented in the field of process calculi (*e.g.* [Mil80,Mil99,SW01]), but has also been applied to functional programming and several extended lambda calculi (*e.g.* [How89,Abr90,How96]). Finding adequate notions of bisimilarity is still an active research topic (see *e.g.* [KW06,SKS11]). Briefly explained, bisimilarity equates two programs s_1, s_2 if all experiments passed for s_1 are also passed by s_2 and vice versa. For applicative similarity (and also bisimilarity) the experiments are evaluation and then recursively testing the obtained values: Abstractions are applied to all possible arguments, data objects are decomposed and the components are tested recursively. Applicative similarity is usually defined co-inductively, *i.e.* as a greatest fixpoint of an operator. Applicative similarity allows convenient and automatable proofs of correctness of program transformations, *e.g.* in mechanizing proofs [DBG97].

Abramsky and Ong showed that applicative bisimilarity is the same as contextual equivalence in a specific simple lazy lambda calculus [Abr90,AO93], and Howe [How89,How96] proved that in classes of lambda-calculi applicative bisimulation is the same as contextual equivalence. This leads to the expectation that some form of applicative bisimilarity may be used for calculi with Haskell's cyclic letrec. However, Howe's proof technique appears to be not adaptable to lambda calculi with cyclic let, since there are several deviations from the requirements for the applicability of Howe's framework. (i) Howe's technique is for call-by-name calculi and it is not obvious how to adapt it to call-by-need evaluation. (ii) Howe's technique requires that the values (results of reduction) are recognizable by their top operator. This does not apply to calculi with `letrec`, since `letrec`-expressions may be values as well as non-values. (iii) Call-by-need calculi with letrec usually require reduction rules to shift and join `letrec`-bindings. These modifications of the syntactic structure of expressions do not fit well into the proof structure of Howe's method.

Nevertheless, Howe's method is also applicable to calculi with non-recursive let even in the presence of nondeterminism [MSS10], where for the nondeterministic case applicative bisimilarity is only sound (but not complete) *w.r.t.* contextual equivalence. However, in the case of (cyclic) letrec and nondeterminism applicative bisimilarity is even unsound *w.r.t.* contextual equivalence [SSSM11]. This raises a question: which call-by-need calculi with letrec permit applicative bisimilarity as a tool for proving contextual equality?

Our Contribution

In [SSSM10] we have already shown 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 bisimilarity indeed holds. However, the full (untyped) core language of Haskell has data constructors, case-expressions and the seq-operator for strict evaluation. Moreover, in [SSMS13] it is shown that the extension of Abramsky's lazy lambda calculus with `case`, constructors, and `seq` is not conservative, i.e. it does not preserve contextual equivalence of expressions. Thus our results obtained in [SSSM10] for the lazy lambda calculus extended by letrec only are not transferable to the language extended by `case`, constructors, and `seq`. For this reason we provide a new proof for the untyped core language of Haskell.

As a model of Haskell's core language we use the call-by-need lambda calculus L_{LR} which was introduced and motivated in [SSSS08]. The calculus L_{LR} extends the lazy lambda calculus with letrec-expressions, data constructors, `case`-expressions for deconstructing the data, and Haskell's `seq`-operator for strict evaluation.

We define the operational semantics of L_{LR} in terms of a small-step reduction, which we call normal order reduction. As it is usual for lazy functional programming languages, evaluation of L_{LR}-expressions successfully halts if a *weak head normal form* is obtained, *i.e.* normal order reduction does not reduce inside the body of abstractions nor inside the arguments of constructor applications.

Our main result in this paper is that several variants of applicative bisimilarities are sound and complete for contextual equivalence in L_{LR}, i.e. coincide with contextual equivalence. Like context lemmas, an applicative bisimilarity can be used as a proof tool for showing contextual equivalence of expressions and for proving correctness of program transformations in the calculus L_{LR}. Since we have completeness of our applicative bisimilarities in addition to soundness, our results can also be used to disprove contextual equivalence of expressions in L_{LR}. Additionally, our result shows that the untyped applicative bisimilarity is sound for a polymorphic variant of L_{LR}, and hence for the typec core language of Haskell.

Having the proof tool of applicative bisimilarity in L_{LR} is also very helpful for more complex calculi if their pure core can be conservatively embedded in the full calculus. An example is our work on Concurrent Haskell [SSS11,SSS12], where our calculus CHF that models Concurrent Haskell has top-level processes with embedded lazy functional evaluation. We have shown in the calculus CHF that Haskell's deterministic core language can be conservatively embedded in the calculus CHF.

The L_{LR} calculus is well studied, and correctness of several important program transformations has been established for it in [SSSS08]. Thus, L_{LR} is an ideal starting point for reasoning and discussion in this paper.

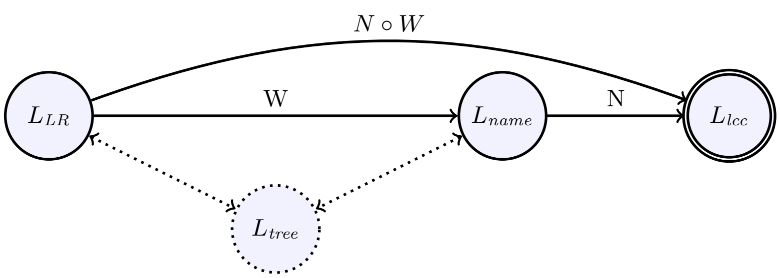


Fig. 1. Overall structure. Solid lines are fully-abstract translations, which are also isomorphisms and identities on letrec-free expressions; dotted lines are convergence preservation to/from the system L_{tree} of infinite trees.

The calculus L_{LR} is rather complex to reason in it directly. Thus we introduce L_{lcc} which can be seen as the calculus L_{LR} without `letrec`-expressions and following a fully-substituting call-by-name reduction instead of a call-by-need reduction. Another view of L_{lcc} is that it minimally extends Abramsky's lazy lambda calculus by Haskell's primitives: data constructors, `case`-expressions, and the `seq`-operator. As shown in [SSMS13], these constructs need to be added explicitly.

In addition to L_{LR} we introduce and describe two of its variants: a call-by-name variant L_{name} and a letrec-free call-by-name variant L_{lcc}. A consequence of our result is that the three calculi L_{LR}, L_{name} and L_{lcc} are isomorphic, modulo the equivalence (see Corollaries 6.17 and 5.33), and also that the embedding of the calculus L_{lcc} into the call-by-need calculus L_{LR} is an isomorphism of the respective term models.

In order to prove soundness and completeness of our applicative similarities for contextual equivalence in L_{LR}, we first prove this for the L_{lcc}, where Howe's technique can be applied. Then we show that for L_{LR} there exists a fully abstract translation to L_{lcc}. This is done via a fully abstract translation $W : L_{LR} \to L_{name}$, i.e. only the evaluation strategy is changed from call-by-need to call-by-name, but the syntax is unchanged. Full-abstractness of W is obtained by showing that the same expressions in the calculi in question correspond to the same infinite unfoldings of expressions with `letrec`, and that the corresponding evaluation (as infinitary rewriting, see [KKSdV97,SS07]) of infinite expressions preserves this equivalence. Then we further translate L_{name} to L_{lcc} by a translation $N : L_{name} \to L_{lcc}$, again showing the full abstractness of the translation. Since there is a fully-abstract translation between any pair of L_{LR}, L_{name}, and L_{lcc}, soundness and completeness of our bisimilarities w.r.t. contextual equivalence can be lifted from L_{lcc} to the other calculi.

The main structure of our reasoning is depicted in Figure 1. We establish full abstractness of translation between calculi $L_{LR}, L_{name}, L_{lcc}$ with respect to contextual equivalence. As an intermediate step, L_{LR} and

L_{name} are shown to be equivalent to the calculus L_{IT} of infinite trees with respect to convergence (dotted lines in the picture).

Related Work

In [Gor99] Gordon shows that bisimilarity and contextual equivalence coincide in an extended call-by-name PCF language. Gordon provides a bisimilarity in terms of a labeled transition system. A similar result is obtained in [Pit97] for PCF extended by product types and lazy lists where the proof uses Howe's method ([How89,How96]; see also [MSS10,Pit11]), and where the operational semantics is a big-step one for an extended PCF-language. Nevertheless, the observation of convergence in the definition of contextual equivalence is restricted to programs (and contexts) of ground type (*i.e.* of type integer or `Bool`). Therefore Ω and $\lambda x.\Omega$ are equal in the calculi considered by Gordon and Pitts. This does not hold in our setting for two reasons: first, we observe termination for functions and thus the empty context already distinguishes Ω and $\lambda x.\Omega$, and second, our languages employ Haskell's seq-operator which permits to test convergence of any expression and thus the context `seq` $[\cdot]$ `True` distinguishes Ω and $\lambda x.\Omega$.

[Jef94] presents an investigation into the semantics of a lambda-calculus that permits cyclic graphs, where a fully abstract denotational semantics is described. However, the calculus is different from our calculi in its expressiveness since it permits a parallel convergence test, which 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 7.7 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$^+$95,AF97,MOW98] present call-by-need lambda calculi with non-recursive `let` and a let-less formulation of call-by-need reduction. For a calculus with non-recursive `let` it is shown in [MOW98] that call-by-name and call-by-need evaluation induce the same observational equivalences. Additionally, the extension of the corresponding calculi by recursive `let` is discussed in [AFM$^+$95,AF97], and further call-by-need lambda calculi with a recursive `let` are presented in [AB97,AB02,NH09]. In [AB02] it is shown that there exist infinite normal forms and that the calculus satisfies a form of confluence. All these calculi correspond to our calculus L_{LR}, but none of them combine recursive `let` with data constructors, case-expressions and `seq`.

In [MS99] a call-by-need calculus is analyzed which is closer to our calculus L_{LR}, since `letrec`, `case`, and constructors are present (but not `seq`). Another difference is that [MS99] use an abstract machine semantics as operational semantics, while their approach to program equivalence is based on contextual equivalence, as is ours.

The operational semantics of call-by-need lambda calculi with `letrec` are investigated in [Lau93] and [Ses97], where the former analyzes the big-step semantics, and the latter investigates the construction of efficient abstract machines for those calculi.

Investigations of the semantics of lazy functional programming languages including the `seq`-operator can be found in [JV06,VJ07].

Outline

In Sect. 2 we introduce some common notions of program calculi, contextual equivalence, similarity and also on translations between those calculi. In Sect. 3 we introduce the extension L_{lcc} of Abramsky's lazy lambda calculus with `case`, constructors, and `seq`, and two letrec-calculi L_{LR}, L_{name} as further syntactic extensions. In Sect. 4 we show that for so-called "convergence admissible" calculi an alternative inductive characterization of similarity is possible. We then use Howe's method in L_{lcc} to show that contextual approximation and a standard version of applicative similarity coincide. Proving that L_{lcc} is convergence admissible then implies that the alternative inductive characterization of similarity can be used for L_{lcc}. In Sect. 5 and 6 the translations W and N are introduced and the full-abstraction results are obtained. In Sect. 7 we show soundness and completeness of our variants of applicative similarity w.r.t. contextual equivalence in L_{LR}. 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 D *is a four-tuple* $(\mathbb{E}, \mathbb{C}, \rightarrow, \mathbb{A})$, *where* $\mathbb{E}$ *are expressions (up to* α*-equivalence),* $\mathbb{C} : \mathbb{E} \rightarrow \mathbb{E}$ *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 (*i.e.*, deterministic), and* $\mathbb{A} \subset \mathbb{E}$ *is a set of* answers *of the calculus.*

For $C \in \mathbb{C}$ *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* $\mathbb{C}$ *with* $Id[s] = s$ *for all expressions* s*, and that* $\mathbb{C}$ *is closed under composition, i.e.* $C_1, C_2 \in \mathbb{C} \implies C_1 \circ C_2 \in \mathbb{C}$. *The* transitive closure *of* $\rightarrow$ *is denoted as* $\xrightarrow{+}$ *and the* transitive and reflexive closure *of* $\rightarrow$ *is denoted as* $\xrightarrow{*}$. *The notation* $\xrightarrow{0\vee 1}$ *means equality or one reduction, and* $\xrightarrow{k}$ *means* k *reductions. Given an expression* s*, a sequence* $s \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$ *is called a* reduction sequence*; it is called an* evaluation *if* s_n *is an answer,* i.e. $s_n \in \mathbb{A}$*; in this case we say* s converges *and denote this as* $s{\downarrow_D}s_n$ *or as* $s{\downarrow_D}$ *if* s_n *is not important. If there is no* s_n s.t. $s{\downarrow_D}s_n$ *then* s diverges*, denoted as* $s{\Uparrow_D}$*. When dealing with multiple calculi, we often use the calculus name to mark its expressions and relations,* e.g. $\xrightarrow{D}$ *denotes a reduction relation in* D*.*

We will have to deal with several calculi and preorders. Throughout this paper we will use the symbol $\preccurlyeq$ for co-inductively defined preorders (i.e. similarities), and $\leq$ for (inductively defined or otherwise defined) contextual preorders. For the corresponding symmetrizations we use $\simeq$ for $\preccurlyeq \cap \succcurlyeq$ and $\sim$ for $\leq \cap \geq$. All the symbols are always indexed by the corresponding calculus and sometimes more restrictions like specific sets of contexts are attached to the indices of the symbols.

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

Definition 2.2 (Contextual Approximation and Equivalence, $\leq_D$ and $\sim_D$). *Let* $D = (\mathbb{E}, \mathbb{C}, \rightarrow, \mathbb{A})$ *be a calculus and* s_1, s_2 *be* D*-expressions.* Contextual approximation $\leq_D$ *and* contextual equivalence $\sim_D$ *are defined as:*

$$s_1 \leq_D s_2 \text{ iff } \forall C \in \mathbb{C}: \quad C[s_1]{\downarrow_D} \Rightarrow C[s_2]{\downarrow_D}$$
$$s_1 \sim_D s_2 \text{ iff } s_1 \leq_D s_2 \wedge s_2 \leq_D s_1$$

A program transformation *is a binary relation* $\eta \subseteq (\mathbb{E} \times \mathbb{E})$*. A program transformation* η *is called* correct *iff* $\eta \subseteq \sim_D$*.*

Note that $\leq_D$ is a precongruence, *i.e.*, $\leq_D$ is reflexive, transitive, and $s \leq_D t$ implies $C[s] \leq_D C[t]$ for all $C \in \mathbb{C}$, and that $\sim_D$ is a congruence, *i.e.* a precongruence and an equivalence relation.

We also define a general notion of similarity for untyped deterministic calculi which is defined co-inductively. We first define the operator $F_{D,\mathcal{Q}}$ on binary relations of expressions:

Definition 2.3. *Let* $D = (\mathbb{E}, \mathbb{C}, \rightarrow, \mathbb{A})$ *be an untyped deterministic calculus and let* $\mathcal{Q} \subseteq \mathbb{C}$ *be a set of functions on expressions (*i.e. $\forall Q \in \mathcal{Q} : Q : \mathbb{E} \rightarrow \mathbb{E}$*). Then the* $\mathcal{Q}$-experiment operator $F_{D,\mathcal{Q}} : 2^{(\mathbb{E}\times\mathbb{E})} \rightarrow 2^{(\mathbb{E}\times\mathbb{E})}$ *is defined as follows for* $\eta \subseteq \mathbb{E} \times \mathbb{E}$*:*

$$s_1\, F_{D,\mathcal{Q}}(\eta)\, s_2 \text{ iff } s_1{\downarrow_D}v_1 \implies \exists v_2. (s_2{\downarrow}v_2 \wedge \forall Q \in \mathcal{Q} : Q(v_1)\, \eta\, Q(v_2))$$

Lemma 2.4. *The operator* $F_{D,\mathcal{Q}}$ *is monotonous* w.r.t. *set inclusion,* i.e. *for all binary relations* η_1, η_2 *on expressions* $\eta_1 \subseteq \eta_2 \implies F_{D,\mathcal{Q}}(\eta_1) \subseteq F_{D,\mathcal{Q}}(\eta_2)$*.*

Proof. Let $\eta_1 \subseteq \eta_2$ and $s_1\, F_{D,\mathcal{Q}}(\eta_1)\, s_2$. The latter assumption implies $s_1{\downarrow_D}v_1 \implies (s_2{\downarrow_D}v_2 \wedge \forall Q \in \mathcal{Q} : Q(v_1)\, \eta_1\, Q(v_2))$. From $\eta_1 \subseteq \eta_2$ we have $s_1{\downarrow}v_1 \implies (s_2{\downarrow_D}v_2 \wedge \forall Q \in \mathcal{Q} : Q(v_1)\, \eta_2\, Q(v_2))$. Thus, $s_1\, F_{\mathcal{Q}}(\eta_2)\, s_2$.

Since $F_{D,\mathcal{Q}}$ is monotonous, its greatest fixpoint exists:

Definition 2.5 ($\mathcal{Q}$-Similarity, $\preccurlyeq_{D,\mathcal{Q}}$). *The behavioral preorder* $\preccurlyeq_{D,\mathcal{Q}}$*, called* $\mathcal{Q}$-similarity*, is defined as the greatest fixed point of* $F_{D,\mathcal{Q}}$*.*

We also provide an inductive definition of behavioral equivalence, which is defined as a contextual preorder where the contexts are restricted to the set $\mathcal{Q}$ (and the empty context).

Definition 2.6. *Let $D = (\mathbb{E}, \mathbb{C}, \rightarrow, \mathbb{A})$ be an untyped deterministic calculus, and $\mathcal{Q} \subseteq \mathbb{C}$. Then the relation $\leq_{D,\mathcal{Q}}$ is defined as follows:*

$$s_1 \leq_{D,\mathcal{Q}} s_2 \text{ iff } \forall n \geq 0 : \forall Q_i \in \mathcal{Q} : Q_1(Q_2(\ldots(Q_n(s_1))))\downarrow_D \implies Q_1(Q_2(\ldots(Q_n(s_2))))\downarrow_D$$

Later in Section 4.1 we will provide a sufficient criterion on untyped deterministic calculi that ensures that $\preccurlyeq_{D,\mathcal{Q}}$ and $\leq_{D,\mathcal{Q}}$ coincide.

We are interested in translations between calculi that are faithful *w.r.t.* the corresponding contextual preorders.

Definition 2.7. *[[SSNSS08,SSNSS09]] A* translation $\tau : (\mathbb{E}_1, \mathbb{C}_1, \rightarrow_1, \mathbb{A}_1) \rightarrow (\mathbb{E}_2, \mathbb{C}_2, \rightarrow_2, \mathbb{A}_2)$ *is a mapping $\tau_E : \mathbb{E}_1 \rightarrow \mathbb{E}_2$ and a mapping $\tau_C : \mathbb{C}_1 \rightarrow \mathbb{C}_2$ such that $\tau_C(Id_1) = Id_2$. The following properties of translations are defined:*

- τ *is* compositional *iff $\tau(C[s]) = \tau(C)[\tau(s)]$ for all C, s.*
- τ *is* convergence equivalent *iff $s\downarrow_1 \iff \tau(s)\downarrow_2$ for all s.*
- τ *is* adequate *iff for all $s, t \in \mathbb{E}_1$: $\tau(s) \leq_2 \tau(t) \implies s \leq_1 t$.*
- τ *is* fully abstract *iff for all $s, t \in \mathbb{E}_1$: $s \leq_1 t \iff \tau(s) \leq_2 \tau(t)$.*
- τ *is an* isomorphism *iff it is fully abstract and a bijection on the quotients* $\tau/\!\sim\, : \mathbb{E}_1/\!\sim\, \rightarrow \mathbb{E}_2/\!\sim$.

Note that isomorphism means an order-isomorphism between the term-models, where the orders are $\leq_1 /\!\sim$ and $\leq_2 /\!\sim$ (which are the relations in the quotient).

Proposition 2.8 ([SSNSS08,SSNSS09]). *If a translation $\tau : (\mathbb{E}_1, \mathbb{C}_1, \rightarrow_1, \mathbb{A}_1) \rightarrow (\mathbb{E}_2, \mathbb{C}_2, \rightarrow_2, \mathbb{A}_2)$ is compositional and convergence equivalent, then it is also adequate.*

Proof. Let $s, t \in \mathbb{E}_1$ with $\tau(s) \leq_2 \tau(t)$ and let $C[s]\downarrow_1$ for some $C \in \mathbb{C}$. It is sufficient to show that this implies $C[t]\downarrow_1$: Convergence equivalence shows that $\tau(C[s])\downarrow_2$. Compositionality implies $\tau(C)[\tau(s)])\downarrow_2$, and then $\tau(s) \leq_2 \tau(t)$ implies $\tau(C)[\tau(t)]\downarrow_2$. Compositionality applied once more implies $\tau(C[t])\downarrow_2$, and then convergence equivalence finally implies $C[t]\downarrow_1$.

3 Three Calculi

In this section we introduce the calculi L_{LR}, L_{name}, and L_{lcc}. L_{LR} is a call-by-need calculus with recursive `let`, data constructors, `case`-expressions, and the `seq`-operator. The calculus L_{name} has the same syntactic constructs as L_{LR}, but uses a call-by-name, rather than a call-by-need, evaluation. The calculus L_{lcc} does not have `letrec`, and also uses a call-by-name evaluation.

For all three calculi we assume that there is a (common) set of *data constructors* c which is partitioned into *types*, such that every constructor c belongs to exactly one type. We assume that for every type T the set of its corresponding data constructors can be enumerated as $c_{T,1}, \ldots, c_{T,|T|}$ where $|T|$ is the number of data constructors of type T. We also assume that every constructor has a fixed arity denoted as $\mathrm{ar}(c)$ which is a non-negative integer. We assume that there is a type *Bool* among the types, with the data constructors `False` and `True` both of arity 0. We require that data constructors occur only fully saturated, *i.e.* a constructor c is only allowed to occur together with $\mathrm{ar}(c)$ arguments, written as $(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})$ where s_i are expressions of the corresponding calculus. We also write $(c\ \overrightarrow{s})$ as an abbreviation for the constructor application $(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})$. All three calculi allow deconstruction via `case`-expressions which are constructed as

$$\texttt{case}_T\ s\ \texttt{of}\ (c_{T,1}\ x_{1,1}\ \ldots\ x_{1,\mathrm{ar}(c_{T,1})} \rightarrow s_1) \ldots (c_{T,|T|}\ x_{|T|,1}\ \ldots\ x_{|T|,\mathrm{ar}(c_{T,|T|})} \rightarrow s_{|T|})$$

where s, s_i are expressions and $x_{i,j}$ are variables of the corresponding calculus. Thus there is a $\texttt{case}_T$-construct for every type T and we require that there is exactly one case-alternative $(c_{T,i}\ x_{i,1}\ \ldots\ x_{i,\mathrm{ar}(c_{T,i})} \rightarrow s_i)$ for every constructor $c_{T,i}$ of type T. In a case-alternative $(c_{T,i}\ x_{i,1}\ \ldots\ x_{i,\mathrm{ar}(c_{T,i})} \rightarrow s_i)$ we call $c_{T,i}\ x_{i,1}\ \ldots\ x_{i,\mathrm{ar}(c_{T,i})}$ a *pattern* and s_i the right hand side of the alternative. All variables in a `case`-pattern must be pairwise distinct. We will sometimes abbreviate the case-alternatives by *alts* if the exact terms of the alternatives are not of interest. As a further abbreviation we sometimes write $\texttt{if}\ s_1\ \texttt{then}\ s_2\ \texttt{else}\ s_3$ for the case-expression $(\texttt{case}_{Bool}\ s_1\ \texttt{of}\ (\texttt{True} \rightarrow s_2)\ (\texttt{False} \rightarrow s_3))$.

We now define the syntax of expressions with `letrec`, *i.e.* the set $\mathbb{E}_{\mathcal{L}}$ of expressions which are used in both of the calculi L_{LR} and L_{name}.

Definition 3.1 (Expressions $\mathbb{E}_{\mathcal{L}}$). *The set $\mathbb{E}_{\mathcal{L}}$ of expressions is defined by the following grammar, where x, x_i are variables:*

$$r, s, t, r_i, s_i, t_i \in \mathbb{E}_{\mathcal{L}} ::= x \mid (s\ t) \mid (\lambda x.s) \mid (\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t)$$
$$\mid (c\ s_1 \ldots s_{\mathrm{ar}(c)}) \mid (\texttt{seq}\ s\ t) \mid (\texttt{case}_T\ s\ \texttt{of}\ alts)$$

We assign the names application*,* abstraction*,* `seq`*-expression, or* `letrec`-expression *to the expressions $(s\ t)$, $(\lambda x.s)$, $(\texttt{seq}\ s\ t)$, or $(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t)$, respectively. A* value *v is defined as an abstraction or a constructor application. A group of* `letrec` *bindings is sometimes abbreviated as Env. We use the notation $\{x_{g(i)} = s_{h(i)}\}_{i=m}^{n}$ for the chain $x_{g(m)} = s_{h(m)}, x_{g(m+1)} = s_{h(m+1)}, \ldots, x_{g(n)} = s_{h(n)}$ of bindings where $g, h : \mathbb{N} \rightarrow \mathbb{N}$,* e.g.*, $\{x_i = s_{i-1}\}_{i=m}^{n}$ means the bindings $x_m = s_{m-1}, x_{m+1} = s_m, \ldots x_n = s_{n-1}$. We assume that 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 $1 \leq i \leq n$.*

$\mathbb{C}_{\mathcal{L}}$ denotes the set of all contexts for the expressions $\mathbb{E}_{\mathcal{L}}$.

Free and bound variables in expressions and α-renamings are defined as usual. The set of free variables in s is denoted as $FV(s)$.

Convention 3.2 (Distinct Variable Convention) *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. All reduction rules are assumed to implicitly α-rename bound variables in the result if necessary.*

In all three calculi we will use the symbol Ω for the specific (`letrec`-free) expression $(\lambda z.(z\ z))\ (\lambda x.(x\ x))$. In all of our calculi Ω is divergent and the least element of the corresponding contextual preorder. This is proven in [SSSS08] for L_{LR} and can easily be proven for the other two calculi using standard methods, such as context lemmas. Note that this property also follows from the Main Theorem 7.6 for all three calculi.

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

We begin with the call-by-need lambda calculus L_{LR} which is exactly the call-by-need calculus of [SSSS08]. It has a rather complex form of reduction rules using variable chains. The justification is that this formulation permits direct syntactic proofs of correctness *w.r.t.* contextual equivalence for a large class of transformations. Several modifications of the reduction strategy, e.g. removing indirections, do not change the semantics of the calculus, however, they appear to be not treatable by syntactic proof methods using diagrams (see [SSSS08]). L_{LR}-expressions are exactly the expressions $\mathbb{E}_{\mathcal{L}}$.

Definition 3.3. *The* reduction rules *for the calculus and language L_{LR} are defined in Fig. 2, 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: the union of (*llet-in*) and (*llet-e*) is called (*llet*). The union of (*llet*), (*lapp*), (*lcase*), and (*lseq*) is called (*lll*).*

For the definition of the normal order reduction strategy of the calculus L_{LR} we use the labeling algorithm in Fig. 3 which detects the position where a reduction rule is applied according to the normal order. It uses the following labels: S (subterm), T (top term), V (visited), and W (visited, but not target). 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 $l \rightsquigarrow r$ 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 takes an expression s as its input and exhaustively applies the rules in Fig. 3 to s^T, where no other subexpression in s is labeled. The label T is used to prevent the labeling algorithm from decending into `letrec`-environments that are not at the

(lbeta)	$C[((\lambda x.s)^S\ t)] \to C[\texttt{letrec}\ x = t\ \texttt{in}\ s]$
(cp-in)	$\texttt{letrec}\ x_1 = (\lambda x.s)^S, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ C[x_m^V]$ $\to \texttt{letrec}\ x_1 = (\lambda x.s), \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ C[(\lambda x.s)]$
(cp-e)	$\texttt{letrec}\ x_1 = (\lambda x.s)^S, \{x_i = x_{i-1}\}_{i=2}^m, Env, y = C[x_m^V]\ \texttt{in}\ t$ $\to \texttt{letrec}\ x_1 = (\lambda x.s), \{x_i = x_{i-1}\}_{i=2}^m, Env, y = C[(\lambda x.s)]\ \texttt{in}\ t$
(lapp)	$C[((\texttt{letrec}\ Env\ \texttt{in}\ s)^S\ t)] \to C[(\texttt{letrec}\ Env\ \texttt{in}\ (s\ t))]$
(lcase)	$C[(\texttt{case}_T\ (\texttt{letrec}\ Env\ \texttt{in}\ s)^S\ \texttt{of}\ alts)]$ $\to C[(\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{case}_T\ s\ \texttt{of}\ alts))]$
(lseq)	$C[(\texttt{seq}\ (\texttt{letrec}\ Env\ \texttt{in}\ s)^S\ t)] \to C[(\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{seq}\ s\ t))]$
(llet-in)	$\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ s)^S \to \texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ s$
(llet-e)	$\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s)^S\ \texttt{in}\ t \to \texttt{letrec}\ Env_1, Env_2, x = s\ \texttt{in}\ t$
(seq-c)	$C[(\texttt{seq}\ v^S\ s)] \to C[s]$ if v is a value
(seq-in)	$(\texttt{letrec}\ x_1 = v^S, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ C[(\texttt{seq}\ x_m^V\ s)])$ $\to (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ C[s])$ if v is a constructor application
(seq-e)	$(\texttt{letrec}\ x_1 = v^S, \{x_i = x_{i-1}\}_{i=2}^m, Env, y = C[(\texttt{seq}\ x_m^V\ s)]\ \texttt{in}\ t)$ $\to (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^m, Env, y = C[s]\ \texttt{in}\ t)$ if v is a constructor application
(case-c)	$C[(\texttt{case}_T\ (c_i\ \overrightarrow{s})^S \texttt{of} \ldots ((c_i\ \overrightarrow{y}) \to t_i) \ldots)] \to C[(\texttt{letrec}\ \{y_i = s_i\}_{i=1}^{\mathrm{ar}(c_i)}\ \texttt{in}\ t_i)]$ if $\mathrm{ar}(c_i) \geq 1$
(case-c)	$C[(\texttt{case}_T\ c_i^S \texttt{of} \ldots\ (c_i \to t_i) \ldots)] \to C[t_i]$ if $\mathrm{ar}(c_i) = 0$
(case-in)	$\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{s})^S, \{x_i = x_{i-1}\}_{i=2}^m, Env$ $\texttt{in}\ C[\texttt{case}_T\ x_m^V\ \texttt{of} \ldots ((c_i\ \overrightarrow{z}) \to t) \ldots]$ $\to \texttt{letrec}\ x_1 = (c_i\ \overrightarrow{y}), \{y_i = s_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=2}^m, Env$ $\texttt{in}\ C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^{\mathrm{ar}(c_i)}\ \texttt{in}\ t)]$ if $\mathrm{ar}(c_i) \geq 1$ and where y_i are fresh
(case-in)	$\texttt{letrec}\ x_1 = c_i^S, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ C[\texttt{case}_T\ x_m^V\ \ldots\ (c_i \to t) \ldots]$ $\to \texttt{letrec}\ x_1 = c_i, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ C[t]$ if $\mathrm{ar}(c_i) = 0$
(case-e)	$\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{s})^S, \{x_i = x_{i-1}\}_{i=2}^m,$ $u = C[\texttt{case}_T\ x_m^V\ \texttt{of} \ldots ((c_i\ \overrightarrow{z}) \to t) \ldots], Env$ $\texttt{in}\ r$ $\to \texttt{letrec}\ x_1 = (c_i\ \overrightarrow{y}), \{y_i = s_i\}_{i=1}^{\mathrm{ar}(c_i)}, \{x_i = x_{i-1}\}_{i=2}^m,$ $u = C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^{\mathrm{ar}(c_i)}\ \texttt{in}\ t)], Env$ $\texttt{in}\ r$ if $\mathrm{ar}(c_i) \geq 1$ and where y_i are fresh
(case-e)	$\texttt{letrec}\ x_1 = c_i^S, \{x_i = x_{i-1}\}_{i=2}^m, u = C[\texttt{case}_T\ x_m^V\ \ldots\ (c_i \to t) \ldots], Env\ \texttt{in}\ r$ $\to \texttt{letrec}\ x_1 = c_i, \{x_i = x_{i-1}\}_{i=2}^m \ldots, u = C[t], Env\ \texttt{in}\ r$ if $\mathrm{ar}(c_i) = 0$

Fig. 2. Reduction rules of L_{LR}

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

Fig. 3. Labeling algorithm for L_{LR}

top of the expression. The labels V and W mark the visited bindings of a chain of bindings, where W is used for variable-to-variable bindings. 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.

Definition 3.4 (Normal Order Reduction of L_{LR}). *Let s be an expression. Then a single normal order reduction step $\xrightarrow{LR}$ is defined as follows: first the labeling algorithm in Fig. 3 is applied to s. If the labeling algorithm terminates successfully, then one of the rules in Fig. 2 is applied, if possible, where the labels S, V must match the labels in the expression s (again s 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 $\xrightarrow{LR,a}$,* e.g. $\xrightarrow{LR,\text{lapp}}$ *applies the rule* (lapp)*.*

The normal order reduction of L_{LR} implements a call-by-need reduction with sharing which avoids substitution of arbitrary expressions. We describe the rules: The rule (lbeta) is a sharing variant of classical β-reduction, where the argument of an abstraction is shared by a new `letrec`-binding instead of substituting the argument in the body of an abstraction. The rules (cp-in) and (cp-e) allow to copy abstractions into needed positions. The rules (lapp), (lcase), and (lseq) allow moving `letrec`-expressions to the top of the term if they are inside a reduction position of an application, a `case`-expression, or a `seq`-expression. To flatten nested `letrec`-expressions, the rules (llet-in) and (llet-e) are added to the reduction. Evaluation of `seq`-expressions is performed by the rules (seq-c), (seq-in), and (seq-e), where the first argument of `seq` must be a value (rule seq-c) or it must be a variable which is bound in the outer `letrec`-environment to a constructor application. Since normal order reduction avoids copying constructor applications, the rules (seq-in) and (seq-e) are required. Correspondingly, the evaluation of `case`-expressions requires several variants: there are again three rules for the cases where the argument of `case` is already a constructor application (rule (case-c)) or where the argument is a variable which is bound to a constructor application (perhaps by several indirections in the `letrec`-environment) which are covered by the rule (case-in) and (case-e). All three rules have two variants: one variant for the case when a constant is scrutinized (and thus no arguments need to be shared by new `letrec`-bindings) and another variant for the case when arguments are present (and thus the arity of the scrutinized constructor is strictly greater than 0). For the latter case the arguments of the constructor application are shared by new `letrec`-bindings, such that the newly created variables can be used as references in the right hand side of the matching alternative.

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

Of course, reduction contexts could also be defined recursively, as in [SSSS08, Definition 1.5], but such a definition is very cumbersome due to a large number of special cases. The labeling algorithm provides a definition that, in our experience, is easier to work with.

By induction on the term structure one can easily verify that the normal order redex, as well as the normal order reduction, is unique. A *weak head normal form in L_{LR} (L_{LR}-WHNF)* is either an abstraction

(gc)	$C[\texttt{letrec}\ \{x_i = s_i\}_{i=1}^{n}\ \texttt{in}\ t] \to C[t]$, if $FV(t) \cap \{x_1,\ldots,x_n\} = \emptyset$
(gc)	$C[\texttt{letrec}\ \{x_i = s_i\}_{i=1}^{n}, \{y_i = t_i\}_{i=1}^{m}\ \texttt{in}\ t] \to C[\texttt{letrec}\ \{y_i = t_i\}_{i=1}^{m}\ \texttt{in}\ t]$, if $(FV(t) \cup \bigcup_{i=1}^{m} FV(t_i)) \cap \{x_1,\ldots,x_n\} = \emptyset$
(lwas)	$C[(s\ (\texttt{letrec}\ Env\ \texttt{in}\ t))] \to C[\texttt{letrec}\ Env\ \texttt{in}\ (s\ t)]$
(lwas)	$C[(c\ s_1\ \ldots (\texttt{letrec}\ Env\ \texttt{in}\ s_i) \ldots s_n)] \to C[\texttt{letrec}\ Env\ \texttt{in}\ (c\ s_1\ \ldots\ s_i\ \ldots s_n)]$
(lwas)	$C[(\texttt{seq}\ s\ (\texttt{letrec}\ Env\ \texttt{in}\ t))] \to C[\texttt{letrec}\ Env\ \texttt{in}\ \texttt{seq}\ s\ t]$

Fig. 4. Transformations for garbage collection and `letrec`-shifting

$(\texttt{letrec}\ Env\ \texttt{in}\ s)^X$	$\rightsquigarrow (\texttt{letrec}\ Env\ \texttt{in}\ s^X)$ if X is S or T
$(s\ t)^{S\vee T}$	$\rightsquigarrow (s^S\ t)$
$(\texttt{seq}\ s\ t)^{S\vee T}$	$\rightsquigarrow (\texttt{seq}\ s^S\ t)$
$(\texttt{case}_T\ s\ \texttt{of}\ alts)^{S\vee T}$	$\rightsquigarrow (\texttt{case}_T\ s^S\ \texttt{of}\ alts)$

Fig. 5. Labeling algorithm for L_{name}

$\lambda x.s$, a constructor application $(c\ s_1\ \ldots\ s_{\mathrm{ar}(c_i)})$, or an expression $(\texttt{letrec}\ Env\ \texttt{in}\ v)$ where v is a constructor application or an abstraction, or an expression of the form $(\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ x_m)$, where $v = (c\ \overrightarrow{s})$. We distinguish abstraction-WHNF (AWHNF) and constructor WHNF (CWHNF) based on whether the value v is an abstraction or a constructor application, respectively. The notions of convergence, divergence and contextual approximation are as defined in Sect. 2. If there is no normal order reduction originating at an expression s then $s\Uparrow_{LR}$. This, in particular, means that expressions for which the labeling algorithm fails to find a redex, or for which there is no matching constructor for a subexpression (that is a WHNF) in a `case` redex position, or expressions with cyclic dependencies like $\texttt{letrec}\ x = x\ \texttt{in}\ x$, are diverging.

Example 3.6. We consider the expression $s_1 := \texttt{letrec}\ x = (y\ \lambda u.u), y = \lambda z.z\ \texttt{in}\ x$. The labeling algorithm applied to s_1 yields $(\texttt{letrec}\ x = (y^V\ \lambda u.u)^V, y = (\lambda z.z)^S\ \texttt{in}\ x^V)^V$. The reduction rule that matches this labeling is the reduction rule (cp-e), *i.e.* $s_1 \xrightarrow{LR} (\texttt{letrec}\ x = ((\lambda z'.z')\ \lambda u.u), y = (\lambda z.z)\ \texttt{in}\ x) = s_2$. The labeling of s_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 rule (lbeta) applicable, *i.e.* $s_2 \xrightarrow{LR} (\texttt{letrec}\ x = (\texttt{letrec}\ z' = \lambda u.u\ \texttt{in}\ z'), y = (\lambda z.z)\ \texttt{in}\ x) = s_3$. The labeling of s_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 s_3, *i.e.* $s_3 \xrightarrow{LR} (\texttt{letrec}\ x = z', z' = \lambda u.u, y = (\lambda z.z)\ \texttt{in}\ x) = s_4$. Now s_4 gets labeled as $(\texttt{letrec}\ x = z'^W, z' = (\lambda u.u)^S, y = (\lambda z.z)\ \texttt{in}\ x^V)^V$, and a (cp-in)-reduction is applicable, *i.e.* $s_4 \xrightarrow{LR} (\texttt{letrec}\ x = z', z' = (\lambda u.u), y = (\lambda z.z)\ \texttt{in}\ (\lambda u.u)) = s_5$. The labeling algorithm applied to s_5 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 s_5, since s_5 is a WHNF.

Concluding, the calculus L_{LR} is defined by the tuple $(\mathbb{E}_{\mathcal{L}}, \mathbb{C}_{\mathcal{L}}, \xrightarrow{LR}, \mathbb{A}_{LR})$ where $\mathbb{A}_{LR}$ are the L_{LR}-WHNFs, where we equate alpha-equivalent expressions, contexts and answers.

In [SSSS08] correctness of several program transformations was shown:

Theorem 3.7 ([SSSS08, Theorems 2.4 and 2.9]). *All reduction rules shown in Fig. 2 are correct program transformations, even if they are used with an arbitrary context C in the rules without requiring the labels. The transformations for garbage collection (gc) and for shifting of* `letrec`*-expressions (lwas) shown in Fig. 4 are also correct program transformations.*

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

Now we define a call-by-name calculus on $\mathbb{E}_{\mathcal{L}}$-expressions. The calculus L_{name} has $\mathbb{E}_{\mathcal{L}}$ as expressions, but the reduction rules are different from L_{LR}. L_{name} does not implement a sharing strategy but instead performs the usual call-by-name beta-reduction and copies arbitrary expressions directly into needed positions.

(beta)	$C[(\lambda x.s)^S\ t] \to C[s[t/x]]$
(gcp)	$C_1[\texttt{letrec}\ Env,\ x = s\ \texttt{in}\ C_2[x^{S \vee T}]] \to C_1[\texttt{letrec}\ Env,\ x = s\ \texttt{in}\ C_2[s]]$
(lapp)	$C[((\texttt{letrec}\ Env\ \texttt{in}\ s)^S\ t)] \to C[(\texttt{letrec}\ Env\ \texttt{in}\ (s\ t))]$
(lcase)	$C[(\texttt{case}_T\ (\texttt{letrec}\ Env\ \texttt{in}\ s)^S\ \texttt{of}\ alts)]$ $\to C[(\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{case}_T\ s\ \texttt{of}\ alts))]$
(lseq)	$C[(\texttt{seq}\ (\texttt{letrec}\ Env\ \texttt{in}\ s)^S\ t)] \to C[(\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{seq}\ s\ t))]$
(seq-c)	$C[(\texttt{seq}\ v^S\ s)] \to C[s]$ if v is a value
(case)	$C[(\texttt{case}_T\ (c\ s_1 \dots s_{ar(c)})^S\ \texttt{of} \dots ((c\ x_1 \dots x_{ar(c)}) \to t) \dots)]$ $\to C[t[s_1/x_1, \dots, s_{ar(c)}/x_{ar(c)}]]$

Fig. 6. Normal order reduction rules $\xrightarrow{name}$ of L_{name}

In Fig. 5 the rules of the labeling algorithm for L_{name} are given. The algorithm uses the labels S and T. For an expression s the labeling starts with s^T.

An L_{name} reduction context R_{name} is any context where the hole is labeled T or S by the labeling algorithm, or more formally they can be defined as follows:

Definition 3.8. Reduction contexts R_{name} *are contexts of the form* $L[A]$ *where the context classes* $\mathcal{A}$ *and* $\mathcal{L}$ *are defined by the following grammar, where* s *is any expression:*

$$L \in \mathcal{L} ::= [\cdot] \mid \texttt{letrec}\ Env\ \texttt{in}\ L$$
$$A \in \mathcal{A} ::= [\cdot] \mid (A\ s) \mid (\texttt{case}_T\ A\ \texttt{of}\ alts) \mid (\texttt{seq}\ A\ s)$$

Normal order reduction $\xrightarrow{name}$ of L_{name} is defined by the following rules shown in Fig. 6 where the labeling algorithm according to Fig. 5 must be applied first. Note that the rules (seq-c), (lapp), (lcase), and (lseq) are identical to the rules for L_{LR} (in Fig. 2), but the labeling algorithm is different.

Unlike L_{LR}, the normal order reduction of L_{name} allows substitution of arbitrary expressions in (beta), (case), and (gcp) rules. An additional simplification (compared to L_{LR}) is that nested `letrec`-expressions are not flattened by reduction (*i.e.* there is no (llet)-reduction in L_{name}). As in L_{LR} the normal order reduction of L_{name} has reduction rules (lapp), (lcase), and (lseq) to move `letrec`-expressions out of an application, a `seq`-expression, or a `case`-expression.

Note that $\xrightarrow{name}$ is unique. An L_{name}-WHNF is defined as an expression either of the form $L[\lambda x.s]$ or of the form $L[(c\ s_1\ \dots\ s_{ar(c)})]$ where L is an $\mathcal{L}$ context. Let $\mathbb{A}_{name}$ be the set of L_{name}-WHNFs, then the calculus L_{name} is defined by the tuple $(\mathbb{E}_{\mathcal{L}}, \mathbb{C}_{\mathcal{L}}, \xrightarrow{name}, \mathbb{A}_{name})$ (modulo α-equivalence).

3.3 The Extended Lazy Lambda Calculus

In this subsection we give a short description of the lazy lambda calculus [Abr90] extended by data constructors, `case`-expressions and `seq`-expressions, denoted with L_{lcc}. Unlike the calculi L_{name} and L_{LR}, this calculus has no `letrec`-expressions. The set $\mathbb{E}_\lambda$ of L_{lcc}-expressions is that of the usual (untyped) lambda calculus extended by data constructors, `case`, and `seq`:

$$r, s, t, r_i, s_i, t_i \in \mathbb{E}_\lambda ::= x \mid (s\ t) \mid (\lambda x.s) \mid (c\ s_1 \dots s_{\mathrm{ar}(c)}) \mid (\texttt{case}_T\ s\ \texttt{of}\ alts) \mid (\texttt{seq}\ s\ t)$$

Contexts $\mathbb{C}_\lambda$ are $\mathbb{E}_\lambda$-expressions where a subexpression is replaced by the hole $[\cdot]$. The set $\mathbb{A}_{lcc}$ of *answers* (or also *values*) are the L_{lcc}-abstractions and constructor applications. Reduction contexts $\mathcal{R}_{lcc}$ are defined by the following grammar, where s is any $\mathbb{E}_\lambda$-expression:

$$R_{lcc} \in \mathcal{R}_{lcc} := [\cdot] \mid (R_{lcc}\ s) \mid \texttt{case}_T\ R_{lcc}\ \texttt{of}\ alts \mid \texttt{seq}\ R_{lcc}\ s$$

An $\xrightarrow{lcc}$-reduction is defined by the three rules shown in Fig. 7, and thus the calculus L_{lcc} is defined by the tuple $(\mathbb{E}_\lambda, \mathbb{C}_\lambda, \xrightarrow{lcc}, \mathbb{A}_{lcc})$ (modulo α-equivalence).

(nbeta)	$R_{lcc}[((\lambda x.s)\ t)] \xrightarrow{lcc} R_{lcc}[s[t/x]]$
(ncase)	$R_{lcc}[\mathtt{case}_T\ (c\ s_1 \ldots s_{\mathrm{ar}(c)})\ \mathtt{of}\ \ldots ((c\ x_1 \ldots x_{\mathrm{ar}(c)}) \to t) \ldots)]$ $\xrightarrow{lcc} t[s_1/x_1, \ldots, s_{\mathrm{ar}(c)}/x_{\mathrm{ar}(c)}]$
(nseq)	$R_{lcc}[\mathtt{seq}\ v\ s] \xrightarrow{lcc} R_{lcc}[s]$, if v is an abstraction or a constructor application

Fig. 7. Normal order reduction $\xrightarrow{lcc}$ of L_{lcc}

4 Properties of Similarity and Equivalences in L_{lcc}

An applicative bisimilarity for L_{lcc} and other alternative definitions are presented in subsection 4.2. As a preparation, we first analyze similarity for deterministic calculi in general.

4.1 Characterizations of Similarity in Deterministic Calculi

In this section we prove that for deterministic calculi (see Def. 2.1), the applicative similarity and its generalization to extended calculi, defined as the greatest fixpoint of an operator on relations, is equivalent to the inductive definition using Kleene's fixpoint theorem.

This implies that for deterministic calculi employing only beta-reduction, applicative similarity can be equivalently defined as $s \preccurlyeq t$, iff for all $n \geq 0$ and closed expressions $r_i, i = 1, \ldots, n$, the implication $(s\ r_1 \ldots r_n){\downarrow_D} \implies (t\ r_1 \ldots r_n){\downarrow_D}$ holds, provided the calculus is convergence-admissible, which means that for all r: $(s\ r){\downarrow_D} v \iff \exists v' : s{\downarrow_D} v' \wedge (v'\ r){\downarrow_D} v$ (see Def. 4.5).

This approach has a straightforward extension to calculi with other types of reductions, such as case- and seq-reductions. The calculi may also consist of a set of open expressions, contexts, and answers, as well as a subcalculus consisting of closed expressions, closed contexts and closed answers. We will use convergence-admissibility only for closed variants of the calculi.

In the following we assume $D = (\mathbb{E}, \mathbb{C}, \to, \mathbb{A})$ to be an untyped deterministic calculus and $\mathcal{Q} \subseteq \mathbb{C}$ be a set of functions on expressions. Note that the relations $\preccurlyeq_{D,\mathcal{Q}}$ and $\leq_{D,\mathcal{Q}}$ are defined in Definitions 2.5 and 2.6, respectively.

Lemma 4.1. *For all expressions $s_1, s_2 \in \mathbb{E}$ the following holds: $s_1 \preccurlyeq_{D,\mathcal{Q}} s_2$ if, and only if, $s_1{\downarrow_D} v_1 \implies (s_2{\downarrow_D} v_2 \wedge \forall Q \in \mathcal{Q} : Q(v_1) \preccurlyeq_{D,\mathcal{Q}} Q(v_2))$.*

Proof. Since $\preccurlyeq_{D,\mathcal{Q}}$ is a fixpoint of $F_{D,\mathcal{Q}}$, we have $\preccurlyeq_{D,\mathcal{Q}} = F_{D,\mathcal{Q}}(\preccurlyeq_{D,\mathcal{Q}})$. This equation is equivalent to the claim of the lemma.

Now we show that the operator $F_{D,\mathcal{Q}}$ is lower-continuous, and thus we can apply Kleene's fixpoint theorem to derive an alternative characterization of $\preccurlyeq_{D,\mathcal{Q}}$.

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

Proposition 4.2. *$F_{\mathcal{Q}}$ is lower-continuous* w.r.t. *countably infinite descending chains $C = \eta_1 \supseteq \eta_2 \supseteq \ldots$,* i.e. *$\mathrm{glb}(F_{\mathcal{Q}}(C)) = F_{\mathcal{Q}}(\mathrm{glb}(C))$ where $F_{\mathcal{Q}}(C)$ is the infinite descending chain $F_{\mathcal{Q}}(\eta_1) \supseteq F_{\mathcal{Q}}(\eta_2) \supseteq \ldots$.*

Proof. "$\supseteq$": Since $\mathrm{glb}(C) = \bigcap\limits_{i=1}^{\infty} \eta_i$, we have for all i: $\mathrm{glb}(C) \subseteq \eta_i$. Applying monotonicity of $F_{\mathcal{Q}}$ yields $F_{\mathcal{Q}}(\mathrm{glb}(C)) \subseteq F_{\mathcal{Q}}(\eta_i)$ for all i. This implies $F_{\mathcal{Q}}(\mathrm{glb}(C)) \subseteq \bigcap\limits_{i=1}^{\infty} F_{\mathcal{Q}}(\eta_i)$, *i.e.* $F_{\mathcal{Q}}(\mathrm{glb}(C)) \subseteq \mathrm{glb}(F_{\mathcal{Q}}(C))$.

"$\subseteq$": Let $(s_1, s_2) \in \mathrm{glb}(F_{\mathcal{Q}}(C))$, *i.e.* for all i: $(s_1, s_2) \in F_{\mathcal{Q}}(\eta_i)$. Unfolding the definition of $F_{\mathcal{Q}}$ gives: $\forall i : s_1{\downarrow_D} v_1 \implies (s_2{\downarrow_D} v_2 \wedge \forall Q \in \mathcal{Q} : Q(v_1)\, \eta_i\, Q(v_2))$. Now we can move the universal quantifier for i inside the formula: $s_1{\downarrow_D} v_1 \implies (s_2{\downarrow_D} v_2 \wedge \forall Q \in \mathcal{Q} : \forall \mathrm{i}\!: Q(v_1)\, \eta_i\, Q(v_2))$. This is equivalent to $s_1{\downarrow_D} v_1 \implies (s_2{\downarrow_D} v_2 \wedge \forall Q \in \mathcal{Q} : Q(v_1)\, (\bigcap\limits_{i=1}^{\infty} \eta_i)\, Q(v_2))$ or $s_1{\downarrow_D} v_1 \implies (s_2{\downarrow_D} v_2 \wedge \forall Q \in \mathcal{Q} : (Q(v_1), Q(v_2)) \in \mathrm{glb}(C))$ and thus $(s_1, s_2) \in F_{\mathcal{Q}}(\mathrm{glb}(C))$.

Definition 4.3. *Let* $\preccurlyeq_{D,\mathcal{Q},i}$ *for* $i \in \mathbb{N}_0$ *be defined as follows:*

$$\preccurlyeq_{D,\mathcal{Q},0} = \mathbb{E} \times \mathbb{E} \quad \mathit{and} \quad \preccurlyeq_{D,\mathcal{Q},i} = F_{D,\mathcal{Q}}(\preccurlyeq_{D,\mathcal{Q},i-1}) \mathit{if}\ i > 0$$

Theorem 4.4. $\preccurlyeq_{D,\mathcal{Q}} = \bigcap_{i=1}^{\infty} \preccurlyeq_{D,\mathcal{Q},i}$

Proof. This follows by Kleene's fixpoint theorem, since $F_{\mathcal{Q}}$ is monotonous and lower-continuous, and since $\preccurlyeq_{D,\mathcal{Q},i+1} \subseteq \preccurlyeq_{D,\mathcal{Q},i}$ for all $i \geq 0$.

This representation of $\preccurlyeq_{D,\mathcal{Q}}$ allows *inductive* proofs to show similarity. Now we show that $\mathcal{Q}$-similarity is identical to $\leq_{D,\mathcal{Q}}$ under moderate conditions, *i.e.* our characterization result will only apply if the underlying calculus is convergence-admissible *w.r.t.* $\mathcal{Q}$:

Definition 4.5. *An untyped deterministic calculus* $(\mathbb{E}, \mathbb{C}, \rightarrow, \mathbb{A})$ *is* convergence-admissible *w.r.t.* $\mathcal{Q}$ *if, and only if* $\forall Q \in \mathcal{Q}, s \in \mathbb{E}, v \in \mathbb{A} : \mathbb{Q}(\sim){\downarrow_{\mathbb{D}}}\approx \iff \exists \approx' : \sim{\downarrow_{\mathbb{D}}}\approx' \wedge \mathbb{Q}(\approx'){\downarrow_{\mathbb{D}}}\approx$

Convergence-admissibility can also be seen as a restriction on choosing the set $\mathcal{Q}$: In most calculi (subsets of) reduction contexts satisfy the property for convergence-admissibility, while including non-reduction contexts into $\mathcal{Q}$ usually breaks convergence-admissibility.

We show some helpful properties of $\leq_{D,\mathcal{Q}}$.

Lemma 4.6. *Let* $(\mathbb{E}, \mathbb{C}, \rightarrow, \mathbb{A})$ *be convergence-admissible* w.r.t. $\mathcal{Q}$. *Then the following holds:*

- $s_1 \leq_{D,\mathcal{Q}} s_2 \implies Q(s_1) \leq_{D,\mathcal{Q}} Q(s_2)$ *for all* $Q \in \mathcal{Q}$
- $s_1 \leq_{D,\mathcal{Q}} s_2, s_1{\downarrow_D}v_1,$ *and* $s_2{\downarrow_D}v_2 \implies v_1 \leq_{D,\mathcal{Q}} v_2$

Proof. The first part is easy to verify. For the second part it is important that D is determinstic. Let $s_1 \leq_{D,\mathcal{Q}} s_2$, and $s_1{\downarrow_D}v_1$, $s_2{\downarrow_D}v_2$ hold. Assume that $Q_1(\ldots(Q_n(v_1))){\downarrow_D}v_1'$ for some $n \geq 0$ where all $Q_i \in \mathcal{Q}$. Convergence-admissibility implies $Q_1(\ldots(Q_n((s_1))){\downarrow_D}v_1'$. Now $s_1 \leq_{D,\mathcal{Q}} s_2$ implies $Q_1(\ldots(Q_n(s_2))){\downarrow_D}v_2'$. Finally, convergence-admissibility (applied multiple times) shows that $s_2{\downarrow_D}v_2$ and $Q_1(\ldots(Q_n(v_2))){\downarrow_D}v_2'$ holds.

We prove that $\preccurlyeq_{D,\mathcal{Q}}$ respects functions $Q \in \mathcal{Q}$ provided the underlying deterministic calculus is convergence-admissible *w.r.t.* $\mathcal{Q}$:

Lemma 4.7. *Let* $(\mathbb{E}, \mathbb{C}, \rightarrow, \mathbb{A})$ *be convergence-admissible* w.r.t. $\mathcal{Q}$. *Then for all* $s_1, s_2 \in E : s_1 \preccurlyeq_{D,\mathcal{Q}} s_2 \implies Q(s_1) \preccurlyeq_{D,\mathcal{Q}} Q(s_2)$ *for all* $Q \in \mathcal{Q}$

Proof. Let $s_1 \preccurlyeq_{D,\mathcal{Q}} s_2$, $Q_0 \in \mathcal{Q}$, and $Q_0(s_1){\downarrow_D}v_1$. By convergence admissibility $s_1{\downarrow_D}v_1'$ holds and $Q_0(v_1'){\downarrow_D}v_1$. Since $s_1 \preccurlyeq_{D,\mathcal{Q}} s_2$ this implies $s_2{\downarrow_D}v_2'$ and for all $Q \in \mathcal{Q} : Q(v_1') \preccurlyeq_{D,\mathcal{Q}} Q(v_2')$. Hence, from $Q_0(v_1'){\downarrow_D}v_1$ we derive $Q_0(v_2'){\downarrow_D}v_2$. Convergence admissibility now implies $Q_0(s_2){\downarrow_D}v_2$.

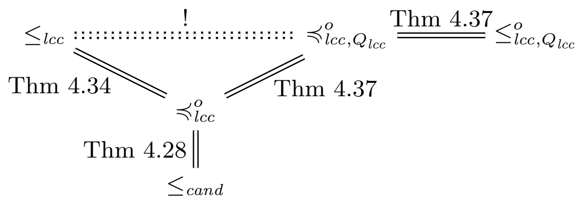
It remains to show for all $Q \in \mathcal{Q}$: $Q(v_1) \preccurlyeq_{D,\mathcal{Q}} Q(v_2)$: Since $Q_0(v_1'){\downarrow_D}v_1$ and $Q_0(v_2'){\downarrow_D}v_2$, applying Lemma 4.1 to $Q_0(v_1') \preccurlyeq_{D,\mathcal{Q}} Q_0(v_2')$ implies $Q(v_1) \preccurlyeq_{D,\mathcal{Q}} Q(v_2)$ for all $Q \in \mathcal{Q}$.

We now prove that $\leq_{D,\mathcal{Q}}$ and $\mathcal{Q}$-similarity coincide for convergence-admissible deterministic calculi:

Theorem 4.8. *Let* $(\mathbb{E}, \mathbb{C}, \rightarrow, \mathbb{A})$ *be convergence-admissible* w.r.t. $\mathcal{Q}$. *Then* $\leq_{D,\mathcal{Q}} = \preccurlyeq_{D,\mathcal{Q}}$.

Proof. "$\subseteq$": Let $s_1 \leq_{D,\mathcal{Q}} s_2$. We use Theorem 4.4 and show $s_1 \preccurlyeq_{D,\mathcal{Q},i} s_2$ for all i. We use induction on i. The base case ($i = 0$) obviously holds. Let $i > 0$ and let $s_1{\downarrow_D}v_1$. Then $s_1 \leq_{D,\mathcal{Q}} s_2$ implies $s_2{\downarrow_D}v_2$. Thus, it is sufficient to show that $Q(v_1) \preccurlyeq_{D,\mathcal{Q},i-1} Q(v_2)$ for all $Q \in \mathcal{Q}$: As induction hypothesis we use that $s_1 \leq_{D,\mathcal{Q}} s_2 \implies s_1 \preccurlyeq_{D,\mathcal{Q},i-1} s_2$ holds. Using Lemma 4.6 twice and $s_1 \leq_{D,\mathcal{Q}} s_2$, we have $Q(v_1) \leq_{D,\mathcal{Q}} Q(v_2)$. The induction hypothesis shows that $Q(v_1) \preccurlyeq_{D,\mathcal{Q},i-1} Q(v_2)$. Now the definition of $\preccurlyeq_{D,\mathcal{Q},i}$ is satisfied, which shows $s_1 \preccurlyeq_{D,\mathcal{Q},i} s_2$.

"$\supseteq$": Let $s_1 \preccurlyeq_{D,\mathcal{Q}} s_2$. By induction on the number n of Q-contexts we show $\forall n, Q_i \in \mathcal{Q}$: $Q_1(\ldots(Q_n(s_1))){\downarrow_D} \implies Q_1(\ldots(Q_n(s_2))){\downarrow_D}$. The base case follows from $s_1 \preccurlyeq_{D,\mathcal{Q}} s_2$. For the induction step we use the following induction hypothesis: $t_1 \preccurlyeq_{D,\mathcal{Q}} t_2 \implies \forall j < n, Q_i \in \mathcal{Q} : Q_1(\ldots(Q_j(t_1))){\downarrow_D} \implies Q_1(\ldots(Q_j(t_2))){\downarrow_D}$ for all t_1, t_2. Let $Q_1(\ldots(Q_n(s_1))){\downarrow_D}$. From Lemma 4.7 we have $r_1 \preccurlyeq_{D,\mathcal{Q}} r_2$, where $r_i = Q_n(s_i)$. Now the induction hypothesis shows that $Q_1(\ldots(Q_{n-1}(r_1))){\downarrow_D} \implies Q_1(\ldots(Q_{n-1}(r_2))){\downarrow_D}$ and thus $Q_1(\ldots(Q_n(s_2))){\downarrow_D}$.



$\leq_{lcc}$	contextual preorder in *lcc*
$\preccurlyeq^o_{lcc}$	open extension of similarity in *lcc*
$\leq_{cand}$	co-inductively defined candidate relation for Howe's technique
$\preccurlyeq^o_{lcc,Q_{lcc}}$	open extension of $\mathcal{Q}$-similarity in L_{lcc} with $\mathcal{Q} = Q_{lcc}$
$\leq^o_{lcc,Q_{lcc}}$	open extension of contextual preorder in L_{lcc} restricted to contexts Q_{lcc}

Fig. 8. Structure of soundness and completeness proofs for similarities in L_{lcc}. The =!= indicates a required equality which can only be proved via Howe's technique.

4.2 Applicative Simulation in L_{lcc}

In this section we will show that applicative similarity and contextual preorder coincide in L_{lcc}.

Notation. In abuse of notation we use higher order abstract syntax as *e.g.* in [How89] for the proof and write $\tau(..)$ for an expression with top operator τ, which may be all possible term constructors, like `case`, application, a constructor, `seq`, or λ, and θ for an operator that may be the head of a value, *i.e.* a constructor or λ.

Definition 4.9. *For a relation η on closed $\mathbb{E}_\lambda$-expressions η^o is the open extension on L_{lcc}: For (open) $\mathbb{E}_\lambda$-expressions s_1, s_2, the relation $s_1\ \eta^o\ s_2$ holds, if for all substitutions σ such that $\sigma(s_1), \sigma(s_2)$ are closed, the relation $\sigma(s_1)\ \eta\ \sigma(s_2)$ holds. Conversely, for binary relations μ on open expressions, $(\mu)^c$ is the restriction to closed expressions.*

A relation μ is *operator-respecting*, iff $s_i\ \mu\ t_i$ for $i = 1, \ldots, n$ implies $\tau(s_1, \ldots, s_n)\ \mu\ \tau(t_1, \ldots, t_n)$. Note that τ and θ may also represent the binding λ using $\lambda(x.s)$ as representing $\lambda x.s$. For consistency of terminology and treatment with that in other papers such as [How89], we assume that removing the top constructor λx in relations is done after a renaming. For example, $\lambda x.s\ \mu\ \lambda y.t$ is renamed before further treatment to $\lambda z.s[z/x]\ \mu\ \lambda z.t[z/y]$ for a fresh variable z.

Plan of Subsection 4.2. We start by explaining the subgoals of the soundness and completeness proofs for similarities in L_{lcc} and its structure, illustrated in Fig. 8. The main result we want to show is that contextual preorder $\leq_{lcc}$ and $\preccurlyeq^o_{lcc,Q_{lcc}}$ coincide, where $\preccurlyeq^o_{lcc,Q_{lcc}}$ is $\mathcal{Q}$-similarity introduced in Definition 2.5 instantiated with a specific small set Q_{lcc} of L_{lcc}-contexts. $\mathcal{Q}$-similarity does not allow a direct proof of soundness and completeness for contextual equivalence using Howe's method [How89,How96], since it is not stated in terms of the syntactic form of values derived by evaluation. We overcome this obstacle by defining another similarity $\preccurlyeq_{lcc}$ in L_{lcc} for which we will perform the proof of soundness and completeness w.r.t. contextual preorder. Since the definition of $\preccurlyeq_{lcc}$ does not obviously imply that $\preccurlyeq_{lcc}$ is a precongruence, a candidate relation $\leq_{cand}$ is defined, which is trivially compatible with contexts, but needs to be shown to be transitive. After proving $\leq_{cand} = \preccurlyeq^o_{lcc}$, i.e. that $\preccurlyeq^o_{lcc}$ is a precongruence, soundness of $\preccurlyeq^o_{lcc}$ w.r.t. contextual preorder $\leq_{lcc}$ follows. Completeness can then also be proven. In a second step we prove that $\preccurlyeq^o_{lcc,Q_{lcc}}$ is sound and complete for contextual equivalence, i.e. $\leq_{lcc} = \preccurlyeq^o_{lcc,Q_{lcc}}$. After showing that L_{lcc} is convergence-admissible we are also able to show that the inductive description $\leq_{lcc,Q_{lcc}}$ of $\mathcal{Q}$-similarity coincides with $\preccurlyeq_{lcc,Q_{lcc}}$.

Another obstacle is that the contextual preorder contains the irregularity $\lambda x.\Omega \leq_{lcc} c\ s_1 \ldots s_n$ for any constructor c. This requires an adapted definition of the similarity relation, and a slightly modified proof route.

In the following let *cBot* be the set of $\mathbb{E}_\lambda$-expressions s with the property that for all $\mathbb{E}_\lambda$-substitutions σ: if $\sigma(s)$ is closed, then $\sigma(s)\Uparrow_{lcc}$. That $\lambda x.s \leq_{lcc} (c\ s_1 \ldots s_n)$ indeed holds is shown in Proposition 4.32. Now we define an applicative similarity $\preccurlyeq_{lcc}$ in L_{lcc} analogous to [How89,How96], where this irregularity is taken into account.

Definition 4.10 (Similarity in L_{lcc}). *Let η be a binary relation on closed $\mathbb{E}_\lambda$-expressions. Let F_{lcc} be the following operator on relations on closed $\mathbb{E}_\lambda$-expressions:*
$s\ F_{lcc}(\eta)\ t$ holds iff

– $s{\downarrow_{lcc}}\lambda x.s' \implies \big(t{\downarrow_{lcc}}\lambda x.t'$ *and* $s'\ \eta^o\ t'$*, or*
$t{\downarrow_{lcc}}(c\ t'_1 \dots t'_n)$ *and* $s' \in cBot\big)$
– $s{\downarrow_{lcc}}(c\ s'_1 \dots s'_n) \implies \big(t{\downarrow_{lcc}}(c\ t'_1 \dots t'_n)$ *and the relation* $s'_i\ \eta\ t'_i$ *holds for all* $i\big)$

Similarity $\preccurlyeq_{lcc}$ *is defined as the greatest fixpoint of the operator* F_{lcc}. *Bisimilarity* $\simeq_{lcc}$ *is defined as* $s \simeq_{lcc} t$ *iff* $s \preccurlyeq_{lcc} t \wedge t \preccurlyeq_{lcc} s$.

Note that the operator F_{lcc} is monotone, hence the greatest fixpoint $\preccurlyeq_{lcc}$ exists.

4.2.1 Similarity and Contextual Preorder Coincide in L_{lcc} Although applying Howe's proof technique is standard, for the sake of completeness, and to demonstrate that the irregularity $\lambda x.\Omega \leq_{lcc} (c\, s_1 \dots s_n)$ can also be treated, we will explicitly show in this section that $\preccurlyeq_{lcc}^o \ = \ \leq_{lcc}$ using Howe's method [How89,How96].

Lemma 4.11. *For a relation* η *on closed expressions it holds* $((\eta)^o)^c = \eta$*, and also* $s\ \eta^o\ t$ *implies* $\sigma(s)\ \eta^o\ \sigma(t)$ *for any substitution* σ. *For a relation* μ *on open expressions,* $\mu \subseteq ((\mu)^c)^o$ *is equivalent to* $s\ \mu\ t \implies \sigma(s)\ (\mu)^c\ \sigma(t)$ *for all closing substitutions* σ.

Proposition 4.12 (Co-Induction). *The principle of co-induction for the greatest fixpoint of* F_{lcc} *shows that for every relation* η *on closed expressions with* $\eta \subseteq F_{lcc}(\eta)$*, we derive* $\eta \subseteq \preccurlyeq_{lcc}$. *This obviously also implies* $(\eta)^o \subseteq (\preccurlyeq_{lcc})^o$.

The fixpoint property of $\preccurlyeq_{lcc}$ implies:

Lemma 4.13. *For a closed value* $\theta_1(s_1,\dots,s_n)$*, and a closed term* t *with* $\theta_1(s_1,\dots,s_n) \preccurlyeq_{lcc} t$*, we have* $t{\downarrow_{lcc}}\ \theta_2(t_1,\dots,t_n)$*, and there are two cases:*

1. $\theta_1 = \theta_2$ *are constructors or* λ *and* $s_i \preccurlyeq_{lcc}^o t_i$ *for all* i.
2. $\theta_1(s_1,\dots,s_n) = \lambda(x.s)$ *with* $s \in cBot$ *and* θ_2 *is a constructor.*

Lemma 4.14. *For two expressions* s,t*:* $s \in cBot$ *implies* $s \preccurlyeq_{lcc}^o t$. *Thus any two expressions* $s,t \in cBot$ *are bisimilar:* $s \simeq_{lcc}^o t$.

Particular expressions in *cBot* are $(\texttt{case}\ (\lambda x.s)\ alts)$ and $(c(s_1,\dots,s_n)\ a_1 \dots a_m)$ for $m \geq 1$; also $s \in cBot$ implies that $(s\ t)$, $(\texttt{seq}\ s\ t)$, $(\texttt{case}\ s\ alts)$ and $\sigma(s)$ are also in *cBot*.

Lemma 4.15. *The relations* $\preccurlyeq_{lcc}$ *and* $\preccurlyeq_{lcc}^o$ *are reflexive and transitive.*

Proof. Reflexivity follows by showing that $\eta := \ \preccurlyeq_{lcc} \cup \{(s,s) \mid s \in \mathbb{E}_\lambda, s \text{ closed}\}$ satisfies $\eta \subseteq F_{lcc}(\eta)$. Transitivity follows by showing that $\eta := \ \preccurlyeq_{lcc} \cup (\preccurlyeq_{lcc} \circ \preccurlyeq_{lcc})$ satisfies $\eta \subseteq F_{lcc}(\eta)$ and then using the co-induction principle.

The goal in the following is to show that $\preccurlyeq_{lcc}$ is a precongruence. This proof proceeds by defining a congruence candidate $\leq_{cand}$ as a closure of $\preccurlyeq_{lcc}$ within contexts, which obviously is operator-respecting, but transitivity needs to be shown. By proving that $\preccurlyeq_{lcc}$ and $\leq_{cand}$ coincide, on the one hand transitivity of $\leq_{cand}$ follows (since $\preccurlyeq_{lcc}$ is transitive) and on the other hand (and more importantly) it follows that $\preccurlyeq_{lcc}$ is operator-respecting (since $\leq_{cand}$ is operator-respecting) and thus a congruence.

Definition 4.16. *The precongruence candidate* $\leq_{cand}$ *is a binary relation on open expressions and is defined as the greatest fixpoint of the monotone operator* F_{cand} *on relations on all expressions:*

1. $x\ F_{cand}(\eta)\ s$ *iff* $x \preccurlyeq_{lcc}^o s$.
2. $\tau(s_1,\dots,s_n)\ F_{cand}(\eta)\ s$ *iff there is some expression* $\tau(s'_1,\dots,s'_n) \preccurlyeq_{lcc}^o s$ *with* $s_i\ \eta\ s'_i$ *for* $i = 1,\dots,n$.

Lemma 4.17. *If some relation* η *satisfies* $\eta \subseteq F_{cand}(\eta)$*, then* $\eta \subseteq \ \leq_{cand}$.

Since $\leq_{cand}$ is a fixpoint of F_{cand}, we have:

Lemma 4.18.

1. $x \leq_{cand} s$ *iff* $x \preccurlyeq^o_{lcc} s$.
2. $\tau(s_1,\ldots,s_n) \leq_{cand} s$ *iff there is some expression* $\tau(s'_1,\ldots,s'_n) \preccurlyeq^o_{lcc} s$ *with* $s_i \leq_{cand} s'_i$ *for* $i = 1,\ldots,n$.

Some technical facts about the precongruence candidate are now proved:

Lemma 4.19. *The following properties hold:*

1. $\leq_{cand}$ *is reflexive.*
2. $\leq_{cand}$ *and* $(\leq_{cand})^c$ *are operator-respecting.*
3. $\preccurlyeq^o_{lcc} \subseteq \leq_{cand}$ *and* $\preccurlyeq_{lcc} \subseteq (\leq_{cand})^c$.
4. $\leq_{cand} \circ \preccurlyeq^o_{lcc} \subseteq \leq_{cand}$.
5. $(s \leq_{cand} s' \wedge t \leq_{cand} t') \implies t[s/x] \leq_{cand} t'[s'/x]$.
6. $s \leq_{cand} t$ *implies that* $\sigma(s) \leq_{cand} \sigma(t)$ *for every substitution* σ.
7. $\leq_{cand} \subseteq ((\leq_{cand})^c)^o$

Proof. Parts (1) – (3) can be shown by structural induction and using reflexivity of $\preccurlyeq^o_{lcc}$. Part (4) follows from the definition, Lemma 4.18, and transitivity of $\preccurlyeq^o_{lcc}$.

For part (5) let $\eta := \leq_{cand} \cup \{(r[s/x], r'[s'/x]) \mid r \leq_{cand} r'\}$. Using co-induction it suffices to show that $\eta \subseteq F_{cand}(\eta)$: In the case $x \leq_{cand} r'$, we obtain $x \preccurlyeq^o_{lcc} r'$ from the definition, and $s' \preccurlyeq^o_{lcc} r'[s'/x]$ and thus $x[s/x] \leq_{cand} r'[s'/x]$. In the case $y \leq_{cand} r$, we obtain $y \preccurlyeq^o_{lcc} r'$ from the definition, and $y[s/x] = y \preccurlyeq^o_{lcc} r'[s'/x]$ and thus $y = y[s/x] \leq_{cand} r'[s'/x]$. If $r = \tau(r_1,\ldots,r_n)$, $r \leq_{cand} r'$ and $r[s/x]\ \eta\ r'[s'/x]$, then there is some $\tau(r'_1,\ldots,r'_n) \preccurlyeq^o_{lcc} r'$ with $r_i \leq_{cand} r'_i$. W.l.o.g. bound variables have fresh names. We have $r_i[s/x]\ \eta\ r'_i[s'/x]$ and $\tau(r'_1,\ldots,r'_n)[s'/x] \preccurlyeq^o_{lcc} r'[s'/x]$. Thus $r[s/x]\ F_{cand}(\eta)\ r'[s'/x]$.

Part (6) follows from item (5). Part (7) follows from item (6) and Lemma 4.11. □

Lemma 4.20. *The middle expression in the definition of* $\leq_{cand}$ *can be chosen to be closed if* s, t *are closed: Let* $s = \tau(s_1,\ldots,s_{\mathrm{ar}(\tau)})$*, such that* $s \leq_{cand} t$ *holds. Then there are operands* s'_i*, such that* $\tau(s'_1,\ldots,s'_{\mathrm{ar}(\tau)})$ *is closed,* $\forall i : s_i \leq_{cand} s'_i$ *and* $\tau(s'_1,\ldots,s'_{\mathrm{ar}(\tau)}) \preccurlyeq^o_{lcc} t$.

Proof. The definition of $\leq_{cand}$ implies that there is an expression $\tau(s''_1,\ldots,s''_{\mathrm{ar}(\tau)})$ such that $s_i \leq_{cand} s''_i$ for all i and $\tau(s''_1,\ldots,s''_{\mathrm{ar}(\tau)}) \preccurlyeq^o_{lcc} t$. Let σ be the substitution with $\sigma(x) := r_x$ for all $x \in FV(\tau(s''_1,\ldots,s''_{\mathrm{ar}(\tau)}))$, where r_x is any closed expression. Lemma 4.19 now shows that $s_i = \sigma(s_i) \leq_{cand} \sigma(s''_i)$ holds for all i. The relation $\sigma(\tau(s''_1,\ldots,s''_{\mathrm{ar}(\tau)})) \preccurlyeq^o_{lcc} t$ holds, since t is closed and due to the definition of an open extension. The requested expression is $\tau(\sigma(s''_1),\ldots,\sigma(s''_{\mathrm{ar}(\tau)}))$.

Since reduction $\xrightarrow{lcc}$ is deterministic:

Lemma 4.21. *If* $s \xrightarrow{lcc} s'$*, then* $s' \preccurlyeq^o_{lcc} s$ *and* $s \preccurlyeq^o_{lcc} s'$.

Lemmas 4.21 and 4.19 imply that $\leq_{cand}$ is right-stable *w.r.t.* reduction:

Lemma 4.22. *If* $s \leq_{cand} t$ *and* $t \xrightarrow{lcc} t'$*, then* $s \leq_{cand} t'$.

We show that $\leq_{cand}$ is left-stable *w.r.t.* reduction:

Lemma 4.23. *Let* s, t *be closed expressions such that* $s = \theta(s_1,\ldots,s_n)$ *is a value and* $s \leq_{cand} t$*. Then there are two possibilities:*

1. $s = \lambda x.s_1$ *and* $t{\downarrow_{lcc}} c(t_1,\ldots,t_n)$*, where* c *is a constructor;*
2. *there is some closed value* $t' = \theta(t_1,\ldots,t_n)$ *with* $t \xrightarrow{lcc,*} t'$ *and for all* $i : s_i \leq_{cand} t_i$.

Proof. The definition of $\leq_{cand}$ implies that there is a closed expression $\theta(t'_1,\ldots,t'_n)$ with $s_i \leq_{cand} t'_i$ for all i and $\theta(t'_1,\ldots,t'_n) \preccurlyeq_{lcc} t$. Lemma 4.13 implies that $t{\downarrow_{lcc}}$, hence either $t \xrightarrow{lcc,*} c(t''_1,\ldots,t''_n)$ or $t \xrightarrow{lcc,*} \lambda x.t''_1$.

- First let $\theta = \lambda$. The case $t \xrightarrow{lcc,*} c(t''_1,\ldots,t''_n)$ is possibility (1) of the lemma.
  In the second case, $t \xrightarrow{lcc,*} \lambda x.t''_1$, Lemma 4.22 implies $\lambda x.s_1 \leq_{cand} \lambda x.t''_1$. Definition of $\leq_{cand}$ and Lemma 4.20 now show that there is some closed $\lambda x.t'''_1$ with $s_1 \leq_{cand} t'''_1$ and $\lambda x.t'''_1 \preccurlyeq_{lcc} \lambda x.t''_1$. The latter relation implies $t'''_1 \preccurlyeq^o_{lcc} t''_1$, which shows $s'_1 \leq_{cand} t''_1$ by Lemma 4.19 (4).

– If θ is a constructor, then there is a closed expression $\theta(t'_1, \dots, t'_n)$ with $s_i \leq_{cand} t'_i$ for all i and $\theta(t'_1, \dots, t'_n) \preccurlyeq_{lcc} t$. The properties of $\preccurlyeq_{lcc}$ imply that $t \xrightarrow{lcc,*} \theta(t''_1, \dots, t''_n)$ with $t'_i \preccurlyeq_{lcc} t''_i$ for all i. By definition of $\leq_{cand}$ and Lemma 4.19 (4), we obtain $s_i \leq_{cand} t''_i$ for all i. □

Proposition 4.24. *Let s, t be closed expressions, $s \leq_{cand} t$ and $s \xrightarrow{lcc} s'$ where s is the redex. Then $s' \leq_{cand} t$.*

Proof. The relation $s \leq_{cand} t$ implies that $s = \tau(s_1, \dots, s_n)$ and by Lemma 4.20 there is some closed $t' = \tau(t'_1, \dots, t'_n)$ with $s_i \leq_{cand} t'_i$ for all i and $t' \preccurlyeq^o_{lcc} t$.

– For the (nbeta)-reduction, $s = (s_1\ s_2)$, where $s_1 = (\lambda x.s'_1)$, s_2 is a closed term, and $t' = (t'_1\ t'_2)$. The relation $(\lambda x.s'_1) = s_1 \leq_{cand} t'_1$ implies that there exists a closed expression $\lambda x.t''_1 \preccurlyeq^o_{lcc} t'_1$ with $s'_1 \leq_{cand} t''_1$.
  ∘ The first case is $t'_1 \xrightarrow{lcc,*} c(\dots)$ and $t''_1 \in cBot$. Lemma 4.19 implies $\lambda x.s'_1 \leq_{cand} \lambda x.t''_1$, and again by Lemma 4.19, we derive $s'_1[s_2/x] \leq_{cand} t''_1[s_2/x]$, where $t''_1[s_2/x] \in cBot$. Then $t''_1[s_2/x] \preccurlyeq^o_{lcc} t$ by Lemma 4.14, which implies $s'_1[s_2/x] \leq_{cand} t$. Since $s \xrightarrow{lcc} s'_1[s_2/x]$, the lemma is proven for this case.
  ∘ The second case is $t'_1 \xrightarrow{lcc,*} \lambda x.t'''_1$ with $t''_1 \preccurlyeq^o_{lcc} t'''_1$. We also obtain $\lambda x.t''_1 \preccurlyeq^o_{lcc} \lambda x.t'''_1$, and by the properties of $\preccurlyeq^o_{lcc}$ w.r.t. reduction, also $t''_1[t'_2/x] \preccurlyeq^o_{lcc} t'''_1[t'_2/x]$. From $t' \xrightarrow{lcc,*} t'''_1[t'_2/x]$ we obtain $t'''_1[t'_2/x] \preccurlyeq_{lcc} t$. Lemma 4.19 and transitivity of $\preccurlyeq_{lcc}$ now show $s'_1[s_2/x] \leq_{cand} t''_1[t'_2/x]$. Hence $s'_1[s_2/x] \leq_{cand} t$, again using Lemma 4.19.
– Similar arguments as for the second case apply to the case-reduction.
– Suppose, the reduction is a (nseq)-reduction. Then $s \leq_{cand} t$ and $s = (\mathtt{seq}\ s_1\ s_2)$. Lemma 4.20 implies that there is some closed $(\mathtt{seq}\ t'_1\ t'_2) \preccurlyeq^o_{lcc} t$ with $s_i \leq_{cand} t'_i$. Since s_1 is a value, Lemma 4.23 shows that there is a reduction $t'_1 \xrightarrow{lcc,*} t''_1$, where t''_1 is a value. There are the reductions $s \xrightarrow{lcc} s_2$ and $(\mathtt{seq}\ t'_1\ t'_2) \xrightarrow{lcc,*} (\mathtt{seq}\ t''_1\ t'_2) \xrightarrow{lcc} t'_2$. Since $t'_2 \preccurlyeq^o_{lcc} (\mathtt{seq}\ t'_1\ t'_2) \preccurlyeq^o_{lcc} t$, and $s_2 \leq_{cand} t'_2$, we obtain $s_2 \leq_{cand} t$. □

Proposition 4.25. *Let s, t be closed expressions, $s \leq_{cand} t$ and $s \xrightarrow{lcc} s'$. Then $s' \leq_{cand} t$.*

Proof. We use induction on the length of the path to the redex. The base case is proven in Proposition 4.24. Let $R[s], t$ be closed, $R[s] \leq_{cand} t$ and $R[s] \xrightarrow{lcc} R[s']$, where we assume that the redex s is not at the top level and that R is an L_{lcc}-reduction context. The relation $R[s] \leq_{cand} t$ implies that $R[s] = \tau(s_1, \dots, s_n)$ and that there is some closed $t' = \tau(t'_1, \dots, t'_n) \preccurlyeq^o_{lcc} t$ with $s_i \leq_{cand} t'_i$ for all i. If $s_j \xrightarrow{lcc} s'_j$, then by induction hypothesis $s'_j \leq_{cand} t'_j$. Since $\leq_{cand}$ is operator-respecting, we also obtain $R[s'] = \tau(s_1, \dots, s_{j-1}, s'_j, s_{j+1}, \dots, s_n) \leq_{cand} \tau(t'_1, \dots, t'_{j-1}, t'_j, t'_{j+1}, \dots, t'_n)$, and from $\tau(t'_1, \dots, t'_n) \preccurlyeq^o_{lcc} t$ we have $R[s'] = \tau(s_1, \dots, s_{j-1}, s'_j, s_{j+1}, \dots, s_n) \leq_{cand} t$.

Lemma 4.26. *If $\lambda x.s, \lambda x.t$ are closed, $\lambda x.s \leq_{cand} \lambda x.t$, and $t \in cBot$, then also $s \in cBot$.*

Proof. For any closed r, we also have $(\lambda x.s)\ r \leq_{cand} (\lambda x.t)\ r$, since $\leq_{cand}$ is operator-respecting. From $t \in cBot$, we obtain that $((\lambda x.t)\ r)\Uparrow_{lcc}$. Now suppose that $(\lambda x.s)\ r \xrightarrow{lcc,*} s'$, where s' is a value. Lemma 4.25 implies that $s' \leq_{cand} (\lambda x.t)\ r$. Now Lemma 4.23 shows that this is impossible. Hence $s \in cBot$.

Now we can prove an improvement of Lemma 4.23:

Lemma 4.27. *Let s, t be closed expressions such that $s = \theta(s_1, \dots, s_n)$ is a value and $s \leq_{cand} t$. Then there are two possibilities:*

1. *$s = \lambda x.s_1$, $t{\downarrow_{lcc}}c(t_1, \dots, t_n)$ where c is a constructor, and $s_1 \in cBot$.*
2. *there is some closed value $t' = \theta(t_1, \dots, t_n)$ with $t \xrightarrow{lcc,*} t'$ and for all $i : s_i \leq_{cand} t_i$.*

Proof. This follows from Lemma 4.23 and Lemma 4.26.

Now we are ready to prove that the precongruence candidate and similarity coincide.

Theorem 4.28. $(\leq_{cand})^c = \preccurlyeq_{lcc}$ *and* $\leq_{cand} = \preccurlyeq^o_{lcc}$.

Proof. Since $\preccurlyeq_{lcc} \subseteq (\leq_{cand})^c$ by Lemma 4.19, we have to show that $(\leq_{cand})^c \subseteq \preccurlyeq_{lcc}$. Therefore it is sufficient to show that $(\leq_{cand})^c$ satisfies the fixpoint equation for $\preccurlyeq_{lcc}$. We show that $(\leq_{cand})^c \subseteq F_{lcc}((\leq_{cand})^c)$. Let $s\ (\leq_{cand})^c\ t$ for closed terms s, t. We show that $s\ F_{lcc}((\leq_{cand})^c)\ t$: If $s \Uparrow_{lcc}$, then $s\ F_{lcc}((\leq_{cand})^c)\ t$ holds by Definition 4.10. If $s{\downarrow_{lcc}}\theta(s_1,\ldots,s_n)$, then $\theta(s_1,\ldots,s_n)\ (\leq_{cand})^c\ t$ by Proposition 4.25.

Lemmas 4.25 and 4.27 show that there are two possibilities:

- $t \xrightarrow{lcc,*} c(t_1,\ldots,t_n)$ for a constructor c, $s{\downarrow_{lcc}}\lambda x.s_1$, and $s_1 \in cBot$.
- $t \xrightarrow{lcc,*} \theta(t_1,\ldots,t_n)$ and for all $i : s_i \leq_{cand} t_i$.

This implies $s\ F_{lcc}((\leq_{cand})^c)\ t$. Thus the fixpoint property of $(\leq_{cand})^c$ *w.r.t.* F_{lcc} holds, and hence $(\leq_{cand})^c\ =\ \preccurlyeq_{lcc}$.

Now we prove the second part. The first part, $(\leq_{cand})^c \subseteq \preccurlyeq_{lcc}$, implies $((\leq_{cand})^c)^o \subseteq \preccurlyeq_{lcc}^o$ by monotonicity. Lemma 4.19 (7) implies $\leq_{cand}\ \subseteq ((\leq_{cand})^c)^o \subseteq \preccurlyeq_{lcc}^o$. The other direction is proven in Lemma 4.19 (3).

Since $\preccurlyeq_{lcc}^o$ is reflexive and transitive (Lemma 4.15) and $(\leq_{cand})^c$ is operator-respecting (Lemma 4.19 (2)), this immediately implies:

Corollary 4.29. *$\preccurlyeq_{lcc}^o$ is a precongruence on expressions $\mathbb{E}_\lambda$. If σ is a substitution, then $s \preccurlyeq_{lcc}^o t$ implies $\sigma(s) \preccurlyeq_{lcc}^o \sigma(t)$.*

Lemma 4.30. $\preccurlyeq_{lcc}^o\ \subseteq\ \leq_{lcc}$.

Proof. Let s, t be expressions with $s \preccurlyeq_{lcc}^o t$ such that $C[s]{\downarrow_{lcc}}$. Let σ be a substitution that replaces all free variables of $C[s], C[t]$ by Ω. The properties of the call-by-name reduction show that also $\sigma(C[s]){\downarrow_{lcc}}$. Since $\sigma(C[s]) = \sigma(C)[\sigma(s)]$, $\sigma(C[t]) = \sigma(C)[\sigma(t)]$ and since $\sigma(s) \preccurlyeq_{lcc}^o \sigma(t)$, we obtain from the precongruence property of $\preccurlyeq_{lcc}^o$ that also $\sigma(C[s]) \preccurlyeq_{lcc} \sigma(C[t])$. Hence $\sigma(C[t]){\downarrow_{lcc}}$. This is equivalent to $C[t]{\downarrow_{lcc}}$, since free variables are replaced by Ω, and thus they cannot overlap with redexes. Hence $\preccurlyeq_{lcc}^o\ \subseteq\ \leq_{lcc}$.

Corollary 4.31. *$s \xrightarrow{lcc} s'$ implies $s \sim_{lcc} s'$. Thus the reduction rules of the calculus L_{lcc} are correct w.r.t. $\sim_{lcc}$ in any context.*

Proof. This follows from Lemmas 4.21 and 4.30.

Now we show a characterization for $\mathbb{E}_\lambda$-expressions, which includes the previously mentioned irregularity of $\leq_{lcc}$:

Proposition 4.32. *Let s be a closed L_{lcc}-expression. Then there are three cases: $s \sim_{lcc} \Omega$, $s \sim_{lcc} \lambda x.s'$ for some s', $s \sim_{lcc} (c\ s_1 \ldots s_n)$ for some terms $s_1,\ldots,s_n$ and constructor c.*

For two closed L_{lcc}-expressions s, t with $s \leq_{lcc} t$: Either $s \sim_{lcc} \Omega$, or $s \sim_{lcc} (c\ s_1 \ldots s_n)$, $t \sim_{lcc} (c\ t_1 \ldots t_n)$ and $s_i \leq_{lcc} t_i$ for all i for some terms $s_1,\ldots,s_n,t_1,\ldots,t_n$ and constructor c, or $s \sim_{lcc} \lambda x.s'$ and $t \sim_{lcc} \lambda x.t'$ for some expressions s', t' with $s' \leq_{lcc}^o t'$, or $s \sim_{lcc} \lambda x.s'$ and $t \sim_{lcc} (c\ t_1 \ldots t_n)$ for some term $s' \in cBot$, expressions $t_1,\ldots,t_n$ and constructor c.

Proof. We apply Lemma 4.30. Corollary 4.31 then shows that using reduction the classification of closed expressions into the classes w.r.t $\sim_{lcc}$ holds.

For two closed L_{lcc}-expressions s, t with $s \preccurlyeq_{lcc} t$: we obtain the classification in the lemma but with $\preccurlyeq_{lcc}$ instead of $\leq_{lcc}$. For the three cases $s \simeq_{lcc} \Omega$, both s, t are equivalent to constructor expressions, and both s, t are equivalent to abstractions, we obtain also that $s \leq_{lcc} t$. In the last case $\lambda x.s' \preccurlyeq_{lcc} (c\ s_1 \ldots s_n)$, we also obtain from the $\preccurlyeq_{lcc}$-definition, that it is valid and from Lemma 4.30, that it implies $\lambda x.s' \leq_{lcc} (c\ s_1 \ldots s_n)$. Other combinations of constructor applications, abstractions and Ω cannot be in $\leq_{lcc}$-relation:

- $(c\ t_1 \ldots t_n) \not\leq_{lcc} \Omega$ and $\lambda x.s \not\leq_{lcc} \Omega$ since the empty context distinguishes them.
- $(c_1\ s_1 \ldots s_n) \not\leq_{lcc} (c_2\ t_1 \ldots t_m)$: Let $C := \texttt{case}_T\ [\cdot]\ (c_1\ x_1 \ldots x_n \to \lambda y.y)$ *alts* where all alternatives in *alts* have Ω as right hand side. Then $C[(c_1\ s_1 \ldots s_n)]{\downarrow_{lcc}}$ but $C[(c_2\ t_1 \ldots t_m)]{\Uparrow_{lcc}}$.
- $(c\ s_1 \ldots s_n) \not\leq_{lcc} (c\ t_1 \ldots t_n)$ if $s_i \not\leq_{lcc} t_i$: Let context D be the witness for $s_i \not\leq_{lcc} t_i$. Then $C = \texttt{case}_T\ [\cdot]\ (c\ x_1 \ldots x_n \to D[x_i])$ distinguishes $(c\ s_1 \ldots s_n)$ and $(c\ t_1 \ldots t_n)$
- $(c\ s_1 \ldots s_n) \not\leq_{lcc} (\lambda x.t)$: The context $\texttt{case}_T\ [\cdot]\ (c\ x_1 \ldots\ x_n \to \lambda y.y)$ *alts* is a witness.

– $\lambda x.s \not\leq_{lcc} \lambda x.t$ if $s \not\leq_{lcc} t$: Let D be the witness for $s \not\leq_{lcc} t$. Then $C = D[([\cdot]\ x)]$ distinguishes $\lambda x.s$ and $\lambda x.t$.
– $\lambda x.s \not\leq_{lcc} (c\ t_1 \ldots t_n)$ if $s \notin cBot$: Since $s \notin cBot$ and $FV(s) \subseteq \{x\}$, there exists a closing substitution $\sigma = \{x \mapsto r\}$ such that $\sigma(s){\downarrow_{lcc}}$. For the context $C = ([\cdot]\ r)$ the expression $C[\lambda x.s]$ converges while $C[(c\ t_1 \ldots t_n)]$ diverges. □

Lemma 4.33. $\leq_{lcc} \quad \subseteq \quad \preccurlyeq^o_{lcc}$.

Proof. The relation $\leq^c_{lcc}$ satisfies the fixpoint condition, *i.e.* $\leq^c_{lcc} \subseteq\ F_{lcc}(\leq^c_{lcc})$, which follows from Corollary 4.31 and Proposition 4.32.

Lemmas 4.30 and 4.33 immediately imply:

Theorem 4.34. $\preccurlyeq^o_{lcc} \quad = \quad \leq_{lcc}$.

4.2.2 Alternative Definitions of Bisimilarity in L_{lcc} We want to analyze the translations between our calculi, and the inherent contextual equivalence. This will require to show that several differently defined relations are all equal to contextual equivalence.

Using Theorem 4.8 we show that in L_{lcc}, behavioral equivalence can also be proved inductively:

Definition 4.35. *The set Q_{lcc} of contexts Q is assumed to consist of the following contexts:*

(i) $([\cdot]\ r)$ *for all closed r,*
(ii) for all types T, constructors c of T, and indices i:
$(\texttt{case}_T\ [\cdot]\ \texttt{of} \ldots (c\ x_1 \ldots x_{\mathrm{ar}(c)} \to x_i) \ldots)$ *where all right hand sides of other* `case`*-alternatives are Ω,*
(iii) for all types T and constructors c of T: $(\texttt{case}_T\ [\cdot]\ \texttt{of} \ldots (c\ x_1 \ldots x_{\mathrm{ar}(c)} \to \texttt{True}) \ldots)$ *where all right hand sides of other* `case`*-alternatives are Ω.*

The relations $\preccurlyeq_{lcc,Q_{lcc}}$, $\leq_{lcc,Q_{lcc}}$ on closed $\mathbb{E}_\lambda$-expressions are defined as in Definition 2.5 and Definition 2.6, respectively.

Lemma 4.36. *The calculus L_{lcc} is convergence-admissible in the sense of Definition 4.5, where the Q-contexts are defined as above.*

Proof. Values in L_{lcc} are L_{lcc}-WHNFs. The contexts Q are reduction contexts in L_{lcc}. Hence every reduction of $Q[s]$ will first evaluate s to v and then evaluate $Q[v]$.

Theorem 4.37. $\preccurlyeq_{lcc} \quad = \quad \preccurlyeq_{lcc,Q_{lcc}} \quad = \quad \leq_{lcc,Q_{lcc}}$ *and* $\preccurlyeq^o_{lcc} = \quad \preccurlyeq^o_{lcc,Q_{lcc}} \quad = \quad \leq^o_{lcc,Q_{lcc}}$

Proof. Theorem 4.8. shows $\preccurlyeq_{lcc,Q_{lcc}} \quad = \quad \leq_{lcc,Q_{lcc}}$ since L_{lcc} is convergence-admissible.

The first equation is proved by showing that the relations satisfy the fixpoint equations of the other one in Definition 4.10 and 2.5, respectively.

– $\preccurlyeq_{lcc} \subseteq F_Q(\preccurlyeq_{lcc})$: Assume $s \preccurlyeq_{lcc} t$ for two closed s, t. If $s{\downarrow_{lcc}}v$, then $t{\downarrow_{lcc}}w$ for values v, w. Since reduction preserves $\simeq_{lcc}$, the fixpoint operator conditions are satisfied if v, w are both abstractions or both constructor applications. If $v = \lambda x.s'$ with $s' \in cBot$ and $w = c(t_1, \ldots, t_n)$, $Q(v)$ diverges for all Q, hence $s\ F_Q(\preccurlyeq_{lcc})\ t$.
– $\preccurlyeq_{lcc,Q_{lcc}} \subseteq F_{lcc}(\preccurlyeq_{lcc,Q_{lcc}})$: Assume $s \preccurlyeq_{lcc,Q_{lcc}}$. Let $s{\downarrow_{lcc}}v$. Then also $t{\downarrow_{lcc}}w$ for some value w. In the cases that v, w are both abstractions or both constructor applications, when using appropriate Q of kind (ii) or (iii), the F_{lcc}-conditions are satisfied. If $v = \lambda x.s'$ and $w = c(t_1, \ldots, t_n)$, we have to show that $s' \in cBot$: this can be done using all Q-contexts of the form $([]\ r)$, since $(w\ r){\Uparrow_{lcc}}$ in any case. □

Definition 4.38. *Let CE_{lcc} be the following set of closed $\mathbb{E}_\lambda$-expressions built from constructors, Ω, and closed abstractions. These can be constructed according to the grammar:*

$$r \in CE_{lcc} ::= \Omega \mid \lambda x.s \mid (c\ r_1 \ldots r_{\mathrm{ar}(c)})$$

where s is any closed $\mathbb{E}_\lambda$-expression.

The set Q_{CE} is defined like the set Q_{lcc} in Definition 4.35, but only expressions r from CE_{lcc} are taken into account in the contexts $([\cdot]\ r)$ in (i).

We summarize several definition alternatives for contextual preorder and applicative simulation for L_{lcc}, where we also include further alternatives. (1) is contextual preorder, (2) the applicative simulation, (3), (4) and (5) are similar to the usual call-by-value variants, where (4) and (5) separate the closing part of contexts, where (5) can be seen as bridging the gap between call-by-need and call-by-name. (7) is the $\mathcal{Q}$-similarity, (8) is a further improved inductive $\mathcal{Q}$-simulation by restricting the set of test arguments for abstractions, and (9) is the co-inductive version of (8).

Theorem 4.39. *In L_{lcc}, all the following relations on open $\mathbb{E}_\lambda$-expressions are identical:*

1. $\leq_{lcc}$.
2. $\preccurlyeq^o_{lcc}$.
3. *The relation $\leq_{lcc,1}$, defined as: $s_1 \leq_{lcc,1} s_2$ iff for all closing contexts C: $C[s_1]{\downarrow_{lcc}} \implies C[s_2]{\downarrow_{lcc}}$.*
4. *The relation $\leq_{lcc,2}$, defined as: $s_1 \leq_{lcc,2} s_2$ iff for all closed contexts C and all closing substitutions σ: $C[\sigma(s_1)]{\downarrow_{lcc}} \implies C[\sigma(s_2)]{\downarrow_{lcc}}$.*
5. *The relation $\leq_{lcc,3}$, defined as: $s_1 \leq_{lcc,3} s_2$ iff for all multi-contexts $M[\cdot,\ldots,\cdot]$ and all substitutions σ: $M[\sigma(s_1),\ldots,\sigma(s_1)]{\downarrow_{lcc}} \implies M[\sigma(s_2),\ldots,\sigma(s_2)]{\downarrow_{lcc}}$.*
6. *The relation $\leq_{lcc,4}$, defined as: $s_1 \leq_{lcc,4} s_2$ iff for all contexts $C[\cdot]$ and all substitutions σ: $C[\sigma(s_1)]{\downarrow_{lcc}} \implies C[\sigma(s_2)]{\downarrow_{lcc}}$.*
7. $\leq^o_{lcc,Q_{lcc}}$
8. *The relation $\leq^o_{lcc,Q_{CE}}$ where $\leq_{lcc,Q_{CE}}$ is as defined on closed $\mathbb{E}_\lambda$-expressions in Definition 2.6 for the set Q_{CE} in Definition 4.38.*
9. *The relation $\preccurlyeq^o_{lcc,Q_{CE}}$ as defined on closed $\mathbb{E}_\lambda$-expressions in Definition 2.5 for the set Q_{CE} in Definition 4.38.*

Proof. – (1) $\iff$ (2) $\iff$ (7): This is Theorem 4.34 and Theorem 4.37

– (1) $\iff$ (3): The "$\Rightarrow$"-direction is obvious. For the other direction let $s_1 \leq_{lcc,1} s_2$ and let C be a context such that $\emptyset \neq FV(C[s_1]) \cup FV(C[s_2]) = \{x_1,\ldots,x_n\}$ and let $C[s_1]{\downarrow_{lcc}}$, *i.e.* $C[s_1] \xrightarrow{lcc,*} v$ where v is an abstraction or a constructor application. Let $C' = (\lambda x_1,\ldots,x_n.C)\ \underbrace{\Omega\ldots\Omega}_{n\text{-times}}$. Then $C'[s_i] \xrightarrow{lcc,\text{nbeta},*} s'_i = C[s_i][\Omega/x_1,\ldots,\Omega/x_n]$ for $i = 1,2$. It is easy to verify that the reduction for $C[s_1]$ can also be performed for s'_i, since no reduction in the sequence $C[s_1] \xrightarrow{lcc,*} v$ can be of the form $R[x_i]$ with R being a reduction context. Thus $C'[s_i]{\downarrow_{lcc}}$. Since $C'[s_i]$ must be closed for $i = 1,2$, the precondition implies $C'[s_2]{\downarrow_{lcc}}$ and also $s'_2{\downarrow_{lcc}}$. W.l.o.g. let $s'_2 \xrightarrow{lcc,*} v'$ where v' is an L_{lcc}-WHNF. It is easy to verify that no term in this sequence can be of the form $R[\Omega]$, where R is a reduction context, since otherwise the reduction would not terminate (since $R[\Omega] \xrightarrow{lcc,+} R[\Omega]$). This implies that we can replace the Ω-expression by the free variables, *i.e.* that $C[s_2]{\downarrow_{lcc}}$. Note that this also shows by the previous items (and Corollary 4.31) that (nbeta) is correct for $\sim_{lcc}$.

– (1) $\iff$ (4): This follows from Corollary 4.31 since closing substitutions can be simulated by a context with subsequent (nbeta)-reduction. This also implies that (nbeta) is correct for $\sim_{lcc,2}$ and by the previous item it also correct for $\sim_{lcc,1}$ (where $\sim_{lcc_i} = \leq_{lcc,i} \cap \geq_{lcc,i}$).

– (6) $\iff$ (1) The direction "$\implies$" is trivial. For the other direction let $s_1 \leq_{lcc} s_2$ and let C be a context, σ be a substitution, such that $C[\sigma(s_1)]{\downarrow_{lcc}}$. Let $\sigma = \{x_1 \to t_1,\ldots x_n \to t_n\}$ and let $C' = C[(\lambda x_1,\ldots,x_n.[\cdot])\ t_1\ \ldots\ t_n]$. Then $C'[s_1] \xrightarrow{\text{nbeta},n} C[\sigma(s_1)]$. Since (nbeta)-reduction is correct for $\sim_{lcc}$, we have $C'[s_1]{\downarrow_{lcc}}$. Applying $s_1 \leq_{lcc} s_2$ yields $C'[s_2]{\downarrow_{lcc}}$. Since $C'[s_2] \xrightarrow{\text{nbeta},n} C[\sigma(s_2)]$ and (nbeta) is correct for $\sim_{lcc}$, we have $C[\sigma(s_2)]{\downarrow_{lcc}}$.

– (5) $\iff$ (6): Obviously, $s_1 \leq_{lcc,3} s_2 \implies s_1 \leq_{lcc,4} s_2$. We show the other direction by induction on n – the number of holes in M – that for all $\mathbb{E}_\lambda$-expressions s_1, s_2: $s_1 \leq_{lcc,4} s_2$ implies $M[\sigma(s_1),\ldots\sigma(s_1)]{\downarrow_{lcc}} \implies M[\sigma(s_2),\ldots\sigma(s_2)]{\downarrow_{lcc}}$.
The base cases for $n = 0,1$ are obvious. For the induction step assume that M has $n > 1$ holes. Let $M' = M[\sigma(s_1),\cdot_2,\ldots,\cdot_n]$ and $M'' = M[\sigma(s_2),\cdot_2,\ldots,\cdot_n]$ Then obviously $M'[\sigma(s_1),\ldots,\sigma(s_1)] = M[\sigma(s_1),\ldots,\sigma(s_1)]$ and thus $M'[\sigma(s_1),\ldots,\sigma(s_1)]{\downarrow_{lcc}}$. For $C = M[\cdot_1,\sigma(s_1),\ldots,\sigma(s_1)]$ we have $C[\sigma(s_1)] = M'[\sigma(s_1),\ldots,\sigma(s_1)]$ and $C[\sigma(s_2)] = M''[\sigma(s_1),\ldots,\sigma(s_1)]$ Since $C[\sigma(s_1)]{\downarrow_{lcc}}$, the relation $s_1 \leq_{lcc,4} s_2$ implies that $C[\sigma(s_2)]{\downarrow_{lcc}}$ and thus $M''[\sigma(s_1),\ldots,\sigma(s_1)]{\downarrow_{lcc}}$. Now the induction hypothesis shows that $M''[\sigma(s_2),\ldots,\sigma(s_2)]{\downarrow_{lcc}}$, since the number of holes of M'' is strictly smaller than n. Since $M''[\sigma(s_2),\ldots,\sigma(s_2)] = M[\sigma(s_2),\ldots,\sigma(s_2)]$ we have $M[\sigma(s_2),\ldots,\sigma(s_2)]{\downarrow_{lcc}}$.

– (7) $\iff$ (8): The direction (7) $\implies$ (8) is trivial.
For the other direction we show that $\leq^o_{lcc,Q_{CE}} \subseteq \leq^o_{lcc,Q_{lcc}}$ by showing that $\leq_{lcc,Q_{CE}} \subseteq \leq_{lcc,Q_{lcc}}$. Let s_1, s_2 be closed expressions with $s_1 \leq_{lcc,Q_{CE}} s_2$ and let $Q_1[\ldots Q_n[s_1]\ldots]\downarrow_{lcc}$ for $Q_i \in Q_{lcc}$. Let m be the number of normal-order-reductions of $Q_1[\ldots Q_n[s_1]\ldots]$ to an L_{lcc}-WHNF. Since the reduction rules are correct w.r.t. $\sim_{lcc}$, for every subexpression r of the contexts Q_i, there is some r' with $r' \leq_{lcc} r$, where $r' \in Q_{CE}$, which is derived from r by (top-down)-reduction, which may also be non-normal order, i.e. $r \xrightarrow{lcc,*} r_{m+1}$ where r_{m+1} has reducible subexpressions (not in an abstraction) only at depth at least $m+1$. All those deep subexpressions are then replaced by Ω, and this construction results in r'. By construction, $r' \leq_{lcc} r$. We do this for all the contexts Q_i, and obtain thus contexts Q'_i. The construction using the depth m shows that $(Q'_1[\ldots[Q'_n[s_1]]])\downarrow_{lcc}$, since the normal-order reduction does not use subexpressions at depth greater than m in those r'. By assumption, this implies $(Q'_1[\ldots[Q'_n[s_2]]])\downarrow_{lcc}$, and since $(Q'_1[\ldots[Q'_n[s_2]]]) \leq_{lcc} (Q'_1[\ldots[Q'_n[s_2]]])$, this also implies $(Q_1[\ldots[Q_n[s_2]]])$.
– (8) $\iff$ (9): This follows for the relations on closed expressions by Theorem 4.8, since the deterministic calculus (see Def. 2.1) for L_{lcc} with Q_{CE} as defined above is convergence-admissible. It also holds for the extensions to open expressions, since the construction for the open extension is identical for both relations. □

Also the following can easily be derived from Theorem 4.39 and Corollary 4.31.

Proposition 4.40. *For open $\mathbb{E}_\lambda$-expressions s_1, s_2, where all free variables of s_1, s_2 are in $\{x_1, \ldots, x_n\}$: $s_1 \leq_{lcc} s_2 \iff \lambda x_1, \ldots x_n.s_1 \leq_{lcc} \lambda x_1, \ldots x_n.s_2$*

Proposition 4.41. *Given any two closed $\mathbb{E}_\lambda$-expressions s_1, s_2: $s_1 \leq_{lcc} s_2$ iff the following conditions hold:*

- *If $s_1\downarrow_{lcc}\lambda x.s'_1$, then either (i) $s_2\downarrow_{lcc}\lambda x.s'_2$, and for all closed $r \in CE_{lcc}$: $s_1\ r \leq_{lcc} s_2\ r$, or (ii) $s_2\downarrow_{lcc}(c\ s''_1 \ldots s''_n)$ and $s'_1 \in cBot$.*
- *if $s_1\downarrow_{lcc}(c\ s'_1 \ldots s'_n)$, then $s_2\downarrow_{lcc}(c\ s''_1 \ldots s''_n)$, and for all $i : s'_i \leq_{lcc} s''_i$*

Proof. The if-direction follows from the congruence property of $\leq_{lcc}$ and the correctness of reductions. The only-if direction follows from Theorem 4.39.

This immediately implies

Proposition 4.42. *Given any two closed $\mathbb{E}_\lambda$-expressions s_1, s_2.*

- *If s_1, s_2 are abstractions, then $s_1 \leq_{lcc} s_2$ iff for all closed $r \in CE_{lcc}$: $s_1\ r \leq_{lcc} s_2\ r$*
- *If $s_1 = (c\ t_1 \ldots t_n)$ and $s_2 = (c'\ t'_1 \ldots t'_m)$ are constructor expressions, then $s_1 \leq_{lcc} s_2$ iff $c = c'$, $n = m$ and for all $i : t_i \leq_{lcc} t'_i$*

We finally consider a more relaxed notion of similarity which allows to use known contextual equivalences as intermediate steps when proving similarity of expressions:

Definition 4.43 (Similarity upto $\sim_{lcc}$). *Let $\preccurlyeq_{lcc,\sim}$ be the greatest fixpoint of the following operator $F_{lcc,\sim}$ on closed L_{lcc}-expressions:*

We define an operator $F_{lcc,\sim}$ on binary relations η on closed L_{lcc}-expressions:
$s\ F_{lcc,\sim}(\eta)\ t$ iff the following holds:

1. *If $s \sim_{lcc} \lambda x.s'$ then there are two possibilities: (i) if $t \sim_{lcc} (c\ t_1 \ldots t_n)$ then $s' \in cBot$, or (ii) if $t \sim_{lcc} \lambda x.t'$ then for all closed $r : ((\lambda x.s')\ r)\ \eta\ ((\lambda x.t')\ r)$;*
2. *If $s \sim_{lcc} (c\ s_1 \ldots s_n)$ then $t \sim_{lcc} (c\ t_1 \ldots t_n)$ and $s_i\ \eta\ t_i$ for all i.*

Obviously, we have $s \preccurlyeq_{lcc,\sim} t$ iff one of the three cases holds: (i) $s \sim_{lcc} \lambda x.s'$, $t \sim_{lcc} \lambda x.t'$, and $(\lambda x.s')\ r \preccurlyeq_{lcc,\sim} (\lambda x.t')\ r$ for all closed r; (ii) $s \sim_{lcc} \lambda x.s'$, $t \sim_{lcc} (c\ t_1 \ldots t_n)$, and $s' \in cBot$, or (iii) $s \sim_{lcc} (c\ s_1 \ldots s_n)$, $t \sim_{lcc} (c\ t_1 \ldots t_n)$, and $s_i \preccurlyeq_{lcc,\sim} t_i$ for all i.

Proposition 4.44. *$\preccurlyeq_{lcc,\sim}\ =\ \preccurlyeq_{lcc}\ =\ \leq^c_{lcc}$, and $\preccurlyeq^o_{lcc,\sim}\ =\ \preccurlyeq^o_{lcc}\ =\ \leq_{lcc}$.*

Proof. We show the first equation via the fixpoint equations. (i) We prove that the relation $\preccurlyeq_{lcc,\sim}$ satisfies the fixpoint equation for $\preccurlyeq_{lcc}$: Let $s \preccurlyeq_{lcc,\sim} t$, where s, t are closed. If $s{\downarrow_{lcc}}(c\, s_1 \ldots s_n)$, then also $s \sim_{lcc} (c\, s_1 \ldots s_n)$ which clearly implies $t{\downarrow_{lcc}}(c\, t_1 \ldots t_n)$, and also $t \sim_{lcc} (c\, t_1 \ldots t_n)$. The relation $\preccurlyeq_{lcc,\sim}$ is a fixpoint of $F_{lcc,\sim}(\eta)$, hence $s_i \preccurlyeq_{lcc,\sim} t_i$ for all i.

If $s{\downarrow_{lcc}}\lambda x.s'$ and $t{\downarrow_{lcc}}\lambda x.t'$ then similar arguments show $((\lambda x.s')\ r) \ \preccurlyeq_{lcc,\sim}\ ((\lambda x.t')\ r)$ for all r. If $s{\downarrow_{lcc}}\lambda x.s'$ and $t{\downarrow_{lcc}}(c\ t_1 \ldots t_n)$, then $s \sim_{lcc} \lambda x.s'$ and $t \sim_{lcc} (c\ t_1 \ldots t_n)$. Again the fixpoint property of $\preccurlyeq_{lcc,\sim}$ shows $s' \in cBot$.

(ii) We prove that the relation $\preccurlyeq_{lcc}$ satisfies the fixpoint equation for $F_{lcc,\sim}$: Let $s \preccurlyeq_{lcc} t$ for closed s, t. We know that this is the same as $s \leq_{lcc} t$. If $s \sim_{lcc} (c\ s_1 \ldots s_n)$, then clearly $s{\downarrow_{lcc}}(c\ s'_1 \ldots s'_n)$ where $(c\ s_1 \ldots s_n) \sim_{lcc} (c\ s'_1 \ldots s'_n)$. Since in this case $t \sim_{lcc} (c\ t_1 \ldots t_n)$ and thus $t{\downarrow_{lcc}}(c\ t'_1 \ldots t'_n)$ where $t \sim_{lcc} (c\, t_1 \ldots t_n) \sim_{lcc} (c\, t'_1 \ldots t'_n)$, and also $s_i \preccurlyeq_{lcc,\sim} t_i$ for all i holds, since reduction is correct. If $s \sim_{lcc} \lambda x.s'$ and $s \sim_{lcc} \lambda x.t'$ then $s{\downarrow_{lcc}}\lambda x.s''$ and $t{\downarrow_{lcc}}\lambda x.t''$ and $((\lambda x.s')\ r) \ \preccurlyeq_{lcc,\sim}\ ((\lambda x.t')\ r)$.

If $s \sim_{lcc} \lambda x.s'$ and $s \sim_{lcc} (c\ t_1 \ldots t_n)$, then for $s{\downarrow_{lcc}}\lambda x.s''$, we have $\lambda x.s' \sim_{lcc} \lambda x.s''$, and since $s \leq_{lcc} t$, the characterization of expressions in Proposition 4.32 shows $s', s'' \in cBot$.

5 The Translation $W : L_{LR} \to L_{name}$

The translation $W : L_{LR} \to L_{name}$ is defined as the identity on expressions and contexts, but the definitions of convergence predicates are changed. In this section we prove that contextual equivalence based on L_{LR}-evaluation and contextual equivalence based on L_{name}-evaluation are equivalent. We use infinite trees to connect both evaluation strategies. In [SS07] a similar result was shown for a lambda calculus without seq, case, and constructors.

5.1 Calculus for Infinite Trees

We define infinite expressions which are intended to be the letrec-unfolding of the $\mathbb{E}_{\mathcal{L}}$-expressions with the extra condition that cyclic variable chains lead to local nontermination represented by `Bot`. We then define the calculus L_{tree} which has infinite expressions and performs reduction on infinite expressions.

Definition 5.1. Infinite expressions $\mathcal{E}_{\mathcal{I}}$ *are defined like expressions* $\mathbb{E}_{\mathcal{L}}$ *without letrec-expressions, adding a constant* `Bot`*, and interpreting the grammar co-inductively,* i.e. *the grammar is as follows*

$$S, T, S_i, T_i \in \mathcal{E}_{\mathcal{I}} ::= x \mid (S_1\ S_2) \mid (\lambda x.S) \mid \texttt{Bot} \\ \mid (c\ S_1 \ldots S_{\mathrm{ar}(c)}) \mid (\texttt{seq}\ S_1\ S_2) \mid (\texttt{case}_T\ S\ \texttt{of}\ alts)$$

In order to distinguish in the following the usual expressions from the infinite ones, we say *tree* or infinite expressions. As meta-symbols we use s, s_i, t, t_i for finite expressions and S, T, S_i, T_i for infinite expressions. The constant `Bot` is without any reduction rule.

In the following definition of a mapping from finite expressions to their infinite images, we sometimes use the explicit binary application operator @ for applications inside the trees (*i.e.* an application in the tree is sometimes written as $(@\ S_1\ S_2)$ instead of $(S_1\ S_2)$), since it is easier to explain, but use the common notation in other places. A *position* is a sequence of positive integers, where the empty position is denoted as ε. We use Dewey notation for positions, *i.e.* the position $i.p$ is the sequence starting with i followed by position p. For an infinite tree S and position p, the notation $S|_p$ means the subtree at position p and $p(S)$ denotes the head symbol of $S|_p$.

Definition 5.2. *The translation $IT : \mathbb{E}_{\mathcal{L}} \to \mathcal{E}_{\mathcal{I}}$ translates an expression $s \in \mathbb{E}_{\mathcal{L}}$ into its infinite tree $IT(s) \in \mathcal{E}_{\mathcal{I}}$. Instead of providing a direct definition of the mapping IT, we provide an algorithm that, given a position p of the infinite tree and a given expression s, computes the label of $IT(s)$ at position p. The computation starts with $s\|_p$ and then proceeds with the rules given in Fig. 9. The first group of rules defines the computed label for the position ε, the second part of the rules describes the general case for positions. If the computation fails (or is undefined), then the position is not valid in the tree $IT(s)$.*

The equivalence of infinite expressions is syntactic modulo α-equal trees.

$C[(s\ t)\|_{\varepsilon}]$	$\mapsto$ @	
$C[(\texttt{case}_T\ \ldots)\|_{\varepsilon}]$	$\mapsto \texttt{case}_T$	
$C[(c\ x_1 \ldots\ x_n \to s)\|_{\varepsilon}]$	$\mapsto (c\ x_1\ \ldots\ x_n)$	for a case-alternative
$C[(\texttt{seq}\ s\ t)\|_{\varepsilon}]$	$\mapsto \texttt{seq}$	
$C[(c\ s_1 \ldots s_n)\|_{\varepsilon}]$	$\mapsto c$	
$C[(\lambda x.s)\|_{\varepsilon}]$	$\mapsto \lambda x$	
$C[x\|_{\varepsilon}]$	$\mapsto x$	if x is a free variable or a lambda-bound variable in $C[x]$

The cases for general positions p:

1.	$C[(\lambda x.s)\|_{1.p}]$	$\mapsto$	$C[\lambda x.(s\|_p)]$
2.	$C[(s\ t)\|_{1.p}]$	$\mapsto$	$C[(s\|_p\ t)]$
3.	$C[(s\ t)\|_{2.p}]$	$\mapsto$	$C[(s\ t\|_p)]$
4.	$C[(\texttt{seq}\ s\ t)\|_{1.p}]$	$\mapsto$	$C[(\texttt{seq}\ s\|_p\ t)]$
5.	$C[(\texttt{seq}\ s\ t)\|_{2.p}]$	$\mapsto$	$C[(\texttt{seq}\ s\ t\|_p)]$
6.	$C[(\texttt{case}_T\ s\ \texttt{of}\ alt_1 \ldots alt_n)\|_{1.p}]$	$\mapsto$	$C[(\texttt{case}_T\ s\|_p\ \texttt{of}\ alt_1 \ldots alt_n)]$
7.	$C[(\texttt{case}_T\ s\ \texttt{of}\ alt_1 \ldots alt_n)\|_{(i+1).p}]$	$\mapsto$	$C[(\texttt{case}_T\ s\ \texttt{of} alt_1 \ldots alt_i\|_p \ldots alt_n)]$
8.	$C[\ldots(c\ x_1\ \ldots\ x_n \to s)\|_{1.p} \ldots]$	$\mapsto$	$C[\ldots(c\ x_1\ \ldots\ x_n \to s\|_p)\ldots]$
9.	$C[(c\ s_1 \ldots s_n)\|_{i.p}]$	$\mapsto$	$C[(c\ s_1 \ldots s_i\|_p \ldots s_n)]$
10.	$C[(\texttt{letrec}\ Env\ \texttt{in}\ s)\|_p]$	$\mapsto$	$C[(\texttt{letrec}\ Env\ \texttt{in}\ s\|_p)]$
11.	$C_1[(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C_2[x\|_p])]$	$\mapsto$	$C_1[(\texttt{letrec}\ x = s\|_p, Env\ \texttt{in}\ C_2[x])]$
12.	$C_1[\texttt{letrec}\ x = s, y = C_2[x\|_p],$ $Env\ \texttt{in}\ t]$	$\mapsto$	$C_1[\texttt{letrec}\ x = s\|_p, y = C_2[x],$ $Env\ \texttt{in}\ t]$
13.	$C_1[(\texttt{letrec}\ x = C_2[x\|_p], Env\ \texttt{in}\ s)]$	$\mapsto$	$C_1[(\texttt{letrec}\ x = C_2[x]\|_p, Env\ \texttt{in}\ s)]$

If the position ε hits the same (let-bound) variable twice, then the result is `Bot`.
(This can only happen by a sequence of rules 11,12,13.)

Fig. 9. Infinite tree construction from positions for fixed s

Example 5.3. The expression $\texttt{letrec}\ x = x, y = (\lambda z.z)\ x\ y\ \texttt{in}\ y$ has the corresponding tree $((\lambda z_1.z_1)\ \texttt{Bot}\ ((\lambda z_2.z_2)\ \texttt{Bot}\ ((\lambda z_3.z_3)\ \texttt{Bot}\ \ldots)))$.

The set $\mathbb{C}_{\mathcal{I}}$ of infinite tree contexts includes any infinite tree where a subtree is replaced by a hole $[\cdot]$. Reduction contexts on trees are defined as follows:

Definition 5.4. *Call-by-name reduction contexts $\mathcal{R}_{tree}$ of L_{tree} are defined as follows, where the grammar is interpreted inductively and $S \in \mathcal{E}_{\mathcal{I}}$:*

$$R, R_i \in \mathcal{R}_{tree} ::= [\cdot] \mid (R\ S) \mid (\texttt{case}\ R\ \texttt{of}\ alts) \mid (\texttt{seq}\ R\ S)$$

Definition 5.5. *An L_{tree}-answer (or an L_{tree}-WHNF) is any infinite $\mathcal{E}_{\mathcal{I}}$-expression S which is an abstraction or constructor application, i.e. $\varepsilon(S) = \lambda x$ or $\varepsilon(S) = c$ for some constructor c. The reduction rules on infinite expressions are allowed in any context and are as follows:*

(betaTr) $((\lambda x.S_1)\ S_2) \to S_1[S_2/x]$
(seqTr) $(\texttt{seq}\ S_1\ S_2) \to S_2$ *if S_1 is an L_{tree}-answer*
(caseTr) $(\texttt{case}_T\ (c\ S_1 \ldots S_n)\ \texttt{of}\ \ldots (c\ x_1 \ldots x_n \to S') \ldots) \to S'[S_1/x_1, \ldots, S_n/x_n]$

If $S = R[S_1]$ for a $\mathcal{R}_{tree}$-context R, and $S_1 \xrightarrow{a} S_2$ for $a \in \{(\text{betaTr}), (\text{caseTr}),$ or $(\text{seqTr})\}$, then we say $S \xrightarrow{tree} S' = R[S_2]$ is a normal order reduction *(tree-reduction) on infinite trees. Here S_1 is the* tree-redex *of the tree-reduction. We also use the convergence predicate $\downarrow_{tree}$ for infinite trees defined as: $S{\downarrow_{tree}}$ iff $S \xrightarrow{tree,*} S'$ and S' is an L_{tree}-WHNF.*

Note that $\xrightarrow{tree,\text{betaTr}}$ and $\xrightarrow{tree,\text{caseTr}}$ only reduce a single redex, but may modify infinitely many positions, since there may be infinitely many positions of a replaced variable x. E.g. a ($tree$,betaTr) of $IT((\lambda x.(\texttt{letrec}\ z = (z\ x)\ \texttt{in}\ z))\ r) = (\lambda x.((\ldots\ (\ldots\ x)\ x)\ x))\ r \to\ ((\ldots\ (\ldots\ r)\ r)\ r))$ replaces the infinite number of occurrences of x by r.

Concluding, the calculus L_{tree} is defined by the tuple $(\mathcal{E}_\mathcal{I}, \mathbb{C}_\mathcal{I}, \xrightarrow{tree}, \mathbb{A}_{tree})$ where $\mathbb{A}_{tree}$ are the L_{tree}-WHNFs.

In the following we use a variant of infinite outside-in developments [Bar84,KKSdV97] as a reduction on trees that may reduce infinitely many redexes in one step. The motivation is that the infinite trees corresponding to finite expressions may require the reduction of infinitely many redexes of the trees for one $\xrightarrow{LR}$- or $\xrightarrow{L_{name}}$-reduction, respectively.

Definition 5.6. *We define an infinite variant of Barendregt's 1-reduction: Let $S \in \mathcal{E}_\mathcal{I}$ be an infinite tree. Let † be a special label, and let M be a set of (perhaps infinitely many) positions of S, which must be redexes w.r.t. the same reduction $a \in \{(\text{betaTr}), (\text{caseTr}),$ or $(\text{seqTr})\}$. Now exactly all positions $m \in M$ of S are labeled with †. By $S \xrightarrow{I,M} S'$ we denote the (perhaps infinite) development top down, defined as follows:*

- *Let $S_0 = S$ and $M_0 = M$.*
- *Iteratively compute M_{i+1} and S_{i+1} from M_i and S_i for $i = 0, 1, 2, \ldots$ as follows:*
  Let d be the length of the shortest position in M_i, and $M_{i,d}$ be the finite set of positions that are the shortest ones in M_i.
  For every $p \in M_{i,d}$ construct an infinite tree T_p from $S_i|_p$ by iterating the following reduction until the root of $S_i|_p$ is not labeled: remove the label from the top of $S_i|_p$, then perform a labeled reduction inheriting all the labels. If this iteration does not terminate, because the root of $S_i|_p$ gets labeled in every step, then the result is $T_p :=$ `Bot` (unlabeled), otherwise a result T_p is computed after finitely many reductions.
  Now construct S_{i+1} by replacing every subtree at a position $p \in M_{i,d}$ in S_i by T_p: for the positions p of S_i that do not have a prefix that is in $M_{i,d}$, we set $p(S_{i+1}) := p(S_i)$ and for $p \in M_{i,d}$ we set $S_i|_p := T_p$.
  Let M_{i+1} be the set of positions in S_{i+1} which carry a label †. The length of the shortest position is now at least $d + 1$. Then iterate again with M_{i+1}, S_{i+1}.
- *S' is defined as the result after (perhaps infinitely many) construction steps $S_1, S_2, \ldots$.*

If the initial set M does not contain a reduction position then we write $S \xrightarrow{I,M,\neg tree} S'$. We write $S \xrightarrow{I,\neg tree} S'$ ($S \xrightarrow{I} S'$, resp.) if there exists a set M such that $S \xrightarrow{I,M,\neg tree} S'$ ($S \xrightarrow{I,M} S'$, resp.).

Example 5.7. We give two examples of standard reduction and $\xrightarrow{I,M}$-reductions.
An $\xrightarrow{LR}$-reduction on expressions corresponds to an $\xrightarrow{I,M}$-reduction on infinite trees and perhaps corresponds to an infinite sequence of infinite *tree*-reductions. Consider `letrec` $y = (\lambda x.y)\ a$ `in` y. The (LR, lbeta)-reduction with a subsequent (LR, llet) reduction results in `letrec` $y = y, x = a$ `in` y. The corresponding infinite tree of `letrec` $y = (\lambda x.y)\ a$ `in` y is $S = (\lambda x_1.((\lambda x_2.((\lambda x_3.((\ldots)\ a))\ a))\ a)$. The $(tree, \text{betaTr})$-reduction-sequence is infinite. let M be the infinite set of positions of all the applications in S, i.e. $M = \{\varepsilon, 1.1, 1.1.1.1, \ldots\}$. Then in the (infinite) development described in Def. 5.6 all intermediate trees have a label at the top, and thus we have $S \xrightarrow{I,M}$ `Bot`. For a set M without ε, the result will be a value tree.

Let the expression be `letrec` y = (`seq True` (`seq` y `False`)) `in` y. Then the $\xrightarrow{LR}$-reduction results in `letrec` y = (`seq` y `False`) `in` y which diverges. The corresponding infinite tree is (`seq True` (`seq` ((`seq True` (`seq` (...) `False`)) `False`))), which has an infinite number of *tree*-reductions, at an infinite number of deeper and deeper positions. Let $M = \{\varepsilon, 1.2, 1.2.1.2, \ldots\}$ be the maximal set. Then $S \xrightarrow{I,M}$ (`seq` (`seq` (`seq`... `False`) `False`) `False`).

There may be S, S' such that $S \xrightarrow{I,M} S'$ as well as $S \xrightarrow{I,M'} S'$ for some sets M, M' where M contains a reduction position, but M' does not contain a reduction position. For example $S = (\lambda x_1.x_1)\ ((\lambda x_2.x_2)\ ((\lambda x_3.x_3) \ldots))$, where a single (betaTr)-reduction at the top reproduces S, as well as a single (betaTr)-reduction of the argument.

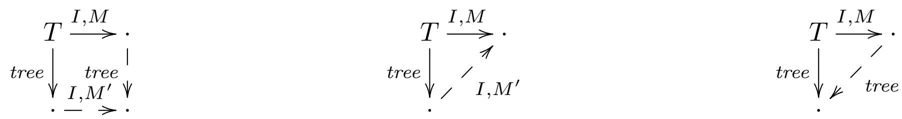
5.2 Standardization of Tree Reduction

Before considering the concrete calculi L_{LR} and L_{name} and their correspondence to the calculus with infinite trees, we show that for an arbitrary reduction sequence on infinite trees resulting in an answer we can construct a *tree*-reduction sequence that results in an L_{tree}-WHNF.

Lemma 5.8. *Let T be an infinite expression. If $T \xrightarrow{I,M,\neg tree} T'$ for some M, where T' is an answer, then T is also an answer.*

Proof. This follows since an answer cannot be generated by $\xrightarrow{I,M,\neg tree}$-reductions, since neither abstractions nor constructor expressions can be generated at the top position.

Lemma 5.9. *Any overlapping between a $\xrightarrow{tree}$-reduction and a $\xrightarrow{I,M}$-reduction can be closed as follows. The trivial case that both given reductions are identical is omitted.*

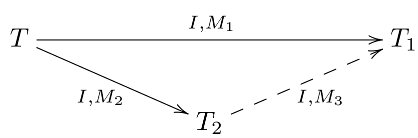


Proof. This follows by checking the overlaps of $\xrightarrow{I}$ with *tree*-reductions. The third diagram applies if the positions of M are removed by the *tree*-reduction. The second diagram applies if the *tree*-redex is included in M and the first diagram is applicable in all other cases.

Lemma 5.10. *Let T be an infinite tree such that there is a tree-reduction sequence of length n to a WHNF T', and let S be an infinite tree with $T \xrightarrow{I,M} S$. Then S has a tree-reduction sequence of length $\leq n$ to a WHNF T''.*

Proof. This follows from Lemma 5.9 by induction on n.

Lemma 5.11. *Consider two reductions $\xrightarrow{I,M_1}$ and $\xrightarrow{I,M_2}$ of the same type (*betaTr*), (*caseTr*) or (*seqTr*). For all trees T, T_1, T_2: if $T \xrightarrow{I,M_1} T_1$, and $T \xrightarrow{I,M_2} T_2$, and $M_2 \subseteq M_1$, then there is a set M_3 of positions, such that $T_2 \xrightarrow{I,M_3} T_1$.*



Proof. The argument is that the set M_3 is computed by labeling the positions in T using M_1, and then by performing the infinite development using the set of redexes M_2, where we assume that the M_1-labels are inherited. The set of positions of marked redexes in T_2 that remain and are not reduced by $T_1 \xrightarrow{I,M_2} T_2$ is exactly the set M_3. □

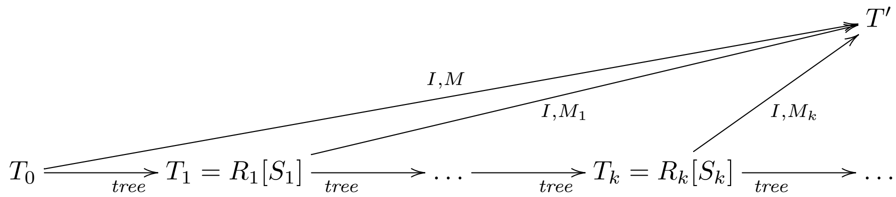
Consider a reduction $T \xrightarrow{I,M} T'$ of type (betaTr), (caseTr) or (seqTr). This reduction may include a redex of a normal order *tree*-reduction. Then the reduction can be split into $T \xrightarrow{tree} T_1 \xrightarrow{I} T'$. This split can be iterated, as long as the remaining $T_1 \xrightarrow{I} T'$ has a *tree*-redex. Nevertheless it may happen that this split does not terminate.

We consider this non-terminating case, *i.e.* let $T_0 \xrightarrow{I,M} T'$ and we can assume that there exist infinitely many $T_1, T_2, \ldots$ and $M_1, M_2, \ldots$, such that for any k: $T_0 \xrightarrow{tree,k} T_k$ and $T_k \xrightarrow{I,M_k} T'$. By induction we can show for every $k \geq 1$: $T_{k-1} = R_{k-1}[S_{k-1}] \to R_{k-1}[S_k] = T_k$ for a reduction context R_k and where S_{k-1} is the redex and S_k is the contractum of $T_{k-1} \to T_k$ and the normal order *tree*-redex of M_k labels a subterm of S_k. This holds, since the infinite development for $T \xrightarrow{I,M} T'$ is performed top down.

This implies that the infinite *tree*-reduction goes deeper and deeper along one path of the tree, or at some point all remaining *tree*-reductions are performed at the same position.

Lemma 5.12. *Let $T \xrightarrow{I,M} T'$ such that $T'{\downarrow_{tree}}$ and M labels the normal order redex of T. Then there exists T'' and M' such that $T \xrightarrow{tree,*} T'' \xrightarrow{M',\neg tree} T'$.*

Proof. Let $T = T_0 \xrightarrow{tree,k} T_k$, $T_k \xrightarrow{I,M_k} T'$ where M_k labels a normal order redex.



We have $T_k = R_k[S_k]$ where R_k is a reduction context, and M_k labels the hole of R_k, which is the normal order redex. The normal order reduction is $T_k = R_k[S_k] \xrightarrow{tree} R_k[S'_k] =: T_{k+1}$. Let p_k be the path of the hole of R_k, together with the constructors and symbols (`case`, `seq`, constructors and @) on the path. Also let $M_k = M_{k,1} \uplus M_{k,2}$, (where $\uplus$ means disjoint union) where the labels of $M_{k,1}$ are in R_k, and the labels $M_{k,2}$ are in S_k. Lemma 5.11, the structure of the expressions and the properties of the infinite top down developments show that the normal order redex can only stay or descend, *i.e.* $h > k$ implies that p_k is a prefix of p_h.

Also, we have $R_k[S_k] \xrightarrow{I,M_k} R'_k[S']$, where $R_k[\cdot] \xrightarrow{M_{k,1}} R'_k[\cdot]$, and $S_k \xrightarrow{I} S'$.
There are three cases:

- The normal order reduction of T_0 halts, *i.e.*, there is a maximal k. Then obviously $T \xrightarrow{tree,*} T_k \xrightarrow{M_k,\neg tree} T'$.
- There is some k, such that $R_k = R_h$ for all $h \geq k$. In this case, $T' = R'_k[s']$. The infinite development $T_0 \xrightarrow{I,M} T'$ will reduce infinitely often at the position of the hole, hence it will plug a `Bot` at position p_k of T', and so $T' = R'_k[\texttt{Bot}]$. But then T' cannot converge, and so this case is not possible.
- The positions p_k of the reduction contexts R_k will grow indefinitely. Then there is an infinite path (together with the constructs and symbols) p such that p_k is a prefix of p for every k. Moreover, p is a position of T'. The sets $M_{k,1}$ are an infinite ascending set *w.r.t.* $\subseteq$, hence there is a limit tree T_∞ with $T \xrightarrow{tree,\infty} T_\infty$, which is exactly the limit of the contexts R_k for $k \to \infty$. There is a reduction $T_\infty \xrightarrow{I,M'} T'$ which is exactly $M' = \bigcup_k M_{k,1}$. Hence T' has the path p, and we see that the tree T' cannot have a normal order redex, since the search for such a redex goes along p and thus does not terminate. This is a contradiction, and hence this case is not possible. ◻

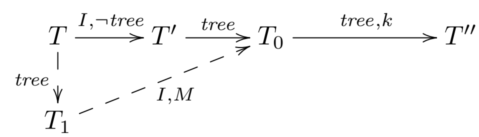
Lemma 5.13. *Let* $T \xrightarrow{I,M,\neg tree} T_1 \xrightarrow{tree} T'$. *Then the reduction can be commuted to* $T \xrightarrow{tree} T_3 \xrightarrow{I,M'} T'$ *for some* M'.

Proof. This follows since the $\xrightarrow{I,M,\neg tree}$-reduction cannot generate a new normal order *tree*-redex. Hence, the normal order redex of T_1 also exists in T. The set M' can be found by labeling T with M, then performing the *tree*-reduction where all labels of M are kept and inherited by the reduction, except for those positions which are removed by the reduction.

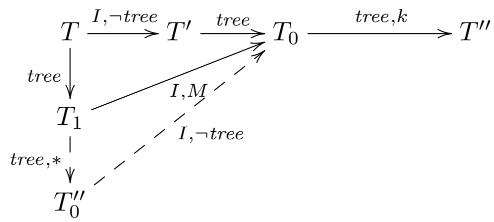
Lemma 5.14. *Let* $T \xrightarrow{I,\neg tree} T'$ *and* $T'{\downarrow_{tree}}$. *Then* $T{\downarrow_{tree}}$.

Proof. We show by induction on k that whenever $T \xrightarrow{I,\neg tree} T' \xrightarrow{tree,k} T''$ where T'' is an L_{tree}-WHNF, then $T{\downarrow_{tree}}$. The base case is $k = 0$ and it holds by Lemma 5.8. For the induction step let $T \xrightarrow{I,\neg tree} T' \xrightarrow{tree} T_0 \xrightarrow{tree,k} T''$. We apply Lemma 5.13 to $T \xrightarrow{I,\neg tree} T' \xrightarrow{tree} T_0$ and thus have $T \xrightarrow{tree} T_1 \xrightarrow{I,M} T_0 \xrightarrow{tree,k} T''$ for some M.

This situation can be depicted by the following diagram where the dashed reductions follow by Lemma 5.13:



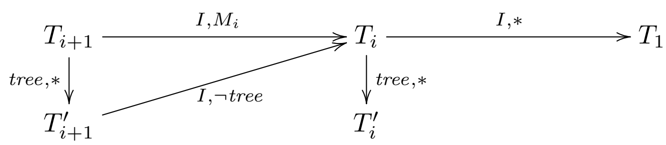
If M does not contain a normal order redex, then the induction hypothesis shows that $T_1{\downarrow_{tree}}$ and thus also $T{\downarrow_{tree}}$. Now assume that M contains a normal order redex. Then we apply Lemma 5.12 to $T_1 \xrightarrow{I,M} T_0$ (note that $T_0{\downarrow_{tree}}$ and hence the lemma is applicable). This shows that $T_1 \xrightarrow{tree,*} T''_0 \xrightarrow{I,\neg tree} T_0$:



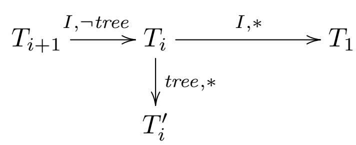
Now we can apply the induction hypothesis to $T_0'' \xrightarrow{\neg tree} T_0 \xrightarrow{tree,k} T''$ and have $T_0''{\downarrow_{tree}}$ which also shows $T{\downarrow_{tree}}$.

Proposition 5.15 (Standardization). *Let $T_1, \ldots, T_k$ be infinite trees such that $T_k \xrightarrow{I,M_{k-1}} T_{k-1} \xrightarrow{I,M_{k-2}} T_{k-2} \ldots \xrightarrow{I,M_1} T_1$, where T_1 is an L_{tree}-WHNF. Then $T_k{\downarrow_{tree}}$*

Proof. We use induction on k. If $k = 1$ then the claim obviously holds since $T_k = T_1$ is already an L_{tree}-WHNF. For the induction step assume that $T_i \xrightarrow{I,M_{i-1}} T_{i-1} \ \ldots \xrightarrow{I,M_1} T_1$ and $T_i{\downarrow_{tree}}$. Let $T_{i+1} \xrightarrow{I,M_i} T_i$. If M_i contains a normal order redex, then we apply Lemma 5.12 and have the following situation



where T'_i is an L_{tree}-WHNF. We apply Lemma 5.14 to $T'_{i+1} \xrightarrow{I,\neg tree} T_i \xrightarrow{tree,*} T'_i$ which shows that $T'_{i+1}{\downarrow_{tree}}$ and thus also $T_{i+1}{\downarrow_{tree}}$.
If M_i contains no normal order redex, we have



where T'_i is an L_{tree}-WHNF. We apply Lemma 5.14 to $T_{i+1} \xrightarrow{I,\neg tree} T_i \xrightarrow{tree,*} T'_i$ and have $T_{i+1}{\downarrow_{tree}}$.

5.3 Equivalence of Tree-Convergence and L_{LR}-Convergence

In this section we will show that L_{LR}-convergence for finite expressions $s \in \mathbb{E}_{\mathcal{L}}$ coincides with convergence for the corresponding infinite tree $IT(s)$.

Lemma 5.16. *Let $s_1, s_2 \in \mathbb{E}_{\mathcal{L}}$ be finite expressions and $s_1 \to s_2$ by a rule (cp), or (lll). Then $IT(s_1) = IT(s_2)$.*

Lemma 5.17. *Let s be a finite expression. If s is an L_{LR}-WHNF then $IT(s)$ is an answer. If $IT(s)$ is an answer, then $s{\downarrow_{LR}}$.*

Proof. If s is an L_{LR}-WHNF, then obviously, $IT(s)$ is a answer. If $IT(s)$ is an answer, then the label computation of the infinite tree for the empty position using s, *i.e.* $s\|_\varepsilon$, must be λx or c for some constructor. If we consider all the cases where the label computation for $s\|_\varepsilon$ ends with such a label, we see that s must be of the form $NL[v]$, where v is an L_{LR}-answer and the contexts NL are constructed according to the grammar:

$$\begin{aligned} NL ::= {} & [\cdot] \mid \texttt{letrec}\ Env\ \texttt{in}\ NL \\ & \mid \texttt{letrec}\ x_1 = NL[\cdot], \{x_i = NL[x_{i-1}]\}_{i=2}^{n}, Env\ \texttt{in}\ NL[x_n] \end{aligned}$$

We show by induction that every expression $NL[v]$, where v is a value, can be reduced by normal order (cp)- and (llet)-reductions to a WHNF in L_{LR}. We use the following induction measure μ on $NL[v]$:

$$\begin{array}{ll} \mu(v) & := 0 \\ \mu(\texttt{letrec}\ Env\ \texttt{in}\ NL[v]) & := 1 + \mu(NL[v]) \\ \mu(\texttt{letrec}\ x_1 = NL_1[v], \{x_i = NL_i[x_{i-1}]\}_{i=2}^{n}, Env\ \texttt{in}\ NL_{n+1}[x_n]) := & \\ \quad \mu(NL_1[v]) + \mu(\texttt{letrec}\ x_2 = NL_2[v], \{x_i = NL_i[x_{i-1}]\}_{i=3}^{n}, Env\ \texttt{in}\ NL_{n+1}[x_n]) & \end{array}$$

The base case obviously holds, since v is already an L_{LR}-WHNF. For the induction step assume that $NL[v'] \xrightarrow{LR,\mathrm{cp}\vee\mathrm{llet},*} t$, where t is an L_{LR}-WHNF for every $NL[v']$ with $\mu(NL[v']) < k$. Let NL, and v be fixed, such that $\mu(NL[v]) = k \geq 1$. There are two cases:

- $NL[v] = \texttt{letrec}\ Env\ \texttt{in}\ NL'[v]$. If NL' is the empty context, then $NL[v]$ is an L_{LR}-WHNF. Otherwise $NL'[v]$ is a `letrec`-expression. Thus we can apply an $(LR, \text{(llet-in)})$-reduction to $NL[v]$, where the measure μ is decreased by one. The induction hypothesis shows the claim.
- $NL[v] = \texttt{letrec}\ x_1 = NL_1[v], \{x_i = NL_i[x_{i-1}]\}_{i=2}^{n}, Env\ \texttt{in}\ NL_{n+1}[x_n]$. If $NL_{n+1}[x_n]$ is a `letrec`-expression, then we can apply an $(LR, \text{llet-in})$-reduction to $NL[v]$ and the measure μ is decreased by 1. If NL_{n+1} is the empty context, and there is some i such that NL_i is not the empty context, then we can choose the largest number i and apply an $(LR, \text{llet-e})$-reduction to $NL[v]$. Then the measure μ is strictly decreased and we can use the induction hypothesis. If all the contexts NL_i for $i = 1, \ldots, n+1$ are empty contexts, then either $NL[v]$ is an L_{LR}-WHNF (if v is a constructor application) or we can apply an (LR, cp) reduction to obtain an L_{LR}-WHNF. □

Lemma 5.18. *Let $s \in \mathbb{E}_{\mathcal{L}}$ such that $s \xrightarrow{LR,a} t$. If the reduction a is* (cp) *or* (lll) *then $IT(s) = IT(t)$. If the reduction a is* (lbeta), (case-c), (case-in), (case-e) *or* (seq-c), (seq-in), (seq-c) *then $IT(s) \xrightarrow{I,M,a'} IT(t)$ for some M, where a' is* (betaTr), (caseTr), *or* (seqTr), *respectively, and the set M contains normal order redexes.*

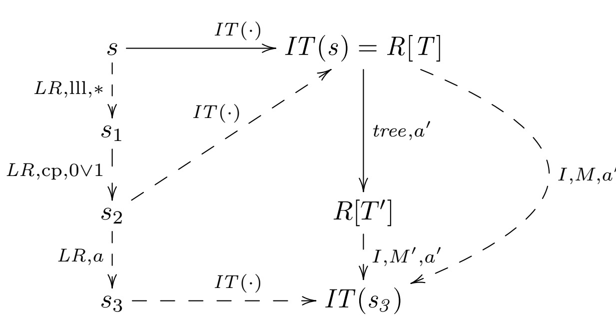
Proof. Only the latter needs a justification. Therefore, we label every redex in $IT(s)$ that is derived from the redex $s \xrightarrow{LR} t$ by $IT(.)$. This results in the set M for $IT(s)$. There will be at least one position in M that is a normal order redex of $IT(s)$.

Proposition 5.19. *Let s be a finite expression such that $s{\downarrow}_{LR}$. Then $IT(s){\downarrow}_{tree}$.*

Proof. We assume that $s \xrightarrow{LR,*} t$, where t is a WHNF. Using Lemma 5.18, we see that there is a finite sequence of reductions $IT(s) \xrightarrow{I,*} IT(t)$. Lemma 5.17 shows that $IT(r)$ is an L_{tree}-WHNF. Now Proposition 5.15 shows that $IT(s){\downarrow}_{tree}$.

We now consider the other direction and show that for every expression s: if $IT(s)$ converges, then s converges, too.

Lemma 5.20. *Let $IT(s) = R[T] \xrightarrow{tree,a'} R[T']$ for some reduction context R. Then $s \xrightarrow{LR,\mathrm{lll},*} s_1 \xrightarrow{LR,\mathrm{cp},0\vee 1} s_2 \xrightarrow{LR,a} s_3$ with $R[T'] \xrightarrow{I,M} IT(s_3)$ where $(a', a) \in \{(\text{betaTr}, \text{lbeta}), (\text{caseTr}, \text{case}), (\text{seqTr}, \text{seq})\}$.*



Proof. Let p be the position of the hole of R. We follow the label computation to T along p inside s and show that the redex corresponding to T can be found in s after some (lll) and (cp) reductions. For applications, `seq`-expressions, and `case`-expressions there is a one-to-one correspondence. If the label computation shifts a position into a "deep" `letrec`, *i.e.* $C[(\texttt{letrec}\ Env\ \texttt{in}\ s)\|_p] \mapsto C[(\texttt{letrec}\ Env\ \texttt{in}\ s\|_p)]$ where C is non-empty, then a sequence of normal order (lll)-reduction moves the environment Env to the top of the expression, where perhaps it is joined with a top-level environment of C. Let $s \xrightarrow{LR,\text{lll},*} s'$. Lemma 5.16 shows that $IT(s') = IT(s)$ and the label computation along p for s' requires fewer steps than the computation for s. Hence this construction can be iterated and terminates. This yields a reduction sequence $s \xrightarrow{LR,\text{lll},*} s_1$ such that the label computation along p for s_1 does not shift the label into deep `letrec`s and where $IT(s) = IT(s_1)$ (see Lemma 5.16). Now there are two cases: Either the redex corresponding to T is also a normal order redex of s_1, or s_1 is of the form $\texttt{letrec}\ x_1 = \lambda x.s', x_2 = x_1, \ldots, x_m = x_{m-1}, \ldots R'[x_m] \ldots$. For the latter case an (LR, cp) reduction is necessary before the corresponding reduction rule can be applied. Again Lemma 5.16 assures that the infinite tree remains unchanged. After applying the corresponding reduction rule, *i.e.* $s_2 \xrightarrow{LR,a} s_3$, the normal order reduction may have changed infinitely many positions of $IT(s_3)$, while $R[T] \xrightarrow{tree,a'} R[T']$ does not change all these positions, but nevertheless Lemma 5.18 shows that there is a reduction $R[T] \xrightarrow{I,M,a'} IT(s_3)$, and Lemma 5.11 shows that also $R[T'] \xrightarrow{I,M',a'} IT(s_3)$ for some M'.

Example 5.21. An example for the proof of the last lemma is the expression $s := \texttt{letrec}\ x = (\lambda y.y)\ x\ \texttt{in}\ x$. Then $IT(s) = (\lambda y.y)\ ((\lambda y.y)\ ((\lambda y.y) \ldots))$. The *tree*-reduction for $IT(s)$ is $IT(s) \xrightarrow{tree,\text{betaTr}} IT(s)$. On the other hand the normal order reduction of L_{LR} reduces to $s' := \texttt{letrec}\ x = (\texttt{letrec}\ y = x\ \texttt{in}\ y)\ \texttt{in}\ x$ and $IT(s') = \texttt{Bot}$. To join the reductions we perform an $\xrightarrow{I,M}$-reduction for $IT(s)$ where all redexes are labeled in M, which also results in `Bot`.

Proposition 5.22. *Let s be an expression such that $IT(s)\downarrow_{tree}$. Then $s\downarrow_{LR}$.*

Proof. The precondition $IT(s)\downarrow_{tree}$ implies that there is a *tree*-reduction sequence of $IT(s)$ to an L_{tree}-WHNF. The base case, where no *tree*-reductions are necessary, is treated in Lemma 5.17. In the general case, let $T \xrightarrow{tree,a'} T'$ be a *tree*-reduction. Lemma 5.20 shows that there are expressions s', s'' with $s \xrightarrow{LR,\text{lll},*}\xrightarrow{LR,\text{cp},0\vee 1} s' \xrightarrow{LR,a} s''$, and $T' \xrightarrow{I,M} IT(s'')$. Lemma 5.10 shows that $IT(s'')$ has a normal order *tree*-reduction to a WHNF where the number of *tree*-reductions is strictly smaller than the number of *tree*-reductions of T to a WHNF. Thus we can use induction on this length and obtain a normal order *LR*-reduction of s to a WHNF.

Propositions 5.19 and 5.22 imply the theorem

Theorem 5.23. *Let s be an $\mathbb{E}_{\mathcal{L}}$-expression. Then $s\downarrow_{LR}$ if and only if $IT(s)\downarrow_{tree}$.*

5.4 Equivalence of Infinite Tree Convergence and L_{name}-convergence

It is easy to observe that several reductions of L_{name} do not change the infinite trees *w.r.t.* the translation $IT(\cdot)$:

Lemma 5.24. *Let $s_1, s_2 \in \mathbb{E}_{\mathcal{L}}$. Then $s_1 \xrightarrow{name,a} s_2$ for $a \in \{\text{gcp}, \text{lapp}, \text{lcase}, \text{lseq}\}$ implies $IT(s_1) = IT(s_2)$.*

Lemma 5.25. *For $(a, a') \in \{(\text{beta}, \text{betaTr}), (\text{case}, \text{caseTr}), (\text{seq}, \text{seqTr})\}$ it holds: If $s_1 \xrightarrow{name,a} s_2$ for $s_i \in \mathbb{E}_{\mathcal{L}}$, then $IT(s_1) \xrightarrow{tree,a'} IT(s_2)$.*

Proof. Let $s_1 := R_{name}[s'_1] \xrightarrow{name,a} R_{name}[s'_2] = s_2$ where s'_1 is the redex of the $\xrightarrow{name}$-reduction and R_{name} is an L_{name}-reduction context. First one can observe that the redex s'_1 is mapped by IT to a unique tree position within a tree reduction context in $IT(s_1)$.

We only consider the (beta)-reduction, since for a (case)- or a (seq)-reduction the reasoning is completely analogous. So let us assume that $s'_1 = ((\lambda x.s''_1)\ s''_2)$. Then IT transforms s'_1 into a subtree $\sigma((\lambda x.IT(s''_1))\ IT(s''_2))$ where σ is a substitution replacing variables by infinite trees. The tree reduction replaces $\sigma((\lambda x.IT(s''_1))\ IT(s''_2))$ by $\sigma(IT(s''_1))[\sigma(IT(s''_2))/x]$, hence the lemma holds.

Proposition 5.26. *Let $s \in \mathbb{E}_{\mathcal{L}}$ 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 $IT(L[v])$, where v is an L_{name}-answer is always an L_{tree}-answer. For the induction step we consider the first reduction of s, say $s \xrightarrow{name} s'$. The induction hypothesis shows $IT(s'){\downarrow_{tree}}$. If the reduction $s \xrightarrow{name} s'$ is $(name,\text{gcp})$, $(name,\text{lapp})$, $(name,\text{lcase})$, or $(name,\text{lseq})$, then Lemma 5.24 implies $IT(s){\downarrow_{tree}}$. If $s \xrightarrow{name,a} s'$ for $a \in \{(\text{beta}), (\text{case}), (\text{seq})\}$, then Lemma 5.25 shows $IT(s) \xrightarrow{tree} IT(s')$ and thus $IT(s){\downarrow_{tree}}$.

Now we show the other direction:

Lemma 5.27. *Let $s \in \mathbb{E}_{\mathcal{L}}$ such that $IT(s) = \mathcal{R}[T]$, where $\mathcal{R}$ is a tree reduction context and T is a value or a redex. Then there are expressions s', s'' such that $s \xrightarrow{name,\text{lapp}\vee\text{lcase}\vee\text{lseq}\vee\text{gcp},*} s'$, $IT(s') = IT(s)$, $s' = R[s'']$, $IT(L[s'']) = T$, where $R = L[A[\cdot]]$ is a reduction context for some $\mathcal{L}$-context L and some $\mathcal{A}$-context A, s'' may be an abstraction, a constructor application, or a* beta-, case- *or* seq-*redex iff T is an abstraction, a constructor application, or a* betaTr-, caseTr- *or* seqTr-*redex, 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, a constructor application, an application, or a betaTr-, caseTr- or seqTr-redex 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 $s \xrightarrow{name,\text{lapp}\vee\text{lcase}\vee\text{lseq}\vee\text{gcp},*} s'$, where s' is as claimed in the lemma.
We consider the label-computation for p to explain the induction measure, where we use the numbers of the rules given in Fig. 9. Let q be such that the label computation for p is of the form $(10)^*.q$ and q does not start with (10). 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 $q = (2 \vee 4 \vee 6)^b.q'$. The base case is (a, a): Then the label computation is of the form $(2 \vee 4 \vee 6)^*$ 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 starts with $(10)^*(2 \vee 4 \vee 6)^+(10)$. Then a normal-order (lapp), (lcase), or (lseq) 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 (10) and $(2\vee4\vee6)$ are interchanged, hence the second component of the measure is strictly decreased.
2. The label computation starts with $(10)^*(2 \vee 4 \vee 6)^*(11)$. Then a normal-order (gcp) can be applied to s resulting in s_1. The length q is strictly decreased by 1, and perhaps one (12)-step is changed into a (11)-step. Hence the measure is strictly decreased.

In every case the claim on the structure of the contexts and s' can easily be verified. □

Lemma 5.28. *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 $IT(s) = \mathcal{R}[S]$ where $\mathcal{R}$ is a reduction context, S a tree-redex with $S \xrightarrow{tree} S'$ and $T = \mathcal{R}[S']$. Let p be the position of the hole of $\mathcal{R}$ in $IT(s)$. We apply Lemma 5.27, which implies that there is a reduction $s \xrightarrow{name,*} s'$, such that $IT(s) = IT(s')$ and $s' = R[s'']$ where $R = L[A[\cdot]]$ is a reduction context and $IT(L[s''])$ is a beta-, case-, or seq-redex. It is obvious that $s' = L[A[s'']] \xrightarrow{name,a} t$. Now one can verify that $IT(t) = T$ must hold.

Proposition 5.29. *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 an L_{tree}-answer. For the base case it is easy to verify that if $IT(s)$ is an L_{tree}-answer, then $s \xrightarrow{name,\text{gcp},*} L[v]$ for some $\mathcal{L}$-context L and some L_{name}-value v. Hence we have $s{\downarrow_{name}}$. The induction step follows by repeated application of Lemma 5.28.

Corollary 5.30. *For all $\mathbb{E}_{\mathcal{L}}$-expressions s: $s{\downarrow_{name}}$ if, and only if $IT(s){\downarrow_{tree}}$.*

Theorem 5.31. $\leq_{name} = \leq_{LR}$.

Proof. In Corollary 5.30 we have shown that L_{name}-convergence is equivalent to infinite tree convergence. In Theorem 5.23 we have shown that L_{LR}-convergence is equivalent to infinite tree convergence. Hence, L_{name}-convergence and L_{LR}-convergence are equivalent, which further implies that both contextual preorders and also the contextual equivalences are identical.

Corollary 5.32. *The translation W is convergence equivalent and fully abstract.*

Since W is the identity on expressions, this implies:

Corollary 5.33. *W is an isomorphism according to Definition 2.7.*

A further and novel consequence of our results is that the general copy rule (gcp) is a correct program transformation in L_{LR}:

Proposition 5.34. *The program transformation (gcp) is correct in L_{name} and L_{LR}.*

Proof. Correctness of (gcp) in L_{name} holds, since for $s, t \in \mathbb{E}_{\mathcal{L}}$ with $s \xrightarrow{gcp} t$ and for any context C: $IT(C[s]) = IT(C[t])$. Hence Corollary 5.30 implies that $C[s]{\downarrow_{name}} \iff C[t]{\downarrow_{name}}$ and thus $s \sim_{name} t$. Theorem 5.31 finally also shows $s \sim_{LR} t$.

6 The Translation $N : L_{name} \to L_{lcc}$

We use multi-fixpoint combinators as defined in [Gol05] to translate letrec-expressions $\mathbb{E}_{\mathcal{L}}$ of the calculus L_{name} into equivalent ones without a `letrec`. The translated expressions are $\mathbb{E}_{\lambda}$ and belong to the calculus L_{lcc}.

Definition 6.1. *Given $n \geq 1$, a family of n fixpoint combinators $\boldsymbol{Y}_i^n$ for $i = 1, \dots, n$ can be defined as follows:*

$$\begin{array}{ll}\boldsymbol{Y}_i^n := \lambda f_1, \dots, f_n.(& (\lambda x_1, \dots, x_n.f_i\ (x_1\ x_1\ \dots x_n)\ \dots\ (x_n\ x_1\ \dots x_n)) \\ & (\lambda x_1, \dots, x_n.f_1\ (x_1\ x_1\ \dots x_n)\ \dots\ (x_n\ x_1\ \dots x_n)) \\ & \dots \\ & (\lambda x_1, \dots, x_n.f_n\ (x_1\ x_1\ \dots x_n)\ \dots\ (x_n\ x_1\ \dots x_n)))\end{array}$$

The idea of the translation is to replace (`letrec` $x_1 = s_1, \dots, x_n = s_n$ `in` t) by $t[B_1/x_1, \dots, B_n/x_n]$ where $B_i := \boldsymbol{Y}_i^n\ F_1 \dots F_n$ and $F_i := \lambda x_1, \dots, x_n.s_i$.

In this way the fixpoint combinators implement the generalized fixpoint property: $\boldsymbol{Y}_i^n\ F_1 \dots F_n \sim F_i\ (\boldsymbol{Y}_1^n\ F_1 \dots F_n) \dots (\boldsymbol{Y}_n^n\ F_1 \dots F_n)$. However, our translation uses modified expressions, as shown below.

Consider the expression $(\boldsymbol{Y}_i^n\ F_1\ \dots\ F_n)$. After expanding the notations we obtain the expression $((\lambda f_1, \dots, f_n.(X_i\ X_1\ \dots\ X_n))\ F_1\ \dots\ F_n)$ where $X_i = \lambda x_1 \dots x_n.(f_i\ (x_1\ x_1\ \dots\ x_n)\ \dots\ (x_n\ x_1\ \dots\ x_n))$. If we reduce further then we get:

$$\begin{array}{l}(\lambda f_1, \dots, f_n.(X_i\ X_1\ \dots\ X_n))\ F_1\ \dots\ F_n \xrightarrow{\text{nbeta},*} (X_i'\ X_1'\ \dots\ X_n'), \\ \text{where } X_i' = \lambda x_1 \dots x_n.(F_i\ (x_1\ x_1\ \dots\ x_n) \dots (x_n\ x_1\ \dots x_n))\end{array}$$

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

Definition 6.2. *The translation $N : L_{name} \to L_{lcc}$ is recursively defined as:*

- $N($`letrec` $x_1 = s_1, \dots, x_n = s_n$ `in` $t) =$
  $(\lambda x_1', \dots, x_n'.(\lambda x_1, \dots x_n.N(t))\ U_1 \dots\ U_n)\ X_1' \dots X_n'$
  where $U_i = x_i'\ x_1' \dots x_n'$,
  $X_i' = \lambda x_1 \dots x_n.F_i(x_1\ x_1 \dots x_n) \dots (x_n\ x_1 \dots x_n)$,
  $F_i = \lambda x_1, \dots, x_n.N(s_i)$.
- $N(s\ t) = (N(s)\ N(t))$
- $N($`seq` $s\ t) = ($`seq` $N(s)\ N(t))$

– $N(c\ s_1\ \dots\ s_{\mathrm{ar}(c)}) = (c\ N(s_1) \dots N(s_{\mathrm{ar}(c)}))$
– $N(\lambda x.s) = \lambda x.N(s)$
– $N(\mathtt{case}_T\ s\ \mathtt{of}\ alt_1\ \dots\ alt_{|T|}) = \mathtt{case}_T\ N(s)\ \mathtt{of}\ N(alt_1)\ \dots\ N(alt_{|T|})$
– *for a case-alternative:* $N(c\ x_1\ \dots\ x_{\mathrm{ar}(c)} \to s) = (c\ x_1\ \dots\ x_{\mathrm{ar}(c)} \to N(s))$
– $N(x) = x$.

We extend N *to contexts by treating the hole as a constant,* i.e. $N([\cdot]) = [\cdot]$. *This is consistent, since the hole is not duplicated by the translation.*

6.1 Convergence Equivalence of N

In the following we will also use the context class $\mathcal{B}$, defined as $\mathcal{B} = L[\mathcal{B}] \mid A[\mathcal{B}] \mid [\cdot]$ ($\mathcal{L}$- and $\mathcal{A}$-contexts are defined as before in Sect. 3.2).

The proof of convergence equivalence of the translation N may be performed directly, but it would be complicated due to the additional (nbeta)-reductions required in L_{lcc}. 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 6.3. *The translation* $N' : L_{name} \to L_{lcc}$ *is recursively defined as:*

– $N'(\mathtt{letrec}\ x_1 = s_1, \dots, x_n = s_n\ \mathtt{in}\ t) = \sigma(N'(t))$, *where*

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

– $N'(s\ t) = (N'(s)\ N'(t))$
– $N'(\mathtt{seq}\ s\ t) = (\mathtt{seq}\ N'(s)\ N'(t))$
– $N'(c\ s_1 \dots s_n) = (c\ N'(s_1) \dots N'(s_n))$
– $N'(\lambda x.s) = \lambda x.N'(s)$
– $N'(\mathtt{case}_T\ s\ \mathtt{of}\ alt_1\ \dots\ alt_{|T|}) = \mathtt{case}_T\ N'(s)\ \mathtt{of}\ N'(alt_1)\ \dots\ N'(alt_{|T|})$
– *for a case-alternative:* $N'(c\ x_1\ \dots\ x_{\mathrm{ar}(c)} \to s) = (c\ x_1\ \dots\ x_{\mathrm{ar}(c)} \to N'(s))$
– $N'(x) = x$.

The extension of N' *to contexts is done only for* $\mathcal{B}$*-contexts and requires an extended notion of contexts that are accompanied by an additional substitution,* i.e. *a* $\mathcal{B}$*-context translates into a pair* (D, σ) *acting as a function on expressions. Filling the hole of a context* (D, σ) *by an expression* s *is by definition* $(D, \sigma)(s) = D[\sigma(s)]$. *The translation for* $\mathcal{B}$*-contexts is defined as*

$N'(C) = (C', \sigma)$, *where* 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 combined substitution affecting the hole of* C'.

This translation does not duplicate holes of contexts.

Lemma 6.4. *The translation* N *is equivalent to* N' *on expressions,* i.e. *for all* $\mathbb{E}_{\mathcal{L}}$*-expressions* s *the equivalence* $N(s) \sim_{lcc} N'(s)$ *holds.*

Proof. This follows from the definitions and correctness of (nbeta)-reduction in L_{lcc} by Theorem 4.31.

We first prove that the translation N' is convergence-equivalent. Due to Lemma 6.4 this will also imply that N is convergence-equivalent. All reduction contexts $L[A[\cdot]]$ in L_{name} translate into reduction contexts R_{lcc} in L_{lcc} since removing the case of `letrec` from the definition of a reduction context in L_{name} results in the reduction context definition in L_{lcc}. However, this cannot be reversed, since a translated expression of L_{name} may have a redex in L_{lcc}, but it is not a normal order redex in L_{name} since (lapp), (lseq), or (lcase) reductions must be performed first to shift `letrec`-expressions out of an application, a `seq`-expression, or a `case`-expression. The lemma below gives a more precise characterization of this relation:

Lemma 6.5. *If $L[A[\cdot]]$ is a reduction context in L_{name}, then $N'(L[A[\cdot]]) = R[\sigma(\cdot)]$, where R is a reduction context in L_{lcc} and σ is a substitution.*

If R is a reduction context in L_{lcc}, and $N'(C') = (R, \sigma)$ for some substitution σ and some context C' in L_{name}, then C' is a $\mathcal{B}$-context.

Proof. The first claim can be shown by structural induction on the context $L[A[\cdot]]$. It holds, since applications are translated into applications, `seq`-expressions are translated into `seq`-expressions, `case`-expressions are translated into `case`-expressions, and `letrec`-expressions are translated into substitutions.

The other part can be shown by induction on the number of translation steps. It is easy to observe that the definition of a reduction context in L_{name} does not descend into `letrec`-expressions below applications, `seq`-, and `case`-expressions. For instance, in ((`letrec` *Env* `in` $((\lambda x.s_1)\ s_2)$) s_3) the reduction contexts are $[\cdot]$ and $([\cdot]\ s_3)$ and the redex is (lapp), *i.e.* the reduction context does not reach $((\lambda x.s_1)\ s_2)$. In general, applications, `seq`-, and `case`-expressions in such cases appear in $\mathcal{B}$-contexts, as defined above. By examining the expression definition we observe that these (lapp), (lseq), and/or (lcase)-redexes are the only cases where non-reduction contexts may be translated into reduction contexts.

Lemma 6.6. *Let $N'(s) = t$. Then:*

1. *If s is an abstraction then so is t.*
2. *If $s = (c\ s_1 \dots s_{\mathrm{ar}(c)})$ then $t = (c\ t'_1 \dots t'_{\mathrm{ar}(c)})$.*

Proof. This follows by examining the translation N'.

We will now use reduction diagrams to show the correspondence of L_{name}-reduction and L_{lcc}-reduction *w.r.t.* the translation N'.

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

In this section we analyze how normal order reduction in L_{name} can be transferred into L_{lcc} via N'. We illustrate this by using reduction diagrams. For $s \xrightarrow{name} t$ we analyze how the reduction transfers to $N'(s)$. The cases are on the rule used in $s \xrightarrow{name} t$:

- (beta) Let $s = R[(\lambda x.s_1)\ s_2]$ be an expression in L_{name}, where R is a reduction context. We observe that in L_{name}: $s \xrightarrow{name} t = R[s_1[s_2/x]]$. Let $N'(R[\cdot]) = (R', \sigma)$. Then the translations for s and t are as follows:

$$N'(s) = R'[\sigma(N'((\lambda x.s_1)\ s_2))] = R'[(\lambda x.\sigma(N'(s_1)))\ \sigma(N'(s_2))]$$
$$N'(t) = N'(R[s_1[s_2/x]]) = R'[\sigma(N'(s_1[s_2/x]))] = R'[\sigma(N'(s_1))[\sigma(N'(s_2))/x]]$$

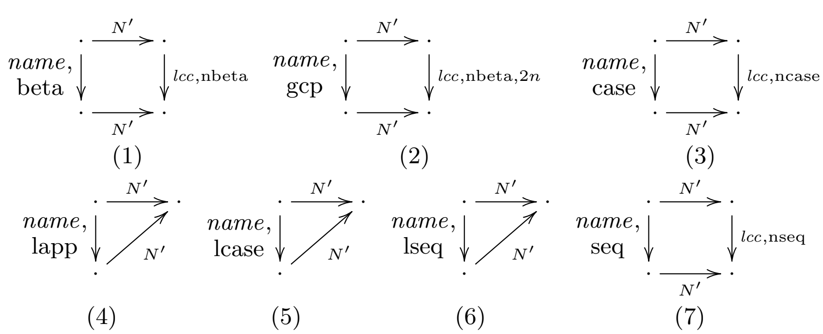


Fig. 10. Diagrams for transferring reductions between L_{name} and L_{lcc}

Since R' is a reduction context in L_{lcc}, this shows $N'(s) \xrightarrow{lcc,\mathrm{nbeta}} N'(t)$. Thus we have the diagram (1) in Figure 10.

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

$$\begin{array}{l} s = L[\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ R[x_1]] \xrightarrow{name,\text{gcp}} \\ t = L[\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ R[s_1]] \end{array}$$

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

$$\begin{aligned} N'(s) &= \sigma_L(\sigma_{Env}(R'[\sigma_R(x_1)])) = \sigma_L(\sigma_{Env}(R'))[\sigma_L(\sigma_{Env}(\sigma_R(x_1)))] \\ &= \sigma_L(\sigma_{Env}(R'))[\sigma_L(\sigma_{Env}(x_1)))] \end{aligned}$$

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 $R''[\sigma_L(\sigma_{Env}(x_1)))] \xrightarrow{lcc,*} R''[\sigma_L(\sigma_{Env}(N'(s_1)))]$ holds.
By definition of the translation N' (Definition 6.3) $\sigma_L(\sigma_{Env}(x_1)) = U_1 = (X'_1 X'_1 \ldots X'_n)$, where $X'_i = \lambda x_1 \ldots x_n.F_i(x_1 x_1 \ldots x_n) \ldots (x_n x_1 \ldots x_n)$, 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

$$\begin{array}{ll} & (\lambda x_1, \ldots, x_n.(F_1(x_1 x_1 \ldots x_n) \ldots (x_n x_1 \ldots x_n))\ X'_1\ \ldots\ X'_n \\ \xrightarrow{\text{nbeta},n} & F_1\ (X'_1 X'_1 \ldots X'_n)\ \ldots\ (X'_n X'_1 \ldots X'_n) \\ = & (\lambda x_1, \ldots, x_n.\sigma_L(N'(s_1))\ (X'_1 X'_1 \ldots X'_n)\ \ldots\ (X'_n X'_1 \ldots X'_n) \\ \xrightarrow{\text{nbeta},n} & \sigma_L(N'(s_1))[U_1/x_1, \ldots, U_n/x_n]. \end{array}$$

Obviously, for all reduction contexts in L_{lcc} we have: $r_1 \xrightarrow{lcc} r_2$ implies $R[r_1] \xrightarrow{lcc} R[r_1]$. Hence $N'(s) \xrightarrow{lcc,\text{nbeta},2n} R''[\sigma_L(N'(s_1))[U_1/x_1, \ldots, U_n/x_n]]$ holds. Since $x_1, \ldots, x_n$ cannot occur free in L, the last expression is the same as $R''[\sigma_L(\sigma_{Env}(N'(s)))]$. Thus we obtain the diagram (2) in Figure 10, where n is the number of bindings in the `letrec`-subexpression where the copied binding is.

– (case) The diagram for this case is marked (3) in Figure 10. The case is analogous to that of (beta): $s = R[\texttt{case}_T\ (c\ \overrightarrow{s_i} \ldots ((c\ \overrightarrow{x_i}) \to r) \ldots] \xrightarrow{name} R[r[s_1/x_1, \ldots, s_{\text{ar}(c)}/x_{\text{ar}(c)}]] = t$. Let $N'(R[\cdot]) = (R', \sigma)$. Then the translations for s and t are as follows:

$$\begin{aligned} N'(s) &= R'[\sigma(N'(\texttt{case}_T\ (c\ s_1 \ldots s_{\text{ar}(c)}) \ldots ((c\ x_1 \ldots x_{\text{ar}(c)}) \to r) \ldots)] \\ &= R'[\texttt{case}_T\ (c\ \sigma(N'(s_1)) \ldots \sigma(N'(s_{\text{ar}(c)})) \ldots ((c\ x_1 \ldots x_{\text{ar}(c)}) \to \sigma(N'(r))) \ldots] \\ N'(t) &= N'(R[r[s_1/x_1, \ldots, s_{\text{ar}(c)}/x_{\text{ar}(c)}]]) \\ &= R'[\sigma(N'(r[s_1/x_1, \ldots, s_{\text{ar}(c)}/x_{\text{ar}(c)}]))] \\ &= R'[\sigma(N'(r))[\sigma(N'(s_1))/x_1, \ldots, \sigma(N'(s_{\text{ar}(c)}))/x_{\text{ar}(c)}]] \end{aligned}$$

Since R' is a reduction context in L_{lcc}, this shows $N'(s) \xrightarrow{lcc} N'(t)$.

– (lapp) Then the reduction is $R[(\texttt{letrec}\ Env\ \texttt{in}\ s_1)\ s_2] \xrightarrow{name} R[(\texttt{letrec}\ Env\ \texttt{in}\ (s_1\ s_2))]$. Since free variables of s_2 do not depend on Env, the translation of s_2 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 $N'(R[(\texttt{letrec}\ Env\ \in\ s_1)\ s_2]) = R'[\sigma_R(\sigma_{Env}(N'(s_1))\ N'(s_2))] = R'[\sigma_R(\sigma_{Env}(N'(s_1\ N'(s_2))] = N'(R[(\texttt{letrec}\ Env\ \texttt{in}\ (s_1\ s_2)])$ and thus the diagram for this case is as the one marked (4) in Figure 10.

– (lcase) The case is analogous to that of (lapp), with the diagram marked as (5) in Figure 10.

– (lseq) The case is analogous to (lapp) and (lcase), with the diagram (6) in Figure 10.

– (seq) $s = R[\texttt{seq}\ v\ s_1] \xrightarrow{name} R[s_1] = t$ where v is an abstraction or a constructor application
Let $N'(R[\cdot]) = (R', \sigma)$. Then the translations for s and t are as follows:

$$\begin{aligned} N'(s) &= R'[\sigma(N'(\texttt{seq}\ v\ s_1))] = R'[\texttt{seq}\ \sigma(N'(v))\ \sigma(N'(s_1))] \\ N'(t) &= R'[\sigma(N'(s_1))] \end{aligned}$$

By Lemma 6.6 $N'(v)$ is a value in L_{lcc} (which cannot be changed by the substitution σ) and thus $N'(s) \xrightarrow{lcc,\text{nseq}} N'(t)$. The diagram for this case is (7) in Figure 10.

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

Lemma 6.7. *Let s be irreducible in L_{name}, but not an L_{name}-WHNF. Then $N'(s)$ is irreducible in L_{lcc} and also not an L_{lcc}-WHNF.*

Proof. Assume that expression s is irreducible in L_{name} but not an L_{name}-WHNF. There are three cases:

1. Expression s is of the form $R[x]$ where x is a free variable in $R[x]$, then let $N'(R) = (R', \sigma)$ and thus $N'(s) = R'[\sigma(x)]$. Since σ only substitutes bound variables, we get $\sigma(x) = x$ and thus $N'(s) = R'[x]$ where x is free in $R'[x]$. Hence $N'(s)$ cannot be an L_{lcc}-WHNF and it is irreducible in L_{lcc}.
2. Expression s is of the form $R[\mathtt{case}_T\ (c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})\ \mathtt{of}\ alts]$, but c does not belong to type T. Let $N'(R) = (R', \sigma)$. Then $N'(s) = R'[\mathtt{case}_T\ (c\ \sigma(N'(s_1))\ \ldots\ \sigma(N'(s_{\mathrm{ar}(c)}))\ \mathtt{of}\ alts']$ which shows that $N'(s)$ is not an L_{lcc}-WHNF and irreducible in L_{lcc}.
3. Expression s is of the form $R[((c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})\ r)]$. Then again $N'(s)$ is not an L_{lcc}-WHNF and irreducible. □

Lemma 6.8. *Let $s \in \mathbb{E}_{\mathcal{L}}$. Then s is an L_{name}-WHNF iff $N'(s)$ is an L_{lcc}-WHNF.*

Proof. If $s = L[\lambda x.s']$ or $s = L[(c\ s_1 \ldots s_{\mathrm{ar}(c)})]$ then $N'(s) = \lambda x.\sigma(N'(s'))$ or $N'(s) = (c\ \sigma(N'(s_1)) \ldots \sigma(N'(s_{\mathrm{ar}(c)})))$ respectively, both of which are L_{lcc}-WHNFs.

For the other direction assume that $N'(s)$ is an abstraction or a constructor application. The analysis of the reduction correspondence in the previous paragraph shows that s cannot have a normal order redex in L_{name}, since otherwise $N'(s)$ cannot be an L_{lcc}-WHNF. Lemma 6.7 shows that s cannot be irreducible in L_{name}, but not an L_{name}-WHNF. Thus s must be an L_{name}-WHNF.

Transferring L_{lcc}-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 $\mathbb{E}_{\mathcal{L}}$-expression and $N'(s) \xrightarrow{lcc} t$. We split the argument into three cases based on whether or not a normal order reduction is applicable to s:

- If $s \xrightarrow{(name)} r$, then we can use the already developed diagrams, since normal-order reduction in both calculi is unique.
- s is a WHNF. This case cannot happen, since then $N'(s)$ would also be a WHNF (see Lemma 6.8) and thus irreducible.
- s is irreducible but not a WHNF. Then Lemma 6.7 implies that $N'(s)$ is irreducible in L_{lcc} which contradicts the assumption $N'(s) \xrightarrow{lcc} t$. Thus this case is impossible.

We summarize the diagrams in the following lemma:

Lemma 6.9. *Normal-order reductions in L_{name} can be transferred into reductions in L_{lcc}, and vice versa, by the diagrams in Figure 10.*

Proposition 6.10. *N' and N are convergence equivalent,* i.e. *for all $\mathbb{E}_{\mathcal{L}}$-expressions s: $s{\downarrow_{name}} \iff N'(s){\downarrow_{lcc}}$ ($s{\downarrow_{name}} \iff N(s){\downarrow_{lcc}}$, resp.).*

Proof. We first prove convergence equivalence of N': Suppose $s{\downarrow_{name}}$. Let $s \xrightarrow{name,k} s_1$ where s_1 is a WHNF. We show that there exists an L_{lcc}-WHNF s_2 such that $N'(s) \xrightarrow{lcc,*} s_2$ by induction on k. The base case follows from Lemma 6.8. The induction step follows by applying a diagram from Lemma 6.9 and then using the induction hypothesis.

For the other direction we assume that $N'(s){\downarrow_{lcc}}$, *i.e.* there exists a WHNF $s_1 \in L_{lcc}$ *s.t.* $N'(s) \xrightarrow{lcc,k} s_1$. By induction on k we show that there exists a L_{name}-WHNF s_2 such that $s \xrightarrow{name,*} s_2$. The base case is covered by Lemma 6.8. The induction step uses the diagrams. Here it is necessary to observe that the diagrams for the reductions (lapp), (lcase), and (lseq) cannot be applied infinitely often without being interleaved with other reductions. This holds, since let-shifting by (lapp), (lcase), and (lseq) moves `letrec`-symbols to the top of the expressions, and thus there are no infinite sequences of these reductions.

It remains to show convergence equivalence of N: Let $s{\downarrow_{name}}$ then $N'(s){\downarrow_{lcc}}$, since N' is convergence equivalent. Lemma 6.4 implies $N'(s) \sim_{lcc} N(s)$ and thus $N(s){\downarrow_{lcc}}$ must hold. For the other direction Lemma 6.4 shows that $N(s){\downarrow_{lcc}}$ implies $N'(s){\downarrow_{lcc}}$. Using convergence equivalence of N' yields $s{\downarrow_{name}}$.

Lemma 6.11. *The translation N is compositional,* i.e. *for all expressions s and all contexts C: $N(C[s]) = N(C)[N(s)]$.*

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

Proposition 6.12. *For all $s_1, s_2 \in \mathbb{E}_{\mathcal{L}}$: $N(s_1) \leq_{lcc} N(s_2) \implies s_1 \leq_{name} s_2$,* i.e. *$N$ is adequate.*

Proof. Since N is convergence-equivalent (Proposition 6.10) and compositional by Lemma 6.11, we derive that N is adequate (see [SSNSS08] and Section 2).

Lemma 6.13. *For* `letrec`*-free expressions $s_1, s_2 \in \mathbb{E}_\lambda$ the following holds: $s_1 \leq_{name} s_2 \implies s_1 \leq_{lcc} s_2$.*

Proof. Note that the claim only makes sense since clearly $\mathbb{E}_\lambda \subseteq \mathbb{E}_{\mathcal{L}}$. Let s_1, s_2 be `letrec`-free such that $s_1 \leq_{name} s_2$. Let C be an L_{lcc}-context such that $C[s_1]{\downarrow_{lcc}}$, *i.e.* $C[s_1] \xrightarrow{lcc,k} \lambda x.s_1'$. By comparing the reduction strategies in L_{name} and L_{lcc}, we obtain that $C[s_1] \xrightarrow{name,k} \lambda x.s_2'$ (by the identical reduction sequence) since $C[s_1]$ is `letrec`-free. Thus, $C[s_1]{\downarrow_{name}}$ and also $C[s_2]{\downarrow_{name}}$, *i.e.* there is a normal order reduction in L_{name} for $C[s_2]$ to a WHNF. Since $C[s_2]$ is `letrec`-free, we can perform the identical reduction in L_{lcc} and obtain $C[s_2]{\downarrow_{lcc}}$.

The language L_{lcc} is embedded into L_{name} (and also L_{LR}) by $\iota(s) = s$.

Proposition 6.14. *For all $s \in \mathbb{E}_{\mathcal{L}}$: $s \sim_{name} \iota(N(s))$.*

Proof. We first show that for all expressions $s \in \mathbb{E}_{\mathcal{L}}$: $s \sim_{name} \iota(N(s))$. 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 6.12) implies $s \sim_{name} \iota(N(s))$.

Proposition 6.15. *For all $s_1, s_2 \in \mathbb{E}_{\mathcal{L}}$: $s_1 \leq_{name} s_2 \implies N(s_1) \leq_{lcc} N(s_2)$.*

Proof. For this proof it is necessary to observe that $\mathbb{E}_\lambda \subseteq \mathbb{E}_{\mathcal{L}}$, thus we can treat L_{lcc} expressions as L_{name} expressions. Let $s_1, s_2 \in \mathbb{E}_{\mathcal{L}}$ and $s_1 \leq_{name} s_2$. By Proposition 6.14: $N(s_1) \sim_{name} s_1 \leq_{name} s_2 \sim_{name} N(s_2)$, thus $N(s_1) \leq_{name} N(s_2)$. Since $N(s_1)$ and $N(s_2)$ are `letrec`-free, we can apply Lemma 6.13 and thus have $N(s_1) \leq_{lcc} N(s_2)$.

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

Theorem 6.16. *N and $N \circ W$ are fully-abstract,* i.e. *for all expressions $s_1, s_2 \in \mathbb{E}_{\mathcal{L}}$: $s_1 \leq_{LR} s_2 \iff N(W(s_1)) \leq_{lcc} N(W(s_2))$.*

Proof. Full-abstractness of N follows from Propositions 6.12 and 6.15. Full-abstractness of $N \circ W$ thus holds, since W is fully-abstract (Corollary 5.32).

Since N is surjective, this and Corollary 6.17 imply:

Corollary 6.17. *N and $N \circ W$ are isomorphisms according to Definition 2.7.*

The results also allow us to transfer the characterization of expressions in L_{lcc} into L_{LR}. With $cBot_{LR}$ we denote the set of $\mathbb{E}_{\mathcal{L}}$-expressions s with the property that for all substitutions σ: if $\sigma(s)$ is closed, then $\sigma(s) \Uparrow_{LR}$.

Proposition 6.18. *Let s be a closed $\mathbb{E}_{\mathcal{L}}$-expression. Then there are three cases: $s \sim \Omega$, $s \sim_{LR} \lambda x.s'$ for some s', $s \sim_{LR} c\ s_1 \ldots s_n$ for some terms $s_1, \ldots, s_n$ and constructor c. Moreover, the three cases are disjoint. For two closed $\mathbb{E}_{\mathcal{L}}$-expressions s, t with $s \leq_{LR} t$: Either $s \sim_{LR} \Omega$, or $s \sim_{LR} c\ s_1 \ldots s_n$, $t \sim c\ t_1 \ldots t_n$ and $s_i \leq_{LR} t_i$ for all i for some terms $s_1, \ldots, s_n, t_1, \ldots, t_n$ and constructor c, or $s \sim_{LR} \lambda x.s'$ and $t \sim_{LR} \lambda x.t'$ for some expressions s', t' with $s' \leq_{LR} t'$, or $s \sim_{LR} \lambda x.s'$ and $t \sim_{LR} c\ t_1 \ldots t_n$ for some term $s' \in cBot_{LR}$, expressions $t_1, \ldots, t_n$ and constructor c.*

Proof. This follows by Proposition 4.32 and since $N \circ W$ is surjective, compositional and fully abstract, and the identity on constructors.

7 On Similarity in L_{LR}

In this section we will explain co-inductive and inductive (bi)similarity for L_{LR}. Our results of the previous sections then enable us to show that these bisimilarities coincide with contextual equivalence in L_{LR}.

7.1 Overview of soundness and completeness proofs for similarities in L_{LR}

Before we give details of the proof for lifiting soundness and completeness of similarities from L_{lcc} to L_{LR}, we show an outline of the proof in Figure 11. The diagram shows fully abstract translations between the calculi L_{LR}, L_{name}, and L_{lcc} defined and studied in Sections 5 and 6, where Corollary 5.32 and Theorem 6.16 show full abstractness for W and N, respectively. These fully-abstract translations that are also surjective, and the identity on letrec-free expressions, allow us to prove that $s_1 \leq_{LR} s_2 \iff N(W(s_1)) \leq_{lcc} N(W(s_2))$. By Theorem 4.37 in L_{lcc}, this is further equivalent to $N(W(s_1)) \preccurlyeq_{lcc}^o N(W(s_2))$. The proof is completed by using the translations by transferring the equations back and forth between L_{LR} and L_{lcc} in this section in order to finally show that $s_1 \leq_{LR} s_2 \iff s_1 \preccurlyeq^o_{LR,Q_{CE}} s_2$ in Theorem 7.6.

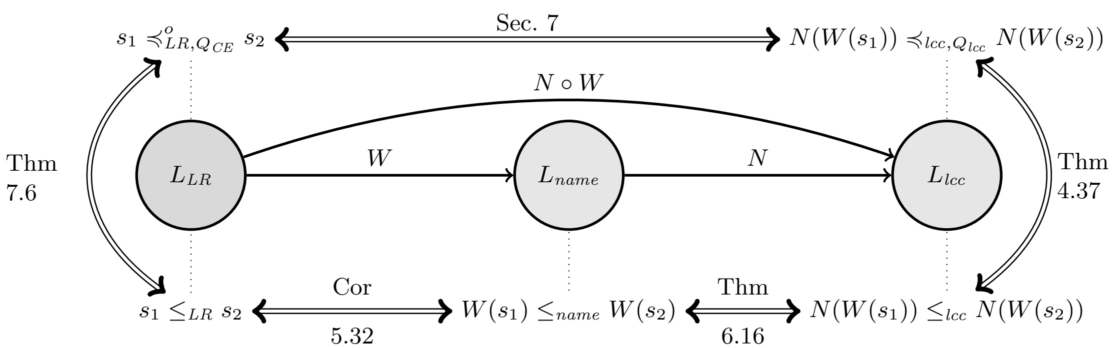


Fig. 11. The structure of the reasoning for the similarities in L_{LR} .

7.2 Similarity in L_{LR}

The definition of L_{LR}-WHNFs implies that they are of the form $R[v]$, where v is either an abstraction $\lambda x.s$ or a constructor application $(c\, s_1 \ \ldots \ s_{\mathrm{ar}(c_i)})$, and where R is an L_{LR}-*AWHNF-context* according to the grammar $R ::= [\cdot] \mid (\texttt{letrec}\ Env\ \texttt{in}\ [\cdot])$ if v is an abstraction, and R is an L_{LR}-*CWHNF-context* according to the grammar $R ::= [\cdot] \mid (\texttt{letrec}\ Env\ \texttt{in}\ [\cdot]) \mid (\texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ x_m)$ if v is a constructor application. Note that L_{LR}-AWHNF-contexts and L_{LR}-CWHNF-contexts are special L_{LR}-reduction contexts, also called L_{LR}*-WHNF-contexts*.

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

Definition 7.1. *Let $\leq_{LR,Q_{CE}}$ be defined for L_{LR} as instantiating the relation $\leq_{\mathcal{Q}}$ in Definition 2.6 with the calculus L_{LR} and the set $\mathcal{Q}$ with Q_{CE} from Definition 4.38.*

The relation $\preccurlyeq_{LR,Q_{CE}}$ is $\mathcal{Q}$-similarity (Definition 2.5) instantiated for the calculus L_{LR} with the set of contexts Q_{CE} (Definition 4.38). Its open extension is denoted with $\preccurlyeq^o_{LR,Q_{CE}}$.

Proposition 7.2. *Let s_1, s_2 be any two closed $\mathbb{E}_{\mathcal{L}}$-expressions. Then $s_1 \leq_{LR} s_2$ iff $s_1 \leq_{LR,Q_{CE}} s_2$.*

Proof. The $\Rightarrow$ direction is trivial. We show $\Leftarrow$, the nontrivial part: Assume that $s_1 \leq_{LR,Q_{CE}} s_2$. Then $N(W(s_1)) \leq_{lcc,Q_{CE}} N(W(s_2))$, since for every $n \geq 0$ and context $Q = Q_n(\ldots(Q_2(Q_1[])\ldots)$ with $Q_i \in Q_{CE}$, we have $N(W(Q)) = Q$, and also $Q(s_i)\downarrow_{LR} \iff Q(s_i)\downarrow_{lcc}$, since $N \circ W$ is convergence-equivalent and compositional, and the identity on `letrec`-free expressions. Now Theorem 4.39 shows $N(W(s_1)) \leq_{lcc} N(W(s_2))$, and then Theorem 6.16 shows $s_1 \leq_{LR} s_2$.

The following lemma is helpful in applying Theorem 4.8.

Lemma 7.3. *The closed part of the calculus L_{LR} is convergence-admissible: For all contexts $Q \in Q_{CE}$, and closed L_{LR}-WHNFs w: $Q(s){\downarrow_{LR}} w$ iff $\exists v : s{\downarrow_{LR}} v$ and $Q(v){\downarrow_{LR}} w$.*

Proof. "⇒": First assume Q is of the form $([\cdot]\ r)$ for closed r. Let $(s\ r){\downarrow_{LR}} w$. There are two cases, which can be verified by induction on the length k of a reduction sequence $(s\ r) \xrightarrow{LR,k} w$: $(s\ r) \xrightarrow{LR,*} ((\lambda x.s')\ r) \xrightarrow{LR,*} w$, where $s \xrightarrow{LR,*} (\lambda x.s')$, and the claim holds. The other case is $(s\ r) \xrightarrow{LR,*} (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ ((\lambda x.s')\ r)) \xrightarrow{LR,*} w$, where $s \xrightarrow{LR,*} (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (\lambda x.s'))$. In this case $((\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (\lambda x.s'))\ r) \xrightarrow{LR,(lapp)} (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ ((\lambda x.s')\ r)) \xrightarrow{LR,*} w$, and thus the claim is proven. The other cases where Q is of the form $(\texttt{case}_T\ [\cdot]\ \texttt{of}\ \ldots)$ can be proven similarly.
The "⇐"-direction can be proven using induction on the length of reduction sequences.

Lemma 7.4. *In L_{LR}, the equation $(\leq^c_{LR})^o \ = \ \leq_{LR}$ holds.*

Proof. If s, t are (open) $\mathbb{E}_{\mathcal{L}}$-expressions with $s \leq_{LR} t$, then $(\lambda x_1. \ldots x_n.s)\ s_1 \ldots s_n \leq^c_{LR} (\lambda x_1. \ldots x_n.t)\ s_1 \ldots s_n$ for closed expressions s_i, and then by correctness of reduction in L_{LR}, $\sigma(s) \leq^c_{LR} \sigma(t)$, and hence $\leq_{LR}\ \subseteq (\leq^c_{LR})^o$.

If for all closing $\mathbb{E}_{\mathcal{L}}$-substitutions σ: $\sigma(s) \leq^c_{LR} \sigma(t)$, then using the fully abstract translations $N \circ W$, we obtain $N \circ W(\sigma)(N \circ W(s)) \leq^c_{lcc} N \circ W(\sigma)(N \circ W(t))$, hence $N \circ W(s) \leq^c_{lcc} N \circ W(t))$ by Theorem 4.39. Again using fully abstractness of $N \circ W$, we obtain $s \leq_{LR} t$.

Theorem 7.5. *In L_{LR}, for closed $\mathbb{E}_{\mathcal{L}}$-expressions s, t the statements $s \preccurlyeq_{LR,Q_{CE}} t$, $s \leq_{LR,Q_{CE}} t$ and $s \leq_{LR} t$ are all equivalent.*

Proof. Lemma 7.3 shows that Theorem 4.8 is applicable for the testing contexts from Q_{CE}, i.e. $\preccurlyeq_{LR,Q_{CE}} = \leq_{LR,Q_{CE}}$ and Proposition 7.2 shows $\leq_{LR,Q_{CE}} = \leq^c_{LR}$

For open $\mathbb{E}_{\mathcal{L}}$-expressions, we can lift the properties from L_{lcc}, which also follows from full abstraction of $N \circ W$ and from Lemma 4.40.

The results above imply the following theorem:

Main Theorem 7.6 $\leq_{LR} = \preccurlyeq^o_{LR,Q_{CE}}$.

Proof. Theorem 7.5 shows $\preccurlyeq_{LR,Q_{CE}} \ = \ \leq_{LR,Q_{CE}} \ = \ \leq^c_{LR}$, hence $\preccurlyeq^o_{LR,Q_{CE}} \ = \ (\leq^c_{LR})^o$. Then Lemma 7.4 shows $(\leq^c_{LR})^o \ = \ \leq_{LR} \ = \ \preccurlyeq^o_{LR,Q_{CE}}$.

The Main Theorem 7.6 implies that our embedding of L_{lcc} into the call-by-need letrec calculus L_{LR} (modulo $\sim$) is isomorphic *w.r.t.* the corresponding term models, *i.e.*

Theorem 7.7. *The identical embedding $\iota : \mathbb{E}_\lambda \to \mathbb{E}_{\mathcal{L}}$ is an isomorphism according to Definition 2.7.*

Remark 7.8. Consider a polymorphically typed variant of L_{LR}, say L^{poly}_{LR}, and a corresponding type-indexed contextual preorder $\leq_{LR,\text{poly},\tau}$ which relates expressions of polymorphic type τ and where the testing contexts are restricted to well-typed contexts, *i.e.* for s, t of type τ the inequality $s \leq_{LR,\text{poly},\tau} t$ holds iff for all contexts C such that $C[s]$ and $C[t]$ are well-typed: $C[s]{\downarrow_{LR}} \implies C[t]{\downarrow_{LR}}$. Obviously for all expressions s, t of type τ the inequality $s \leq_{LR} t$ implies $s \leq_{LR,\text{poly},\tau} t$, since any test (context) performed for $\leq_{LR,\text{poly},\tau}$ is also included in the tests for $\leq_{LR}$ (there are more contexts). Thus the main theorem implies that $\preccurlyeq^o_{LR,Q_{CE}}$ is sound *w.r.t.* the typed preorder $\leq_{LR,\text{poly},\tau}$. Of course completeness does not hold, and requires another definition of similarity which respects the typing.

7.3 Similarity upto $\sim_{LR}$

A more comfortable tool to prove program equivalences in L_{LR} is the following similarity definition which allows to simplify intermediate expressions that are known to be equivalent.

Definition 7.9 (Similarity upto $\sim_{LR}$). *Let $\preccurlyeq_{LR,\sim}$ be the greatest fixpoint of the following operator $F_{LR,\sim}$ on closed $\mathbb{E}_{\mathcal{L}}$-expressions:*

We define an operator $F_{LR,\sim}$ on binary relations η on closed L_{lcc}-expressions:
$s\ F_{LR,\sim}(\eta)\ t$ iff the following holds:

1. *If $s \sim_{LR} \lambda x.s'$ then there are two possibilities: (i) if $t \sim_{LR} (c\ t_1 \ldots t_n)$ then $s' \in cBot_{LR}$, or (ii) if $t \sim_{LR} \lambda x.t'$ then for all closed r : $((\lambda x.s')\ r)\ \eta\ ((\lambda x.t')\ r)$;*
2. *If $s \sim_{LR} (c\ s_1\ \ldots\ s_n)$ then $t \sim_{LR} (c\ t_1 \ldots t_n)$ and $s_i\ \eta\ t_i$ for all i.*

Lemma 7.10. $\leq^c_{LR} \subseteq \preccurlyeq_{LR,\sim}$

Proof. We show that $\eta := \leq^c_{LR}$ is $F_{LR,\sim}$-dense, i.e. $\eta \subseteq F_{LR,\sim}(\eta)$.

Let $s\ \eta\ t$ and $s \sim_{LR} \lambda x.s'$. Since $s \leq^c_{LR} t$ either $t \sim_{LR} \lambda x.t'$ or $t \sim_{LR} c\ t_1\ \ldots\ t_n$ and $s' \in cBot_{LR}$. For the latter case we are finished. For the former case we have $\lambda x.t' \sim^c_{LR} t$. Since $\leq^c_{LR}$ is a precongruence, this implies $((\lambda x.s')\ r) \leq_{LR} ((\lambda x.t')\ r)$ for all closed $\mathbb{E}_{\mathcal{L}}$-expressions r. Thus we conclude $s\ F_{LR,\sim}(\eta)\ t$.

Now let $s\ \eta\ t$ and $s \sim^c_{LR} c\ s_1\ \ldots\ s_n$. Then $t \sim^c_{LR} (c\ t_1\ \ldots\ t_n)$ by Proposition 6.18. The contexts $C_i := (\texttt{case}\ []\ \texttt{of} \ldots (c\ x_1 \ldots\ x_n \to x_i) \ldots)$ where all other right hand sides of case-alternatives are $\perp$, show that also $s_i \leq_{LR} t_i$ must hold, since otherwise $s \leq^c_{LR} t$ cannot hold. Thus also in this case $s\ F_{LR,\sim}(\eta)\ t$ holds.

Lemma 7.11. $N(W(\preccurlyeq_{LR,\sim})) \subseteq \preccurlyeq_{lcc,\sim}$.

Proof. We show that $\eta := \{(N(W(s)), N(W(t))) \mid s \preccurlyeq_{LR,\sim} t\}$ is $F_{lcc,\sim}$-dense (see Definition 4.43), i.e. $\eta \subseteq F_{lcc,\sim}(\eta)$. Let $s \preccurlyeq_{LR,\sim} t$ for closed s, t. If $N(W(s)) \sim_{lcc} \lambda x.s'$, then also $s \sim_{LR} \lambda x.s'$. Now there are two cases: If $t \sim_{LR} (c\ t_1\ \ldots\ t_n)$ then $s' \in cBot_{LR}$ must hold. Then also $s' \in cBot$ and $N(W(t)) \sim_{lcc} (c\ t_1\ \ldots\ t_n)$ and we are finished. If $t \sim_{LR} \lambda x.t'$ then for all closed $\mathbb{E}_{\mathcal{L}}$-expressions r: $(\lambda x.s')\ r \preccurlyeq_{LR,\sim} (\lambda x.t')\ r$ (by unfolding the fixpoint equation for $F_{LR,\sim}$). Since $N \circ W$ is surjective, compositional and fully abstract, this also shows $N(W(\lambda x.s'))\ r\ \eta\ N(W(\lambda x.t'))\ r$ for all L_{lcc}-expressions r.

If $N(W(s)) \sim_{lcc} (c\ s_1 \ldots s_n)$, then also $s \sim_{LR} (c\ s_1 \ldots s_n)$. Now $s \preccurlyeq_{LR,\sim} t$ shows that $t \sim_{LR} (c\ t_1\ \ldots\ t_n)$ such that for all i: $s_i \preccurlyeq_{LR,\sim} t_i$. Hence $(s_i, t_i) \in \eta$ and also $N(W(t)) \sim_{lcc} (c\ t_1\ \ldots\ t_n)$, since $N \circ W$ is fully abstract.

Theorem 7.12. $\leq_{LR} = \preccurlyeq^o_{LR,\sim}$

Proof. For the closed relations, one direction of the equation $\preccurlyeq_{LR,\sim} = \leq^c_{LR}$ is Lemma 7.10, the other direction follows from Lemma 7.11, since $s \preccurlyeq_{LR,\sim} t$ implies $N(W(s)) \preccurlyeq_{lcc,\sim} N(W(t))$ which in turn implies $N(W(s)) \leq^c_{lcc} N(W(t))$ and finally, full-abstraction of $N \circ W$ shows $s \leq^c_{LR} t$.

For the open extension the claimed equality holds, since $s \leq_{LR} t$ iff $\sigma(s) \leq_{LR} \sigma(t)$ for all closing substitutions σ: This holds, since for $\sigma = \{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$ the equation $\sigma(s) \sim_{LR} \texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ s$ holds by correctness of the general copy rule (gcp) (Proposition 5.34) and of garbage collection (gc) (Theorem 3.7).

We demonstrate the use of similarity upto $\sim_{LR}$ in the following example:

Example 7.13. As an example we prove the the list law $R[map\ (\lambda x.\texttt{True})\ (repeat\ u)] \sim_{LR} R'[(repeat\ \texttt{True})]$ where u is a closed expression and R', R, resp. contains the definition of *repeat*, or *repeat* and *map*, resp., i.e. the corresponding $\mathbb{E}_{\mathcal{L}}$-expressions are:

$$
\begin{array}{l}
s := \texttt{letrec} \\
\qquad repeat = \lambda x.\texttt{Cons}\ x\ (repeat\ x), \\
\qquad map = \lambda f.\lambda xs.\texttt{case}_{List}\ xs\ \texttt{of}\ (\texttt{Nil} \to \texttt{Nil})\ (\texttt{Cons}\ y\ ys \to \texttt{Cons}\ (f\ y)\ (map\ f\ ys)) \\
\quad \texttt{in}\ map\ (\lambda x.\texttt{True})\ (repeat\ u) \\
t := \texttt{letrec} \\
\qquad repeat = \lambda x.\texttt{Cons}\ x\ (repeat\ x), \\
\quad \texttt{in}\ repeat\ \texttt{True}
\end{array}
$$

Let $\eta := \{(t, s), (s, t)\} \cup \{(\texttt{True}, \texttt{True})\}$. We show that $\eta \subseteq F_{LR,\sim}(\eta)$ which implies $s \preccurlyeq_{LR,\sim} t$ as well as as $t \preccurlyeq_{LR,\sim} s$ and thus by Theorem 7.12 also $s \sim_{LR} t$.

Evaluating s and t in normal order first shows: $s \sim_{LR} v_1, t \sim_{LR} v_2$ with

$$
\begin{array}{l}
v_1 = \texttt{letrec} \\
\qquad \mathit{repeat} = \lambda x.\texttt{Cons}\ x\ (\mathit{repeat}\ x), \\
\qquad \mathit{map} = \lambda f.\lambda xs.\texttt{case}_{List}\ xs\ \texttt{of}\ (\texttt{Nil} \rightarrow \texttt{Nil})\ (\texttt{Cons}\ y\ ys \rightarrow \texttt{Cons}\ (f\ y)\ (\mathit{map}\ f\ ys)) \\
\qquad f_1 = (\lambda x.\texttt{True}), x_1 = t, xs_1 = \texttt{Cons}\ x_1'\ x_2', x_1' = x_1, x_2' = (\mathit{repeat}\ t), y_1 = x_1', ys_1 = x_2' \\
\quad \texttt{in Cons}\ (f_1\ y_1)\ (\mathit{map}\ f_1\ ys_1)
\end{array}
$$

$$
\begin{array}{l}
v_2 = \texttt{letrec} \\
\qquad \mathit{repeat} = \lambda x.\texttt{Cons}\ x\ (\mathit{repeat}\ x), \\
\qquad x_1 = \texttt{True} \\
\quad \texttt{in Cons}\ x_1\ (\mathit{repeat}\ x_1))
\end{array}
$$

Using correctness of garbage collection, copying of bindings (gcp), shifting constructors over `letrec`, and the other correct reduction rules (see Theorem 3.7 and Proposition 5.34), we can simplify as follows: $v_1 \sim_{LR}$ `Cons True` s and $v_2 \sim_{LR}$ `Cons True` t. Now the proof is finished, since obviously `True` η `True` and $s\ \eta\ t$, $t\ \eta\ s$.

8 Conclusion

In this paper we have shown that co-inductive applicative bisimilarity, in the style of Howe, and also the inductive variant, is equivalent to contextual equivalence in a deterministic call-by-need calculus with letrec, case, data constructors, and `seq` which models the (untyped) core language of Haskell. This also shows soundness of untyped applicative bisimilarity for the polymorphically typed variant of L_{LR}. As a further work one may try to establish a coincidence of the typed applicative bisimilarity and contextual equivalence for a polymorphically typed core language of Haskell.

References

AB97. Zena M. Ariola and Stefan Blom. Cyclic lambda calculi. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings*, volume 1281 of *Lecture Notes in Computer Science*, pages 77–106. Springer, 1997.

AB02. Z. M. Ariola and S. Blom. Skew confluence and the lambda calculus with letrec. *Ann. Pure Appl. Logic*, 117:95–168, 2002.

Abr90. S. 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 1995*, pages 233–246, San Francisco, California, 1995. ACM.

AK94. Z. M. Ariola and J. W. Klop. Cyclic Lambda Graph Rewriting. In *LICS 1994*, pages 416–425. IEEE, 1994.

AO93. S. Abramsky and C.-H. L. 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.

DBG97. Louise A. Dennis, Alan Bundy, and Ian Green. Using a generalisation critic to find bisimulations for coinductive proofs. In *Proceedings of the 14th International Conference on Automated Deduction*, CADE-14, pages 276–290, London, UK, UK, 1997. Springer-Verlag.

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

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

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

How89. D. Howe. Equality in lazy computation systems. In *LICS 1989*, pages 198–203, 1989.

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

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

JV06. P. Johann and J. Voigtländer. The impact of seq on free theorems-based program transformations. *Fund. Inform.*, 69(1–2):63–102, 2006.

KKSdV97. R. Kennaway, J. W. Klop, M. Ronan Sleep, and F.-J. de Vries. Infinitary lambda calculus. *Theoret. Comput. Sci.*, 175(1):93–125, 1997.

KW06. V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In *POPL 2006*, pages 141–152, New York, NY, USA, 2006. ACM.

Lau93. J. Launchbury. A natural semantics for lazy evaluation. In *POPL 1993*, pages 144–154, 1993.

Mil80. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

Mil99. R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge university press, 1999.

Mor68. J.H. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, MIT, 1968.

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

MS99. A. K. D. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *POPL 1999*, pages 43–56. ACM, 1999.

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

NH09. K. Nakata and M. Hasegawa. Small-step and big-step semantics for call-by-need. *J. Funct. Program.*, 19:699–722, 2009.

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

Pit97. A. M. Pitts. Operationally-based theories of program equivalence. In *Semantics and Logics of Computation*. Cambridge University Press, 1997.

Pit11. A. M. Pitts. Howe's method for higher-order languages. In D. Sangiorgi and J. Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, volume 52 of *Cambridge Tracts Theoret. Comput. Sci.*, chapter 5, pages 197–232. Cambridge University Press, November 2011.

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

Ses97. P. Sestoft. Deriving a lazy abstract machine. *J. Funct. Programming*, 7(3):231–264, 1997.

SKS11. D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimulations for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 33(1):5, 2011.

SS07. M. Schmidt-Schauß. Correctness of copy in calculi with letrec. In *RTA 2007*, volume 4533 of *LNCS*, pages 329–343. Springer, 2007.

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

SSMS13. M. Schmidt-Schauß, E. Machkasova, and D. Sabel. Extending Abramsky's lazy lambda calculus: (non)-conservativity of embeddings. In *24th International Conference on Rewriting Techniques and Applications (RTA 2013)*, volume 21 of *LIPIcs*, pages 239–254, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

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

SSNSS09. M. Schmidt-Schauß, J. Niehren, J. Schwinghammer, and D. Sabel. Adequacy of compositional translations for observational semantics. Frank report 33, Inst. f. Informatik, Goethe-University, Frankfurt, 2009. `http://www.ki.informatik.uni-frankfurt.de/papers/frank/`.

SSS11. D. Sabel and M. Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In *PPDP 2011*, pages 101–112, New York, NY, USA, July 2011. ACM.

SSS12. David Sabel and Manfred Schmidt-Schauß. Conservative concurrency in Haskell. In Nachum Dershowitz, editor, *Proceedings of the 27th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2012), 25-28 June 2012, Dubrovnik, Croatia*, pages 561–570, 2012.

SSSM10. M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Simulation in the call-by-need lambda-calculus with letrec. In *RTA 2010*, volume 6 of *LIPIcs*, pages 295–310. Schloss Dagstuhl, 2010.

SSSM11. M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Counterexamples to applicative simulation and extensionality in non-deterministic call-by-need lambda-calculi with letrec. *Inf. Process. Lett.*, 111(14):711–716, 2011.

SSSS08. M. Schmidt-Schauß, M. Schütz, and D. Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.

SW01. D. Sangiorgi and D. Walker. *The π-calculus: a theory of mobile processes.* Cambridge university press, 2001.

VJ07. J. Voigtländer and P. Johann. Selective strictness and parametricity in structural operational semantics, inequationally. *Theor. Comput. Sci*, 388(1–3):290–318, 2007.