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

July 4, 2012

Abstract. This paper shows equivalence of applicative similarity and contextual approximation, and hence also of 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. Bisimilarity simplifies equivalence proofs in the calculus and opens a way for more convenient correctness proofs for program transformations.
The proof is by a fully abstract and surjective transfer of the contextual approximation into a call-by-name calculus, which is an extension of Abramsky's lazy lambda calculus. In the latter calculus equivalence of similarity and contextual approximation can be shown by Howe's method. Using an equivalent but inductive definition of behavioral preorder 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 tress, 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, and also with a call-by-need letrec calculus with a less complex definition of reduction than LR.

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 the value 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 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 given the expressions, contexts, a notion of evaluation, and the set of values, contextual equivalence can be defined, and thus it 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 correctness proofs of *e.g.* correctness of program transformations.

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 obtained by reduction have a canonical top operator. This does not apply to calculi with `letrec`, since `letrec`-expressions are a result of evaluation and `letrec` is not canonical. (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. In this paper we extend our previous results to a full core language of Haskell which makes our results applicable to a large set of features of real life programming languages. As a model of Haskell's core language we use the call-by-need lambda calculus L_{LR} which was introduced, motivated, and analyzed in [SSSS08]. The calculus L_{LR} extends the usual 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 of this paper is that applicative bisimilarity in the coinductive as well as in the inductive variant and contextual equivalence coincide in L_{LR}. Consequently, 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 besides soundness of applicative bisimilarity we also show completeness, our results can also be used

to disprove contextual equivalence of expressions in L_{LR}. Our result also shows that the untyped applicative bisimilarity is sound for a polymorphic variant of L_{LR}, and hence for Haskell. Although the proof is worked out for the language L_{LR}, we claim that this result also holds in the calculus L_{LR} without `seq`, since our proofs will also be valid if `seq` is not there.

We also introduce and describe two variant calculi of L_{LR}: a call-by-name variant L_{name} and letrec-free 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 L_{LR} into the call-by-need calculus L_{LR} is an isomorphism of the respective term models.

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 [SSS11a] where we recently have shown that Haskell's deterministic core language can be conservatively embedded in the calculus CHF modelling Concurrent Haskell (see the technical report [SSS11b]). Although the calculus is restricted to have a monomorphic type system and a slightly type-restricted `seq`, program transformations and optimizations of the untyped core language of Haskell (which is equivalent to L_{LR}) are also valid in CHF, since every untyped program equivalence remains valid in a typed setting as long as the equivalence is correctly typed.

Since the traditional definition of normal order reduction in L_{LR} (see [SSSS08]) is rather complex, we provide another standard reduction which also implements call-by-need evaluation, but, compared to L_{LR}, it more eagerly copies values and is closer to an abstract machine. The corresponding calculus is called L_{need}. We show that convergence in L_{LR} and convergence in L_{need} coincide, and thus both calculi are equivalent. A consequence is that our results are also applicable to the simpler calculus L_{need}.

Outline of the Proof

The main proof technique to obtain our result is to translate the expressions of L_{LR} into the untyped calculus 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 call-by-need reduction. Another view on L_{lcc} is that L_{lcc} minimally extends Abramsky's lazy lambda calculus by Haskell's primitives: data constructors, `case`-expressions, and the `seq`-operator. Note that data constructors *cannot* be adequately encoded in the lazy lambda calculus without data constructors, unless some type system is added to the calculus with constructors (this was for instance observed in [SSNSS08]). We will also argue that the `seq`-operator is necessary as a primitive of the language.

In more detail, the translation is performed in two steps: the translation W translates L_{LR} into L_{name}, a call-by-name calculus *with* `letrec` (and `case`, constructors, and `seq`), and the translation $N : L_{name} \to L_{lcc}$ which then removes the `letrec`-expressions by using a family of fixpoint combinators.

We will show that W and N are fully-abstract translations, and thus their composition is also fully-abstract. Full-abstractness means that contextual equivalence is reflected and preserved by the translation: Let $\sim_{LR}$ and $\sim_{lcc}$ be the contextual equivalences of the calculi L_{LR} and L_{lcc}, then the translation $N \circ W : L_{LR} \to L_{lcc}$ is fully-abstract if $s_1 \sim_{LR} s_2 \iff (N \circ W)(s_1) \sim_{lcc} (N \circ W)(s_2)$ holds for all s_1, s_2. For the translation N the full-abstractness proof is performed syntactically by comparing reductions in L_{name} and L_{LR}. For the translation W we show full-abstractness by analyzing the reductions of L_{LR} and L_{name} *w.r.t.* their infinite unfoldings which remove all `letrec`-expressions and a corresponding evaluation (as infinitary rewriting, see [KKSdV97,SS07]) of infinite expressions. Full abstraction of $N \circ W$ then shows that the contextual equivalence between L_{LR} and L_{lcc} can be transferred. Coincidence of bisimilarity and contextual equivalence in L_{lcc} can be shown by Howe's method [How96], hence to accomplish our result bisimilarity must be transferred from L_{lcc} back into L_{LR} *w.r.t.* the translations. This final part is done by proving coincidence of co-inductive bisimilarity and an inductive variant of behavioral equivalence, and by using full abstraction of $N \circ W$ again.

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 $\lambda x.\Omega$.

In [Jef94] there is 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 7.11 shows that this approach is extensible to a cyclic `let`.

Related work on calculi with recursive bindings includes the following foundational papers. An early paper that proposes cyclic let-bindings (as graphs) is [AK94], where reduction and confluence properties are discussed. [AFM+95,AF97,MOW98] present call-by-need lambda calculi with non-recursive

`let` and a let-less formulation of call-by-need reduction. For a calculus with non-recursive `let` it is shown in [MOW98] that call-by-name and call-by-need evaluation induce the same observational equivalences. Call-by-need lambda calculi with a recursive `let` that correspond to our calculus L_{LR} are also presented in [AFM+95,AF97,AB02,NH09]. In [AB02] it is shown that there exist infinite normal forms and that the calculus satisfies a form of confluence. Nevertheless these calculi are not extended by 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 as ours based on contextual equivalence. The calculus L_{LR} was introduced and analyzed in [SSSS08] to show correctness of Nöcker's strictness analysis using abstract reduction. As side results, in [SSSS08] a lot of program transformations are proved correct *w.r.t.* contextual equivalence.

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.

An analysis of adding `seq` to a lazy functional programming language and its consequences for the semantics are investigated in depth 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 two letrec-calculi L_{LR}, L_{name} and the extension L_{lcc} of Abramsky's lazy lambda calculus with `case`, constructors, and `seq`. In Sect. 4 we show that for so-called "convergence admissible" calculi an alternative inductive characterization of bisimilarity is possible. We then use Howe's method in L_{lcc} to show that contextual approximation and 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 that bisimilarity and contextual equivalence are the same in the call-by-need calculus with letrec. In Sect. 8 we discuss that the `seq`-operator is necessary as a primitive of our language, since adding `seq` is not a conservative extension. In Sect. 9 we show that the simpler call-by-need calculus L_{need} is isomorphic to L_{LR}. Finally, we conclude in Sect. 10.

2 Common Notions and Notations for Calculi

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

Definition 2.1. *An untyped deterministic* calculus $\mathcal{D}$ *(DC, for short) is a four-tuple* $(\mathbb{E}, \mathbb{C}, \to, \mathbb{A})$*, where* $\mathbb{E}$ *are expressions,* $\mathbb{C} : \mathbb{E} \to \mathbb{E}$ *is a set of functions (which usually represents contexts),* $\to$ *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*. The* transitive closure *of* $\to$ *is denoted as* $\xrightarrow{+}$ *and the* transitive and reflexive closure *of* $\to$ *is denoted as* $\xrightarrow{*}$*. The notation* $\xrightarrow{0 \vee 1}$ *means one or no reduction, and* $\xrightarrow{k}$ *means* k *reductions. Given an expression* s*, a sequence* $s \to s_1 \to \ldots \to s_n$ *is called a* reduction sequence*; it is called an* evaluation *if* s_n *is an answer,* i.e. $s_n \in W$*; in this case we say* s converges *and denote this as* $s{\downarrow}s_n$ *or as* $s{\downarrow}$ *if* s_n *is not important. If there is no* s_n s.t. $s{\downarrow}s_n$ *then* s diverges*, denoted as* $s{\Uparrow}$*. When dealing with multiple calculi, we often use the calculus name to mark its expressions and relations,* e.g. $\xrightarrow{\mathcal{D}}$ *denotes a reduction relation in* $\mathcal{D}$*.*

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

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

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

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.

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

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

The behavioral preorder $\leq_{b,\mathcal{Q}}$*, called* $\mathcal{Q}$-similarity*, is defined as the greatest fixed point of* $F_{\mathcal{Q}}$*.*

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

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

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

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

Definition 2.5. *[[SSNSS08,SSNSS09]] A* translation $\tau : (\mathbb{E}_1, \mathbb{C}_1, \to_1, \mathbb{A}_1) \to (\mathbb{E}_2, \mathbb{C}_2, \to_2, \mathbb{A}_2)$ *is a mapping $\tau_E : \mathbb{E}_1 \to \mathbb{E}_2$ and a mapping $\tau_C : \mathbb{C}_1 \to \mathbb{C}_2$ 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 $\tau/\sim : \mathbb{E}_1/\sim \to \mathbb{E}_2/\sim$.*

Note that isomorphism means an order-isomorphism between the term-models, where the orders are $\overline{\leq_1}$ and $\overline{\leq_2}$.

Proposition 2.6. *([SSNSS08,SSNSS09]) If a translation $\tau : (\mathbb{E}_1, \mathbb{C}_1, \to_1, \mathbb{A}_1) \to (\mathbb{E}_2, \mathbb{C}_2, \to_2, \mathbb{A}_2)$ 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 present the three calculi: L_{LR}, L_{name}, and L_{lcc}, that we use in the paper: there is the call-by-need calculus, L_{LR} with letrec, and two call-by-name calculi: L_{name} with letrec, and L_{lcc} without letrec. The first two calculi have the same syntax, but differences in their reduction strategies, and the last one is without `letrec`.

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 $(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})$.

Another common construct of all calculi is a case-expression. It is constructed as

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

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 $(c_{T,i}\ x_{i,1}\ \ldots\ x_{i,\mathrm{ar}(c_{T,i})} \to s_i)$ for every constructor $c_{T,i}$ of type T. In a case-alternative $(c_{T,i}\ x_{i,1}\ \ldots\ x_{i,\mathrm{ar}(c_{T,i})} \to s_i)$ we call $c_{T,i}\ x_{i,1}\ \ldots\ x_{i,\mathrm{ar}(c_{T,i})}$ the *pattern* and s_i the right hand side of the alternative. We assume that all variables in `case`-patterns are pairwise distinct, *i.e.* every two patterns have non-overlapping sets of variables. 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 $\texttt{if}\ s_1\ \texttt{then}\ s_2\ \texttt{then}\ s_3$ for the case expression $(\texttt{case}_{Bool}\ s_1\ \texttt{of}\ (\texttt{True} \to s_2)\ (\texttt{False} \to s_3))$.

We now define the syntax of expression 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 $g, h : \mathbb{N} \to \mathbb{N}$,* e.g.*, $\{x_i = s_{i-1}\}_{i=m}^{n}$ means the bindings $x_m = s_{m-1}, x_{m+1} = s_m, \ldots x_n = s_{n-1}$. We assume that 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 $1 \leq i \leq n$.*

With $\mathbb{C}_{\mathcal{L}}$ we denote 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 7.9 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 syntactic proofs of correctness *w.r.t.* contextual equivalence of lots of transformations, whereas the simpler calculus L_{need} (see Section 9) resists syntactical correctness proofs using diagrams. 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. 1, where the labels S, V are used for the exact definition of the normal-order reduction below. Several reduction rules are denoted by their name prefix: 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. 2, which detects the position to which a reduction rule is applied according to the normal order. It uses the following labels: S (subterm), T (top term), V (visited), 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 has as input an expression s and then exhaustively applies the rules in Fig. 2 to s^T, where no other subexpression in s is labeled. The label T is used to prevent the labeling algorithm from walking into `letrec`-environments that are not at the 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.

(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) $\texttt{letrec}\ x_1 = (\lambda x.s)^S, \{x_i = x_{i-1}\}_{i=2}^m, Env, y = C[x_m^V]\ \texttt{in}\ t$
$\to \texttt{letrec}\ x_1 = (\lambda x.s), \{x_i = x_{i-1}\}_{i=2}^m, Env, y = C[(\lambda x.s)]\ \texttt{in}\ t$

(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) $\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s)^S\ \texttt{in}\ t \to \texttt{letrec}\ Env_1, Env_2, x = s\ \texttt{in}\ t$

(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

(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. 1. 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. 2. Labeling algorithm for L_{LR}

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. 2 is applied to s. If the labeling algorithm terminates successfully, then one of the rules in Fig. 1 is applied, if possible, where the labels S, V must match the labels in the expression s (again t may have more labels). The* normal order redex *is defined as the left-hand side of the applied reduction rule. The notation for a normal-order reduction that applies the rule a is $\xrightarrow{LR,a}$,* 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. 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 to copy 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. The rules (lapp), (lcase), and (lseq)

are used to move `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.

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.*

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 $\lambda x.s$, 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), respectively, if 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. Note that black holes, *i.e.* expressions with cyclic dependencies in a normal order reduction context, diverge, *e.g.* $\texttt{letrec}\ x = x\ \texttt{in}\ x$. Other diverging expressions without an infinite evaluation are open expressions where a free variable appears (perhaps after several reductions) in reduction position, or expressions which are dynamically untyped, *i.e.* expressions which reduce to an expression of the form $R[((c\ s_1 \ldots s_{\mathrm{ar}(c)})\ t)]$ or of the form $R[\texttt{case}_T\ (c\ s_1 \ldots s_{\mathrm{ar}(c)})\ \texttt{of}\ alts]$ where R is a reduction context and c does not belong to type T.

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 only 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.

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

$$
\begin{array}{|ll|}
\hline
(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ s)^X & \rightsquigarrow (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ s^X), \text{ if } X \text{ is } S \text{ or } T \\
(s\ t)^{S\vee T} & \rightsquigarrow (s^S\ t) \\
(\texttt{seq}\ s\ t)^{S\vee T} & \rightsquigarrow (\texttt{seq}\ s^S\ t) \\
(\texttt{case}_T\ s\ \texttt{of}\ \mathit{alts})^{S\vee T} & \rightsquigarrow (\texttt{case}_T\ s^S\ \texttt{of}\ \mathit{alts}) \\
\hline
\end{array}
$$

Fig. 3. Labeling algorithm for L_{name}

$$
\begin{array}{|ll|}
\hline
\text{(seq-c)} & C[(\texttt{seq}\ v^S\ s)] \to C[s] \qquad \text{if } v \text{ is a value} \\
\text{(lapp)} & C[((\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ s)^S\ t)] \to C[(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (s\ t))] \\
\text{(lcase)} & C[(\texttt{case}_T\ (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ s)^S\ \texttt{of}\ \mathit{alts})] \\
 & \quad \to C[(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (\texttt{case}_T\ s\ \texttt{of}\ \mathit{alts}))] \\
\text{(lseq)} & C[(\texttt{seq}\ (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ s)^S\ t)] \to C[(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (\texttt{seq}\ s\ t))] \\
\text{(beta)} & C[(\lambda x.s)^S\ t] \to C[s[t/x]] \\
\text{(gcp)} & C_1[\texttt{letrec}\ \mathit{Env},\ x = s\ \texttt{in}\ C_2[x^{S\vee T}]] \to C_1[\texttt{letrec}\ \mathit{Env},\ x = s\ \texttt{in}\ C_2[s]] \\
\text{(case)} & C[(\texttt{case}_T\ (c\ s_1 \ldots s_{ar(c)})^S\ \texttt{of} \ldots ((c\ x_1 \ldots x_{ar(c)}) \to t) \ldots)] \\
 & \quad \to C[t[s_1/x_1, \ldots, s_{ar(c)}/x_{ar(c)}]] \\
\hline
\end{array}
$$

Fig. 4. Normal order reduction rules $\xrightarrow{name}$ of L_{name}

not implement a sharing strategy but instead performs the usual call-by-name beta-reduction and copies arbitrary expressions directly into needed positions.

In Fig. 3 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 3.7. 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:*

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

Normal order reduction $\xrightarrow{name}$ of L_{name} is defined by the following rules shown in Fig. 4 where the labeling algorithm according to Fig. 3 must be applied first. Note that the rules (seq-c), (lapp), (lcase), and (lseq) are identical to the rules for L_{LR} (in Fig. 1), 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 $L[(c\ s_1\ \ldots\ s_{ar(c)})]$ 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 $(\mathbb{E}_{\mathcal{L}}, \mathbb{C}_{\mathcal{L}}, \xrightarrow{name}, \mathbb{A}_{name})$.

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}. Compared to the syntax of 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`:

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

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 show in Fig. 5, and thus the calculus L_{lcc} is defined by the tuple $(\mathbb{E}_\lambda, \mathbb{C}_\lambda, \xrightarrow{lcc}, \mathbb{A}_{lcc})$.

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

Fig. 5. 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 (DC, 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 calculi employing only beta-reduction, applicative similarity can be equivalently defined as $s \leq_b t$, iff for all $n \geq 0$ and closed expressions $r_i, i = 1, \ldots, n$, the implication $(s\ r_1 \ldots r_n)\downarrow \implies (t\ r_1 \ldots r_n)\downarrow$ holds, provided the calculus is convergence-admissible, which means that for all r: $(s\ r)\downarrow v \iff \exists v' : s\downarrow v' \wedge (v'\ r)\downarrow v$.

This can be applied to calculi that also have 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 $(\mathbb{E}, \mathbb{C}, \to, \mathbb{A})$ to be an untyped deterministic calculus and $\mathcal{Q} \subseteq \mathbb{C}$ be a set of functions on expressions. Note that the relations $\leq_{b,\mathcal{Q}}$ and $\leq_{\mathcal{Q}}$ are defined in Definitions 2.3 and 2.4, respectively.

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

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

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

Lemma 4.2. *The operator $F_{\mathcal{Q}}$ is monotonous* w.r.t. *set inclusion,* i.e. *for all binary relations η_1, η_2 on expressions $\eta_1 \subseteq \eta_2 \implies F_{\mathcal{Q}}(\eta_1) \subseteq F_{\mathcal{Q}}(\eta_2)$.*

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

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

Proposition 4.3. *$F_{\mathcal{Q}}$ 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 $\mathrm{glb}(C) = \bigcap_{i=1}^{\infty} \eta_i$, 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 $F_{\mathcal{Q}}(\mathrm{glb}(C)) \subseteq \bigcap_{i=1}^{\infty} F_{\mathcal{Q}}(\eta_i)$, *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: $\forall i : s_1{\downarrow}v_1 \implies (s_2{\downarrow}v_2 \wedge \forall Q \in \mathcal{Q} : Q(v_1)\,\eta_i\,Q(v_2))$. Now we can move the universal quantifier for i inside the formula: $s_1{\downarrow}v_1 \implies (s_2{\downarrow}v_2 \wedge \forall Q \in \mathcal{Q} : \forall \mathrm{i}\text{:}\ Q(v_1)\,\eta_i\,Q(v_2))$. This is equivalent to $s_1{\downarrow}v_1 \implies (s_2{\downarrow}v_2 \wedge \forall Q \in \mathcal{Q} : Q(v_1)\,(\bigcap_{i=1}^{\infty} \eta_i)\,Q(v_2))$ or $s_1{\downarrow}v_1 \implies (s_2{\downarrow}v_2 \wedge \forall Q \in \mathcal{Q} : (Q(v_1), Q(v_2)) \in \mathrm{glb}(C))$ and thus $(s_1, s_2) \in F_{\mathcal{Q}}(\mathrm{glb}(C))$.

Definition 4.4. *Let $\leq_{b,\mathcal{Q},i}$ for $i \in \mathbb{N}_0$ be defined as follows:*

$$\leq_{b,\mathcal{Q},0} = \mathbb{E} \times \mathbb{E} \quad \textit{and} \quad \leq_{b,i} = F_{\mathcal{Q}}(\leq_{b,\mathcal{Q},i-1}), \textit{if } i > 0$$

Theorem 4.5. $\leq_{b,\mathcal{Q}} = \bigcap_{i=1}^{\infty} \leq_{b,\mathcal{Q},i}$

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

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

Definition 4.6. *An untyped deterministic calculus $(\mathbb{E}, \mathbb{C}, \to, \mathbb{A})$ is* convergence-admissible *w.r.t. $\mathcal{Q}$ if, and only if $\forall Q \in \mathcal{Q}, s \in \mathbb{E} : Q(s){\downarrow}v \iff \exists v' : s{\downarrow}v' \wedge Q(v'){\downarrow}v$*

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

Lemma 4.7. *Let $(\mathbb{E}, \mathbb{C}, \to, \mathbb{A})$ be convergence-admissible* w.r.t. $\mathcal{Q}$. *Then the following holds:*

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

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

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

Lemma 4.8. *Let $(\mathbb{E}, \mathbb{C}, \to, \mathbb{A})$ be convergence-admissible* w.r.t. $\mathcal{Q}$. *Then for all $s_1, s_2 \in E : s_1 \leq_{b,\mathcal{Q}} s_2 \implies Q(s_1) \leq_{b,\mathcal{Q}} Q(s_2)$ for all $Q \in \mathcal{Q}$*

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

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

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

Theorem 4.9. *Let* $(\mathbb{E}, \mathbb{C}, \to, \mathbb{A})$ *be convergence-admissible* w.r.t. $\mathcal{Q}$. *Then* $\leq_{\mathcal{Q}} = \leq_{b,\mathcal{Q}}$.

Proof. "$\subseteq$": Let $s_1 \leq_{\mathcal{Q}} s_2$. We use Theorem 4.5 and show $s_1 \leq_{b,\mathcal{Q},i} s_2$ for all i. We use induction on i. The base case ($i = 0$) obviously holds. Let $i > 0$ and let $s_1\downarrow v_1$. Then $s_1 \leq_{\mathcal{Q}} s_2$ implies $s_2\downarrow v_2$. Thus, it is sufficient to show that $Q(v_1) \leq_{b,\mathcal{Q},i-1} Q(v_2)$ for all $Q \in \mathcal{Q}$: As induction hypothesis we use that $s_1 \leq_{\mathcal{Q}} s_2 \implies s_1 \leq_{b,\mathcal{Q},i-1} s_2$ holds. Using Lemma 4.7 twice and $s_1 \leq_{\mathcal{Q}} s_2$, we have $Q(v_1) \leq_{\mathcal{Q}} Q(v_2)$. The induction hypothesis shows that $Q(v_1) \leq_{b,\mathcal{Q},i-1} Q(v_2)$. Now the definition of $\leq_{b,\mathcal{Q},i}$ is satisfied, which shows $s_1 \leq_{b,\mathcal{Q},i} s_2$.

"$\supseteq$": Let $s_1 \leq_{b,\mathcal{Q}} s_2$. By induction on the number n of observers we show $\forall n, Q_i \in \mathcal{Q} : Q_1(\ldots(Q_n(s_1)))\downarrow \implies Q_1(\ldots(Q_n(s_2)))\downarrow$. The base case follows from $s_1 \leq_{b,\mathcal{Q}} s_2$. For the induction step we use the following induction hypothesis: $t_1 \leq_{b,\mathcal{Q}} t_2 \implies \forall j < n, Q_i \in \mathcal{Q} : Q_1(\ldots(Q_j(t_1)))\downarrow \implies Q_1(\ldots(Q_j(t_2)))\downarrow$ for all t_1, t_2. Let $Q_1(\ldots(Q_n(s_1)))\downarrow$. From Lemma 4.8 we have $r_1 \leq_{b,\mathcal{Q}} r_2$, where $r_i = Q_n(s_i)$. Now the induction hypothesis shows that $Q_1(\ldots(Q_{n-1}(r_1)))\downarrow \implies Q_1(\ldots(Q_{n-1}(r_2)))\downarrow$ and thus $Q_1(\ldots(Q_n(s_2)))\downarrow$.

4.2 Bisimulation in L_{lcc}

We define a (standard) applicative similarity $\leq_{b,lcc}$ in L_{lcc} analogous to [How89,How96]. We then show that similarity in L_{lcc} is equivalent to contextual preorder, and also give further characterizations of similarity. These characterizations will allow us to lift the properties of bisimilarity to the calculi L_{name} and L_{LR}.

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 for all closed* $r \in \mathbb{E}_\lambda$ *the relation* $s'[r/x]\ \eta\ t'[r/x]$ *holds*$\big)$
- $s\downarrow_{lcc}(c\ s_1' \ldots s_n') \implies \big(\ t\downarrow_{lcc}(c\ t_1' \ldots t_n')$ *and the relation* $s_i'\ \eta\ t_i'$ *holds for all* $i\big)$

Similarity $\leq_{b,lcc}$ *is defined as the greatest fixpoint of the operator* F_{lcc}.

The definition of $\leq_{b,lcc}$ can also be viewed as instantiation of $\leq_{b,\mathcal{Q}}$ where the set $\mathcal{Q}$ are *not* contexts, but functions on terms. We will later define a relation $\leq_{b,\mathcal{Q}}$ (in Definition 4.33) which is equivalent $\leq_{b,lcc}$, but where $\mathcal{Q}$ is a set of contexts. Note that the operator F_{lcc} is monotone, hence the greatest fixpoint $\leq_{b,lcc}$ exists.

Definition 4.11. *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.*

4.2.1 Bisimilarity and Contextual Equivalence Coincide in L_{lcc} Although it is rather standard, for the sake of completeness we will show in this section that $\leq^o_{b,lcc}$ and $\leq_{lcc}$ coincide using Howe's method [How89,How96]. 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. 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 represent also the binding λ using $\lambda(x.s)$ as representing $\lambda x.s$. In order to stick to terms, and be consistent with the treatment in other papers like [How89], we assume that removing the top constructor $\lambda x.$ 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.

Lemma 4.12. *For a relation η on closed expressions 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 the $\mu\ \subseteq ((\mu)^c)^o$ is equivalent to $s\ \mu\ t\ \Longrightarrow\ \sigma(s)\ (\mu)^c\ \sigma(t)$ for all closing substitutions σ.*

Proposition 4.13 (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 $\eta\ \subseteq \leq_{b,lcc}$. This obviously also implies $(\eta)^o\ \subseteq\ (\leq_{b,lcc})^o$.*

The fixpoint property of $\leq_{b,lcc}$ implies:

Lemma 4.14. *For closed values $\theta(s_1 \ldots s_n), \theta(t_1 \ldots t_n)$, we have $\theta(s_1 \ldots s_n)\ \leq_{b,lcc}\ \theta(t_1 \ldots t_n)$ iff $s_i\ \leq^o_{b,lcc}\ t_i$. In the concrete syntax, if θ is a constructor, then $\theta(s_1 \ldots s_n)\ \leq_{b,lcc}\ \theta(t_1 \ldots t_n)$ iff $s_i\ \leq_{b,lcc}\ t_i$, and $\lambda x.s \leq_{b,lcc} \lambda x.t$ iff $s \leq^o_{b,lcc} t$.*

Lemma 4.15. *The relations $\leq_{b,lcc}$ and $\leq^o_{b,lcc}$ are reflexive and transitive.*

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

The goal in the following is to show that $\leq_{b,lcc}$ is a precongruence. This proof proceeds by defining a congruence candidate $\leq_{cand}$ as a closure of $\leq_{b,lcc}$ within contexts, which obviously is operator respecting. This relation is not known to be transitive. Then we show that $\leq_{b,lcc}$ and $\leq_{cand}$ coincide.

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

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 $x \leq^o_{b,lcc} s$.*
2. *$\tau(s_1,\ldots,s_n) \leq_{cand} s$ iff there is some expression $\tau(s'_1,\ldots,s'_n) \leq^o_{b,lcc} s$ 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. *$\leq^o_{b,lcc} \subseteq\ \leq_{cand}$ and $\leq_{b,lcc} \subseteq (\leq_{cand})^c$.*
4. *$\leq_{cand} \circ \leq^o_{b,lcc}\ \subseteq\ \leq_{cand}$.*
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 $\leq^o_b$. Part (4) follows from the definition, Lemma 4.18, and transitivity of $\leq^o_{b,lcc}$.

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 $x \leq^o_{b,lcc} r'$ from the definition, and $s' \leq^o_{b,lcc} r'[s'/x]$ and thus $x[s/x] \leq_{cand} r'[s'/x]$. In the case $y \leq_{cand} r$, we obtain $y \leq^o_{b,lcc} r'$ from the definition, and $y[s/x] = y \leq^o_{b,lcc} r'[s'/x]$ and thus $y = y[s/x] \leq_{cand} r'[s'/x]$. If $r = \tau(r_1,\ldots,r_n)$ and $r \leq_{cand} r'$ and $r[s/x]\ \eta\ r'[s'/x]$. Then there is some $\tau(r'_1,\ldots,r'_n) \leq^o_{b,lcc} r'$ 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 $\tau(r'_1,\ldots,r'_n)[s'/x] \leq^o_{b,lcc} r'[s'/x]$. Thus $r[s/x]\ F_{cand}(\eta)\ r'[s'/x]$.

Part (6) follows from item (5). Part (7) follows from item (6) and Lemma 4.12. □

Lemma 4.20. *The middle expression in the definition of* $\leq_{cand}$ *can be chosen as 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* $\tau(s'_1,\ldots,s'_{\mathrm{ar}(\tau)}) \leq^o_{b,lcc} s$*.*

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 $\tau(s''_1,\ldots,s''_{\mathrm{ar}(\tau)}) \leq^o_{b,lcc} t$. 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 $\sigma(\tau(s''_1,\ldots,s''_{\mathrm{ar}(\tau)})) \leq^o_{b,lcc} t$ 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* $s' \leq^o_{b,lcc} s$ *and* $s \leq^o_{b,lcc} s'$*.*

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 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 $\theta(t'_1,\ldots,t'_n) \leq_{b,lcc} t$. Consider the case $s = \lambda x.s'$. Then there is some closed $\lambda x.t' \leq_{b,lcc} t$ with $s' \leq_{cand} t'$. The relation $\lambda x.t' \leq_{b,lcc} t$ implies that $t \xrightarrow{lcc,*} \lambda x.t''$. Lemma 4.21 now implies $\lambda x.s' \leq_{cand} \lambda x.t''$. Definition of $\leq_{cand}$ and Lemma 4.20 now show that there is some closed $\lambda x.t^{(3)}$ with $s' \leq_{cand} t'''$ and $\lambda x.t''' \leq_{b,lcc} \lambda x.t''$. The latter relation implies $t''' \leq^o_{b,lcc} t''$, which shows $s' \leq_{cand} t''$ by Lemma 4.19 (4).

If θ is a constructor, then there is a closed expression $\theta(t'_1,\ldots,t'_n)$ with $s_i \leq_{cand} t'_i$ for all i and $\theta(t'_1,\ldots,t'_n) \leq_{b,lcc} t$. The definition of $\leq_{b,lcc}$ implies that $t \xrightarrow{lcc,*} \theta(t''_1,\ldots,t''_n)$ with $t'_i \leq_{b,lcc} t''_i$ for all i. By definition of $\leq_{cand}$, 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 $s = \tau(s_1,\ldots,s_n)$ and that there is some closed $t' = \tau(t'_1,\ldots,t'_n)$ with $s_i \leq_{cand} t'_i$ for all i and $t' \leq^o_{b,lcc} t$.

- For the (nbeta)-reduction, $s = (s_1\ s_2)$, where $s_1 = (\lambda x.s'_1)$, s_2 is a closed term, and $t' = t'_1\ t'_2$. Lemma 4.23 and $s_1 \leq_{cand} t'_1$ show that $t'_1 \xrightarrow{lcc,*} \lambda x.t''_1$ with $\lambda x.s'_1 \leq_{cand} \lambda x.t''_1$ and also $s'_1 \leq_{cand} t''_1$. From $t' \xrightarrow{lcc,*} t''_1[t'_2/x]$ we obtain $t''_1[t'_2/x] \leq_{b,lcc} t$. Lemma 4.19 now shows $s'_1[s_2/x] \leq_{cand} t''_1[t'_2/x]$. Hence $s'_1[s_2/x] \leq_{cand} t$, again using Lemma 4.19.

- Similar arguments apply to the case-reduction.
- Suppose, the reduction is a (nseq)-reduction. Then $s \leq_{cand} t$ and $s = (\texttt{seq}\ s_1\ s_2)$. Lemma 4.20 implies that there is some closed $(\texttt{seq}\ t_1'\ t_2') \leq^o_{b,lcc} t$ 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 $(\texttt{seq}\ t_1'\ t_2') \xrightarrow{lcc,*} (\texttt{seq}\ t_1''\ t_2') \xrightarrow{lcc} t_2'$. Since $t_2' \leq^o_{b,lcc} (\texttt{seq}\ t_1'\ t_2') \leq^o_{b,lcc} t$, 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 hole. The base case is proved 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 $R[s] = \tau(s_1, \ldots, s_n)$ and that there is some closed $t' = \tau(t_1', \ldots, t_n') \leq^o_{b,lcc} t$ 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 obtain also $R[s'] = \tau(s_1, \ldots, s_{j-1}, s_j', s_{j+1}, \ldots, s_n) \leq_{cand} \tau(t_1', \ldots, t_{j-1}', t_j', t_{j+1}', \ldots, t_n')$, and from $\tau(t_1', \ldots, t_n') \leq^o_{b,lcc} t$, also $R[s'] = \tau(s_1, \ldots, s_{j-1}, s_j', s_{j+1}, \ldots, s_n) \leq_{cand} t$.

Now we are ready to prove that the precongruence candidate and similarity coincide. First we prove this for the relations on closed expressions and then consider (possibly) open expressions.

Theorem 4.26. $(\leq_{cand})^c = \leq_{b,lcc}$.

Proof. Since $\leq_{b,lcc} \subseteq (\leq_{cand})^c$ by Lemma 4.19, we have to show that $(\leq_{cand})^c \subseteq \leq_{b,lcc}$. Therefore it is sufficient to show that $(\leq_{cand})^c$ satisfies the fixpoint equation for $\leq_{b,lcc}$. 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 $\neg(s{\downarrow_{lcc}})$, then $s\ F_{lcc}((\leq_{cand})^c)\ t$ holds by Lemma 4.19. If $s{\downarrow_{lcc}}\theta(s_1, \ldots, s_n)$, then $\theta(s_1, \ldots, s_n)\ (\leq_{cand})^c\ t$ by Lemma 4.25. Lemma 4.23 shows that $t \xrightarrow{lcc,*} \theta(t_1, \ldots, t_n)$ and for all $i : s_i \leq_{cand} t_i$. This implies $s\ F_{lcc}((\leq_{cand})^c)\ t$, since $\theta(t_1, \ldots, t_n) \leq^o_{b,lcc} t$. We have proved the fixpoint property of $(\leq_{cand})^c$ *w.r.t.* F_{lcc}, and hence $(\leq_{cand})^c = \leq_{b,lcc}$.

Theorem 4.27. $\leq_{cand} = \leq^o_{b,lcc}$.

Proof. Theorem 4.26 shows $(\leq_{cand})^c \subseteq \leq_{b,lcc}$. Hence $((\leq_{cand})^c)^o \subseteq \leq^o_{b,lcc}$ by monotonicity. Lemma 4.19 (7) implies $\leq_{cand} \subseteq ((\leq_{cand})^c)^o \subseteq \leq^o_{b,lcc}$.

This immediately implies:

Corollary 4.28. *$\leq^o_{b,lcc}$ is a precongruence on expressions $\mathbb{E}_\lambda$. If σ is a substitution, then $s \leq^o_{b,lcc} t$ implies $\sigma(s) \leq^o_{b,lcc} \sigma(t)$.*

Lemma 4.29. $\leq^o_{b,lcc} \subseteq \leq_{lcc}$.

Proof. Let s, t be expressions with $s \leq^o_{b,lcc} t$ 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 $\sigma(s) \leq^o_{b,lcc} \sigma(t)$, we obtain from the precongruence property of $\leq^o_{b,lcc}$ that also $\sigma(C[s]) \leq_{b,lcc} \sigma(C[t])$. 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 $\leq^o_{b,lcc} \subseteq \leq_{lcc}$.

Lemma 4.30. $\leq_{lcc} \subseteq \leq^o_{b,lcc}$

Proof. We show that $\leq^c_{lcc}$ satisfies the fixpoint condition, *i.e.* $\leq^c_{lcc} \subseteq F_{lcc}(\leq^c_{lcc})$: Let s, t be closed and $s \leq_{lcc} t$. If $s{\downarrow_{lcc}}\theta(s_1, \ldots, s_n)$, then also $t{\downarrow_{lcc}}$. Using the appropriate `case`-expressions as contexts, it is easy to see that $t{\downarrow_{lcc}}\theta(t_1, \ldots, t_n)$. Now we have to show that $s_i \leq^o_{lcc} t_i$. This could be done using an appropriate context C_i that selects the components, *i.e.* $C_i[s] \xrightarrow{lcc,*} s_i$ and $C_i[t] \xrightarrow{lcc,*} t_i$ and Lemmas 4.21 and 4.29 show that $r \xrightarrow{lcc} r'$ implies $r \leq_{lcc} r'$ holds. Moreover, since $\leq^o_{lcc}$ is obviously a precongruence, we obtain that $s_i \leq^o_{lcc} t_i$. Thus the proof is finished.

Lemmas 4.29 and 4.30 immediately imply:

Proposition 4.31. $\leq^o_{b,lcc} = \leq_{lcc}$.

Corollary 4.32. *The reduction rules of the calculus L_{lcc} are correct in any context.*

Proof. It immediately follows for closed redexes. For open redexes, $\leq_{lcc}$ is equivalent to the open extension of $\leq_{b,lcc}$, hence correctness follows, since $\leq_{lcc}$-equality holds under all closing substitutions.

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 identical to contextual equivalence.

Using Theorem 4.9 we show that in L_{lcc}, behavioral equivalence can also be proved inductively:

Definition 4.33. *The set $\mathcal{Q}$ 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 $\leq_{b,\mathcal{Q}}$, $\leq_{\mathcal{Q}}$ on closed $\mathbb{E}_\lambda$-expressions are defined as in Definition 2.3 and Definition 2.4, respectively.

Lemma 4.34. *The calculus L_{lcc} is convergence-admissible in the sense of Definition 4.6, 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 $Q[v]$.

Corollary 4.35. $\leq_{b,lcc} = \leq_{b,\mathcal{Q}} = \leq_{\mathcal{Q}}$.

Proof. $\leq_{b,lcc} = \leq_{b,\mathcal{Q}}$, since the definitions are the same with the exception of the (ncase)-reductions which are required according to Definition 4.33 of $\leq_{b,\mathcal{Q}}$, where the context in (ii) performs a projection of a constructor application to a component, and the context (iii) performs a test for the top constructor. Nevertheless, the definitions are equivalent.

Theorem 4.9. shows $\leq_{b,\mathcal{Q}} = \leq_{\mathcal{Q}}$ since L_{lcc} is convergence-admissible by Lemma 4.34.

Definition 4.36. *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:*

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

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

The set $\mathcal{Q}_{CE}$ is defined like the set $\mathcal{Q}$ in Definition 4.33, but only expressions r from CE_{lcc} are taken into account in the contexts $([\cdot]\ r)$ in (i).

Theorem 4.37. *In L_{lcc}, all the following relations on open $\mathbb{E}_\lambda$-expressions are identical:*

1. $\leq_{lcc}$.
2. $\leq^o_{b,lcc}$.
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 $M[\cdot,\dots,\cdot]$ and all substitutions σ: $M[\sigma(s_1),\dots,\sigma(s_1)]{\downarrow_{lcc}} \implies M[\sigma(s_2),\dots,\sigma(s_2)]{\downarrow_{lcc}}$.*
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. *The relation $\leq^o_{b,lcc,4}$ where $\leq_{b,lcc,4}$ is defined using the Kleene-construction,* i.e.
   $\leq_{b,lcc,1} = \bigcap_{i\geq 0} \leq'_{b,i}$, *where $\leq'_{b,0}$ is the full relation, and $\leq'_{b,i+1} := F_{lcc}(\leq'_{b,i})$ for all $i > 0$.*

8. *The relation $\leq^o_{\mathcal{Q}}$ where $\leq_{\mathcal{Q}}$ is as defined on closed $\mathbb{E}_\lambda$-expressions in Definition 2.4 for the set $\mathcal{Q}$ in Definition 4.33.*
9. *The relation $\leq^o_{\mathcal{Q}_{CE}}$ where $\leq_{\mathcal{Q}_{CE}}$ is as defined on closed $\mathbb{E}_\lambda$-expressions in Definition 2.4 for the set $\mathcal{Q}_{CE}$ in Definition 4.36.*
10. *The relation $\leq^o_{b,\mathcal{Q}_{CE}}$ as defined on closed $\mathbb{E}_\lambda$-expressions in Definition 2.3 for the set $\mathcal{Q}_{CE}$ in Definition 4.36.*

Proof. – (1) $\iff$ (2): This is Lemma 4.31.

– (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 every reduction in the sequence $C[s_1] \xrightarrow{lcc,*} v$ cannot 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 every term in this sequence cannot be of the form $R[\Omega]$ where R is a reduction context, since otherwise the reduction would not terminate (since $R[\Omega] \xrightarrow{lcc,+} R[\Omega]$). 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 4.32) that (nbeta) is correct for $\sim_{lcc}$.

– (1) $\iff$ (4): This follows from Corollary 4.32 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) One direction 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) $\iff$ (2): This follows from Theorem 4.5, since the L_{lcc} is a DC for the set $\mathcal{Q}$ of contexts, since the reduction rules in L_{lcc} are correct, and since the convergence test in nested $\mathcal{Q}$-contexts is as strong as the relations $\leq_{b,lcc}$.
- (8) $\iff$ (2): This follows for the relations on closed $\mathbb{E}_\lambda$-expressions by Theorem 4.9, since the DC $\mathcal{Q}$ for L_{lcc} described before is convergence admissible. It also holds for the extensions to open expressions, since the construction for the open extension is identical for both relations.
- (8) $\iff$ (9): The direction (8) $\implies$ (9) is trivial. For the other direction we show that $\leq_{b,lcc,3} \subseteq \leq_{b,lcc,2}$: let $s_1 \leq_{b,lcc,3} s_2$ and let $Q_1[\ldots Q_n[s_1]\ldots]{\downarrow_{lcc}}$ for $Q_i \in \mathcal{Q}$. Let m be the number of reductions of $Q_1[\ldots Q_n[s_2]\ldots]$ to an L_{lcc}-WHNF.

  Since the reduction rules are correct for $\sim_{lcc}$ for every subexpression r of the contexts Q_i, the relation $r \sim_{lcc} r'$ holds where r' is either a closed abstraction, Ω, or a constructor expression that is evaluated to depth at least $m+1$: Every position p of length $m+1$ of r'_i either only hits constructors as labels or there is a prefix p' of p such that only constructors are hit, with the exception of the last one, which may be Ω, or an abstraction. For all the argument subexpressions r, let r''_i be constructed from the expressions r' where at positions p of length $m+1$ that only hit constructors, the expression at depth $m+1$ is replaced by Ω. Accordingly, let Q'_i be the result of reducing the subexpressions of Q_i and let Q''_i be the result of Q'_i Thus $Q_1[\ldots[Q_n[s_1]]] \sim_{lcc} Q'_1[\ldots[Q'_n[s_1]]]$, which implies $(Q'_1[\ldots[Q'_n[s_1]]]){\downarrow_{lcc}}$. Inspecting the reductions one can verify that $Q'_1[\ldots[Q'_n[s_1]]]$ reduces to a WHNF in at most m reduction steps. Since a normal order reduction of length m cannot use the subexpressions at depth $m+1$ (within constructors), we also have $(Q''_1[\ldots[Q''_n[s_1]]]){\downarrow_{lcc}}$. Now $s \leq^o_{b,lcc,3} t$ shows $(Q''_1[\ldots[Q''_n[s_2]]]){\downarrow_{lcc}}$. Since $r''_i \leq_{lcc} r'_i$, we also have $(Q'_1[\ldots[Q'_n[s_2]]]){\downarrow_{lcc}}$. Applying contextual equivalence again for $r_i \sim r'_i$ shows $(Q_1[\ldots[Q_n[s_2]]]){\downarrow_{lcc}}$., *i.e.* $s_1 \leq^o_{b,lcc,2} s_2$.
- (9) $\iff$ (10): This follows for the relations on closed expressions by Theorem 4.9, since the DC for L_{lcc} with $\mathcal{Q}_{CE}$ 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 4.37 and Corollary 4.32.

Proposition 4.38. *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 4.39. *Given any two closed $\mathbb{E}_\lambda$-expressions s_1, s_2. Then $s_1 \leq_{lcc} s_2$ iff the following conditions hold:*

- *If $s_1{\downarrow}\lambda x.s'_1$, then $s_2{\downarrow}\lambda x.s'_2$, and for all closed L_{lcc}-abstractions r and for $r = \Omega$: $s_1\ r \leq_{lcc} s_2\ r$*

– *if* $s_1{\downarrow}(c\ t_1 \ldots t_n)$*, then* $s_2{\downarrow}(c\ t'_1 \ldots t'_n)$*, and for all* $i : t_i \leq_{lcc} t'_i$

Proof. The if-direction follows from the congruence property of $\leq_{lcc}$ and the correctness of reductions. The only-if direction follows from Theorem 4.37.

This immediately implies

Proposition 4.40. *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* L_{lcc}*-abstractions* r *and for* $r = \Omega$*:* $s_1\ r \leq_{lcc} s_2\ r$
– *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* $i : t_i \leq_{lcc} t'_i$

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. We will prove that contextual equivalence based on L_{LR}-evaluation and contextual equivalence based on L_{name}-evaluation are equivalent. We will use infinite trees to connect both evaluation strategies. Note that [SS07] already shows a similar result, for a lambda calculus without 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 as syntax 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 coinductively,* 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` in expressions is without any reduction rule. It will represent cyclic bindings that are only via variable bindings like $x = y, y = x$.

In the following definition of a mapping from finite expressions to their infinite image, 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 at 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.

$C[(s\ t)|_\varepsilon] \mapsto @$
$C[(\texttt{case}_T\ \ldots)|_\varepsilon] \mapsto \texttt{case}_T$
$C[(c\ x_1 \ldots\ x_n \to s)|_\varepsilon] \mapsto (c\ x_1\ \ldots\ x_n)$ 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 using rule 10,11,12,13 repeatedly, then the result is `Bot`.

Fig. 6. Infinite tree construction from positions for fixed s

Definition 5.2. *Let $e \in \mathbb{E}_\mathcal{L}$. The translation $IT :: \mathbb{E}_\mathcal{L} \to \mathcal{E}_\mathcal{I}$ translates an expression s into its infinite tree $IT(s)$. 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 it computes the label of $IT(s)$ at position s. The computation starts with $s|_p$ and then proceeds with the rules given in Fig. 6. The first rules define 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.

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

The set $\mathbb{C}_\mathcal{I}$ of infinite tree contexts includes any infinite trees where a subtree (at finite depth) is replaced by the 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 where $S \in \mathcal{E}_{\mathcal{I}}$:*

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

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. 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) $(\texttt{case}_T\ (c\ S_1 \ldots S_n)\ \texttt{of}\ \ldots (c\ x_1 \ldots x_n) \to S') \to S'[S_1/x_1, \ldots, S_n/x_n]$

*If a reduction rule (*betaTr*), (*caseTr*), or (*seqTr*) is applied within an $\mathcal{R}_{tree}$-context, then we say it is a* normal order reduction *(tree-reduction) on infinite trees and write $S \xrightarrow{tree} S'$. We also use the convergence predicate $\downarrow_{tree}$ for infinite trees, which is defined accordingly. The* redex *of a reduction on infinite trees is the (infinite) subtree which is modified by the reduction rule.*

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 M be a set of (perhaps infinitely many) positions of S. These are intended to be the redexes, and can also be viewed as a notation for the labeled redexes. We restrict the sets M in reductions such that only positions of redexes of the same reduction rule are contained.*

By $S \xrightarrow{I,M} S'$ we denote the (perhaps infinite) development top down, i.e. *the reduction sequence constructs a new infinite tree S' top-down by using labeled reduction for every labeled redex, where the label of the redex is removed before the reduction, and the others are inherited. If the reduction does not terminate for a subtree at the top level of the subtree, then this subtree is replaced by the constant* `Bot` *in the result S'. If the subtree is of the form $(op\ S_1 \ldots S_n)$ where op stands for any syntactic construct, and no superexpression carries a reduction label, then for all i let S'_i be the resulting infinite tree for the infinite development of S_i. Then the resulting tree is $(op\ S'_1 \ldots S'_n)$. This recursively defines the resulting tree top-down.*

If the reduction $S \xrightarrow{I,M} S'$ does not contain a normal order tree-redex, 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 tree standard 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 $(\lambda x.(\lambda x.(\ldots)\ a))\ a$, and the $(tree, \text{betaTr})$-reduction-sequence is infinite.

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

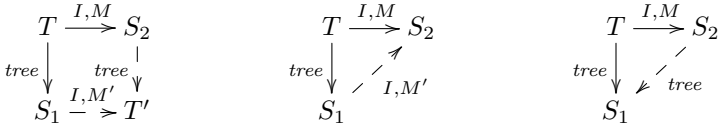
5.2 Standardization of Tree Reduction

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 $T \xrightarrow{I,\neg tree} T'$, where T' is an answer, then T is also an answer.*

Proof. This follows since an answer cannot be generated by $\xrightarrow{I,\neg tree}$-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.*

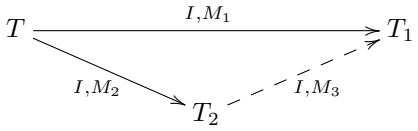


Proof. This follows by checking the overlaps of $\xrightarrow{I}$ with *tree*-reductions. The third diagram applies if the positions of M are removed by the *tree*-reduction. The second diagram applies if the *tree*-redex is included in M and the first diagram is applicable in all other cases.

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

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. □

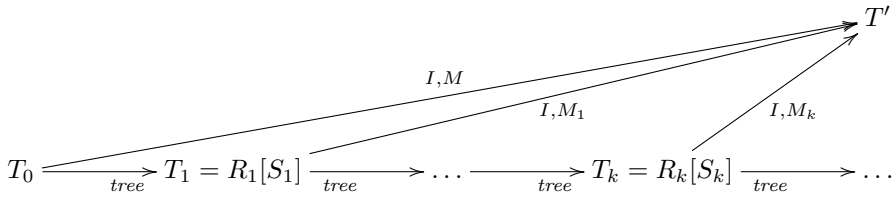
Consider a reduction $T \xrightarrow{I,M} T'$ of type (betaTr), (caseTr) or (seqTr). This reduction may include a redex of a normal order *tree*-reduction. Then the reduction can be split into $T \xrightarrow{tree} T_1 \xrightarrow{I} T'$. This split can be iterated, as long as the remaining $T_1 \xrightarrow{I}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

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 $M_k = M_{k,1} \cup M_{k,2}$, 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 $T' = R'_k[\bot]$. But than 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. □

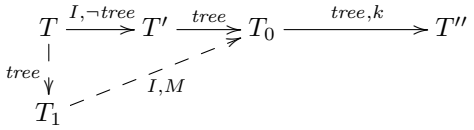
Lemma 5.13. *Let $T \xrightarrow{I,M,\neg tree} T_1 \xrightarrow{tree} T'$. Then the reduction can be commuted to $T \xrightarrow{tree} T_3 \xrightarrow{I,M'} T'$ for some M'.*

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.

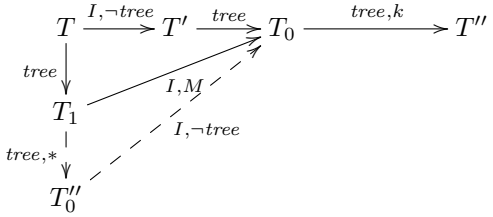
Lemma 5.14. *Let* $T \xrightarrow{I,\neg tree} T'$ *and* $T'{\downarrow_{tree}}$. *Then* $T{\downarrow_{tree}}$.

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:



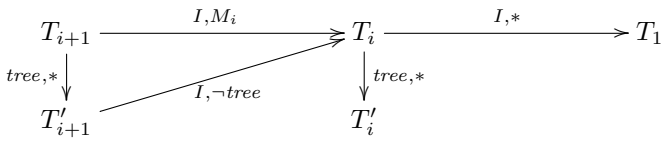
If M does not contain a normal order redex, then the induction hypothesis shows that $T_1{\downarrow_{tree}}$ and thus also $T{\downarrow_{tree}}$. Now assume that M contains a normal order redex. Then we apply Lemma 5.12 to $T_1 \xrightarrow{I,M} T_0$ (note that $T_0{\downarrow_{tree}}$ and hence the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

$$\begin{array}{ccccc} T_{i+1} & \xrightarrow{I,\neg tree} & T_i & \xrightarrow{I,*} & T_1 \\ & & \downarrow{\scriptstyle tree,*} & & \\ & & T'_i & & \end{array}$$

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 $NL[v]$:

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

The base case obviously holds, since v is already an L_{LR}-WHNF. For the induction step assume that $NL[v'] \xrightarrow{LR,\text{cp}\vee\text{llet},*} t$ 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.*

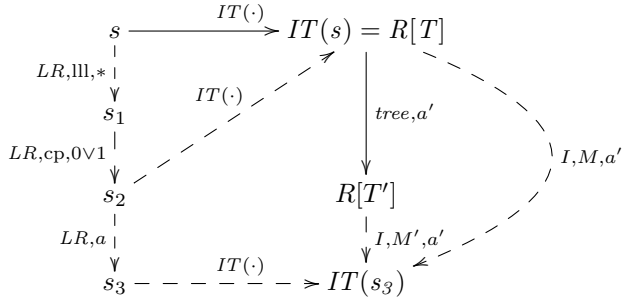
Proof. Only the latter needs a justification. Therefore, we label every redex in $IT(s)$ that is derived from the redex $s \xrightarrow{LR} t$ by $IT(.)$. This results in the set M for $IT(s)$. There will be at least one position in M that is a normal order redex of $IT(s)$.

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 $s \xrightarrow{LR,\text{lll},*} s_1 \xrightarrow{LR,\text{cp},0\vee1} s_2 \xrightarrow{LR,a} s_3$ 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 $s \xrightarrow{LR,\mathrm{lll},*} s'$. 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 $s \xrightarrow{LR,\mathrm{lll},*} s_1$ 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 $IT(s) \xrightarrow{tree,\mathrm{betaTr}} IT(s)$. 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 $s \xrightarrow{LR,\mathrm{lll},*} \xrightarrow{LR,\mathrm{cp},0\vee1} s' \xrightarrow{LR,a} s''$, 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 a ($name$,gcp)-, ($name$,lapp)-, ($name$,lcase)-, or ($name$,lseq)-reduction, 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. 6. 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 is of the form $(10)^*(2\vee4\vee6)^+(10)\ldots$. 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 $(2 \vee 4 \vee 6)$ are interchanged, hence the second component of the measure is strictly decreased.
2. The label computation is of the form $(10)^*(2\vee4\vee6)^*(11)\ldots$. 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 $s \xrightarrow{name,\mathrm{gcp},*} L[v]$ 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 also 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 2.5.*

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 $i = 1, \ldots, n$ can be defined as follows:*

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

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

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

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

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

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:*

- $N(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t) =$
  $(\lambda x_1', \ldots, x_n'.(\lambda x_1, \ldots x_n.N(t)) \ U_1 \ldots \ U_n) \ X_1' \ldots X_n'$
  where $U_i = x_i' \ x_1' \ldots x_n'$,
  $X_i' = \lambda x_1 \ldots x_n.F_i(x_1 \ x_1 \ldots x_n) \ldots (x_n \ x_1 \ldots x_n)$,
  $F_i = \lambda x_1, \ldots, x_n.N(s_i)$.
- $N(s \ t) = (N(s) \ N(t))$
- $N(\texttt{seq}\ s \ t) = (\texttt{seq}\ N(s) \ N(t))$
- $N(c \ s_1 \ \ldots \ s_{\mathrm{ar}(c)}) = (c \ N(s_1) \ldots N(s_{\mathrm{ar}(c)}))$

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

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 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:*

- $N'(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t) = \sigma(N'(t))$ *where*

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

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

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 4.32.

Now 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 can not 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 C. 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 $s = (c\ s_1 \ldots s_{\mathrm{ar}(c)})$ then $t = (c\ t'_1 \ldots t'_{\mathrm{ar}(c)})$.*

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:

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

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

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

– (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

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

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 $X_i' = \lambda x_1 \ldots x_n.F_i(x_1x_1 \ldots x_n) \ldots (x_nx_1 \ldots x_n)$, and $F_i = \lambda x_1, \ldots, x_n.\sigma_L(N'(s_i))$, *i.e.*, $N'(t) = R''[U_1]$.
Performing the applications, we transform U_1 in $2n$ steps as

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

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 $N'(s) \xrightarrow{lcc,\mathrm{nbeta},2n} R''[\sigma_L(N'(s_1))[U_1/x_1,\ldots,U_n/x_n]]$ 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 the diagram is as follows:

$$\begin{array}{ccc} \cdot & \xrightarrow{N'} & \cdot \\ {\scriptstyle name,\mathrm{gcp}}\downarrow & & \downarrow{\scriptstyle lcc,\mathrm{nbeta},2n} \\ \cdot & \xrightarrow{N'} & \cdot \end{array}$$

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

– (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:

$$\begin{array}{ccc} \cdot & \xrightarrow{N'} & \cdot \\ {\scriptstyle name,\mathrm{lapp}}\downarrow & \nearrow_{N'} & \\ \cdot & & \end{array}$$

– (case) The diagram for this case is:

$$\begin{array}{ccc} \cdot & \xrightarrow{N'} & \cdot \\ {\scriptstyle name,\mathrm{case}}\downarrow & & \downarrow{\scriptstyle lcc,\mathrm{ncase}} \\ \cdot & \xrightarrow{N'} & \cdot \end{array}$$

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

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

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

– (lcase) The case is analogous to that of (lapp), with the diagram:

$$\begin{array}{ccc} \cdot & \xrightarrow{N'} & \cdot \\ {\scriptstyle name,\mathrm{lcase}}\downarrow & \nearrow_{N'} & \\ \cdot & & \end{array}$$

– (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:

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

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:

$$\begin{array}{ccc} \cdot & \xrightarrow{N'} & \cdot \\ {\scriptstyle name,\text{seq}}\downarrow & & \downarrow{\scriptstyle lcc,\text{nseq}} \\ \cdot & \xrightarrow{N'} & \cdot \end{array}$$

– (lseq) The case is analogous to (lapp) and (lcase), with the diagram:

$$\begin{array}{ccc} \cdot & \xrightarrow{N'} & \cdot \\ {\scriptstyle name,\text{lseq}}\downarrow & \nearrow_{N'} & \\ \cdot & & \end{array}$$

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 $R[\texttt{case}_T\ (c\ s_1\ \ldots\ s_{\text{ar}(c)})\ \texttt{of}\ alts]$, but c does not belong to type T. Let $N'(R) = (R', \sigma)$. Then $N'(s) = R'[\texttt{case}_T\ (c\ \sigma(N'(s_1))\ \ldots\ \sigma(N'(s_{\text{ar}(c)}))\ \texttt{of}\ alts']$ which shows that $N'(s)$ is not an L_{lcc}-WHNF and irreducible in L_{lcc}.
3. Expression s is of the form $R[((c\ s_1\ \ldots\ s_{\text{ar}(c)})\ r)]$. 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 $s = L[(c\ s_1 \ldots s_{\text{ar}(c)})]$ then $N'(s) = \lambda x.\sigma(N'(s'))$ or $N'(s) = (c\ \sigma(N'(s_1)) \ldots \sigma(N'(s_{\text{ar}(c)})))$ 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.

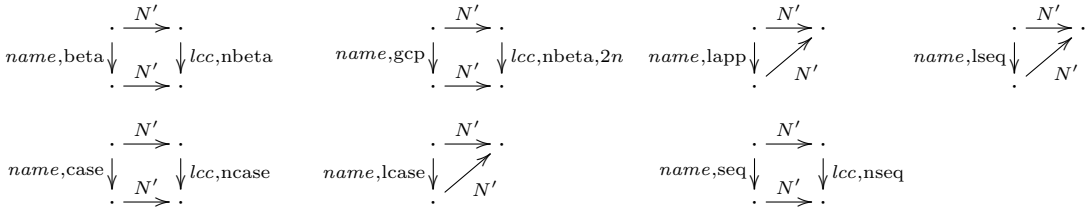
Transferring L_{lcc}-reductions into L_{name}-reductions
We will now analyze how normal order reductions for $N'(s)$ can be transferred into normal order reductions for s in L_{name}.

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 following diagrams:*



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 is necessary to observer that $\mathbb{E}_{\lambda} \subseteq \mathbb{E}_{\mathcal{L}}$ and 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: $N(s_1) \sim_{name} s_1 \leq_{name} s_2 \sim_{name} N(s_2)$ and 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 2.5.*

7 On Similarity in L_{LR}

In this section we will define co-inductive bisimilarity for L_{LR} and by showing that L_{LR} is convergence admissible we also derive an equivalent inductive characterization. Our results of the previous sections then enable us to show that bisimilarity coincides with contextual equivalence 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}:

Proposition 7.1. *Given any two closed $\mathbb{E}_{\mathcal{L}}$-expressions s_1, s_2. Then $s_1 \leq_{LR} s_2$ iff the following conditions hold:*

- *If $s_1{\downarrow_{LR}}\ t_1$ where t_1 is an AWHNF, then $s_2{\downarrow_{LR}}\ t_2$, where t_2 is also an AWHNF, and for all $r \in CE_{lcc}$ (see section 4.2.2): $t_1\ r \leq_{LR} t_2\ r$*
- *if $s_1{\downarrow_{LR}}\ t_1$, where t_1 is a CWHNF, then also $s_2{\downarrow_{LR}}\ t_2$, where t_2 is a CHWNF, and if $t_1 = R[(c\ t'_1 \ldots t'_n)]$ for an L_{LR}-CWHNF-context R, then $t_2 = R'[(c\ t''_1 \ldots t''_n)]$ for an L_{LR}-CWHNF-context R', and for all $i : R[t'_i] \leq_{LR} R'[t''_i]$.*

Proof. The $\Rightarrow$ direction is trivial. We show $\Leftarrow$, the nontrivial part: First assume that $s_1{\downarrow_{LR}}\ t_1$ and t_1 is an AWHNF. Then by assumption, $s_2{\downarrow}t_2$ where t_2 is an AWHNF and for all closed $r \in CE_{lcc}$ the inequation $t_1\ r \leq_{LR} t_2\ r$ holds. We transfer the problem to L_{lcc} as follows: $N(W(s_1))$ and $N(W(s_2))$ are closed $\mathbb{E}_{\lambda}$-expressions. Since the translation $N \circ W$ is surjective, every closed $\mathbb{E}_{\lambda}$-expression is in the image of $N \circ W$. Thus for every closed $\mathbb{E}_{\lambda}$-expression r' that is in CE_{lcc}, there is some $\mathbb{E}_{\mathcal{L}}$-expression r, such that $N(W(r)) = r'$. The expression r can be chosen letrec-free, since every closed $\mathbb{E}_{\lambda}$-expression is the image of a letrec-free $\mathbb{E}_{\mathcal{L}}$-expression. We have $N(W(s_1))\ r'{\downarrow_{lcc}} \implies N(W(s_2))\ r'{\downarrow_{lcc}}$, since $N(W(s_1\ r)) = (N(W(s_1))\ N(W(r)))$, and since $N \circ W$ is fully abstract. In the other case that $s_1{\downarrow_{LR}}\ R[(c\ s'_1 \ldots s'_n)$ where R is an L_{LR}-WHNF context, the assumption implies that $s_2{\downarrow_{LR}}\ R'[(c\ s''_1 \ldots s''_n)]$, where R' is an L_{LR}-WHNF context. Using the translation $N \circ W$, we see that $N(W(R[(c\ s'_1 \ldots s'_n)])) = (c\ N(W(R[s'_1])) \ldots N(W(R[s'_n])))$, and $N(W(R'[(c\ s''_1 \ldots e''_n)])) = (c\ N(W(R[s''_1])) \ldots N(W(R[s''_n])))$, and also $N(W(R[s'_i])) \leq_{lcc} N(W(R'[s''_i]))$. We apply Proposition 4.39 and obtain $N(W(s_1)) \leq_{lcc} N(W(s_2))$. Now Theorem 6.16 shows $s_1 \leq_{LR} s_2$. □

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

Lemma 7.2. *The closed part of the calculus L_{LR} is convergence-admissible: For all contexts $Q \in \mathcal{Q}_{CE}$, and closed L_{LR}-WHNFs w: $Q(s)\downarrow w$ iff $\exists v : s\downarrow v$ and $Q(v)\downarrow w$.*

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

Definition 7.3. *We define similarity $\leq_{b,LR}$ in L_{LR} as follows:*
Let s, t be closed $\mathbb{E}_{\mathcal{L}}$-expressions and η be a binary relation on closed $\mathbb{E}_{\mathcal{L}}$-expressions. We define an operator F_{LR} on binary relations on closed $\mathbb{E}_{\mathcal{L}}$-expressions:
$s\ F_{LR}(\eta)\ t$ iff the following holds:
$s\downarrow_{LR} v_1$ implies that $t\downarrow_{LR} v_2$, and the following

1. *If v_1 is an AWHNF, then v_2 is an AWHNF and for all $r \in CE_{lcc}$: $(v_1\ r)\ \eta\ (v_2\ r)$;*
2. *If v_1 is a CWHNF and if $v_1 = R[c\ s_1 \ldots s_n]$ where R is an L_{LR}-CWHNF-context, then $v_2 = R'[c\ t_1 \ldots t_n]$ where R' is an L_{LR}-CWHNF-context, and $R[s_i]\ \eta\ R'[t_i]$ for all i.*

The relation $\leq_{b,LR}$ is defined to be the greatest fixpoint of F_{LR} within binary relations on closed $\mathbb{E}_{\mathcal{L}}$-expressions. Its open extension is denoted with $\leq^o_{b,LR}$.

Lemma 7.4. *If in the definition above the restricted open extension using only closed expressions from CE_{lcc} is applied, then the relation $\leq^o_{b,LR}$ does not change.*

Proposition 7.5. *In L_{LR}, for closed $\mathbb{E}_{\mathcal{L}}$-expressions s, t the statement $s \leq_{b,LR} t$ is equivalent to the conjunction of the following conditions for s, t: For all $n \geq 0$, and for all contexts Q_i from $\mathcal{Q}_{CE}$ (see paragraph 4.2.2), $Q_1[\ldots Q_n[s]\ldots]\downarrow_{LR} \implies Q_1[\ldots Q_n[t]\ldots]\downarrow_{LR}$.*

Proof. Lemma 7.2 shows that Theorem 4.9 is applicable for the testing contexts from $\mathcal{Q}_{CE}$.

Now we can prove that the similarity $\leq_{b,LR}$ is equivalent to the contextual preorder on closed $\mathbb{E}_{\mathcal{L}}$-expressions:

Theorem 7.6. *For closed $\mathbb{E}_{\mathcal{L}}$-expressions s, t: $s \leq_{b,LR} t \iff s \leq_{LR} t$.*

Proof. Let $\leq^c_{LR}$ be the restriction of $\leq_{LR}$ to closed $\mathbb{E}_{\mathcal{L}}$-expressions. It is easy to verify that $\leq^c_{LR} \subseteq F_{LR}(\leq^c_{LR})$ and thus for closed $\mathbb{E}_{\mathcal{L}}$-expressions s,t: $s \leq_{LR} t \implies s \leq_{b,LR} t$. For the other direction let $s \leq_{b,LR} t$. The criterion in Proposition 7.5 then implies that for all $n \geq 0$ and contexts $Q_i \in \mathcal{Q}_{CE}$: $Q_1[\ldots Q_n[s]\ldots]{\downarrow_{LR}} \implies Q_1[\ldots Q_n[t]\ldots]{\downarrow_{LR}}$. Full-abstraction of $N \circ W$ (see Theorem 6.16) implies that $N(W(Q_1[\ldots Q_n[s]\ldots])){\downarrow_{lcc}} \implies N(W(Q_1[\ldots Q_n[t]\ldots]){\downarrow_{lcc}}$. Since N and W translate applications into applications, case-expressions into case-expressions and constructors to themselves, and since the translations are compositional, this also shows that $N(W(Q_1))[\ldots[N(W(Q_n))[N(W(s))]\ldots]{\downarrow_{lcc}} \implies N(W(Q_1))[\ldots[N(W(Q_n))[N(W(t))]\ldots]{\downarrow_{lcc}}$. Moreover, since every L_{lcc}-context from $\mathcal{Q}_{CE}$ is an $N \circ W$-image of the same context seen as an L_{LR}-context, we also conclude that $N(W(s)) \leq_{b,lcc,3} N(W(t))$. Now Theorem 4.37 and full abstraction of $N \circ W$ finally show $s \leq_{LR} t$.

Using the characterization in Proposition 7.5, it is possible to prove non-trivial equations, as shown in the example below.

Example 7.7. We consider two fixpoint combinators Y_1 and Y_2, where Y_1 is defined non-recursively, while Y_2 uses recursion. The definitions are: $Y_1 := \lambda f.((\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x)))$, $\quad Y_2 := \texttt{letrec}\ \mathit{fix} = \lambda f.f\ (\mathit{fix}\ f)\ \texttt{in}\ \mathit{fix}$.

Using Proposition 7.5 we can easily derive that $Y_1\ K \sim_{LR} Y_2\ K$ where $K := \lambda a.(\lambda b.a)$: For any $m \geq 0$ the expressions $Q_1[Q_2[\ldots[Q_m[(Y_i\ K)]]]]$ for $i = 1, 2$ converge if none of the contexts Q_i is a case-expression, and otherwise both expressions diverge.

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 4.38.

Lemma 7.8. *Let s,t be any $\mathbb{E}_{\mathcal{L}}$-expressions, and let the free variables of s,t be in $\{x_1,\ldots,x_n\}$. Then $s \leq_{LR} t \iff \lambda x_1,\ldots,x_n.s \leq_{LR} \lambda x_1,\ldots,x_n.t$*

The results above imply the following theorem:

Main Theorem 7.9 $\leq_{LR} = \leq^o_{b,LR}$.

Remark 7.10. 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 inequation $s \leq_{LR,\text{poly},\tau} t$ 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 inequation $s \leq_{LR} t$ 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 $\leq^o_{b,LR}$ 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.

The Main Theorem 7.9 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 7.11. *The identical embedding $\iota : \mathbb{E}_\lambda \to \mathbb{E}_\mathcal{L}$ is an isomorphism according to Definition 2.5*

8 The Operator seq Makes a Difference

Let $L_{lcc,\neg seq}$ be the calculus L_{lcc} without `seq`-expressions (and without (seq)-reduction), *i.e.* $L_{lcc,\neg seq}$ is Abramsky's lazy lambda calculus extended with `case`-expressions and data constructors. Hence L_{lcc} extends $L_{lcc,\neg seq}$ by the `seq`-operator. We show that the extension L_{lcc} of $L_{lcc,\neg seq}$ is not conservative, *i.e.*, the contextual equality is not preserved, when `seq` is added to $L_{lcc,\neg seq}$, since there are more contexts (note that this is the same example as in [SSSS08]). The assumption is that there is at least the Boolean type with the constructors `True` and `False`. We also show that there is no compositional and convergence equivalent translation that makes the calculi L_{lcc} and $L_{lcc,\neg seq}$ isomorphic.

Definition 8.1. *We define the following expressions in L_{lcc}:*

$Y := \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$ $\quad$ `Bot` $:= Y\ I$ $\quad$ `Top` $:= Y\ K$
$s := \lambda f.(\texttt{if}\ f\ (\lambda x.\texttt{Bot})\ \texttt{then True else Bot})$ $\quad$ $t := \lambda f.\texttt{if}\ (f\ \texttt{Bot})\ \texttt{then True else Bot}$

Lemma 8.2. *The equivalence $s\ \sim_{lcc,\neg seq}\ t$ holds.*

Proof. A context lemma holds in the lazy lambda calculus with case and constructors [SSS10]. Applicative bisimulation can be used in the lazy lambda calculus with case and constructors, since it is a call-by-name and deterministic calculus with the usual reduction rules. Hence it is sufficient to test for s, t whether contexts of the form $([\cdot]\ f)$ result in the same answer or both do not converge.
We compare the evaluation results of $(s\ f)$ and $(t\ f)$. If f is a constant function, then either $s\ f$ and $t\ f$ are not convergent, or f is a constant function with result `True`, and $s\ f$ and $t\ f$ are both convergent. If f analyses the argument using a case-expression or applying the argument to another expression, then $s\ f$ will result in `Bot`. Also $t\ f$ will result in non-convergence in this case. If f simply returns its argument, then also both applications $s\ f$ and $(t\ f)$ result in non-convergence.

Lemma 8.3. *We have $s \not\sim_{lcc} t$.*

Proof. We apply s, t to the function $\lambda x.\texttt{seq}\ x\ \texttt{True}$: Then $(s\ (\lambda x.\texttt{seq}\ x\ \texttt{True}))\downarrow$, but $(t\ (\lambda x.\texttt{seq}\ x\ \texttt{True}))\Uparrow$.

Proposition 8.4. *The natural embedding of $L_{lcc,\neg seq}$ into L_{lcc} is not conservative.*

Proof. The lemmas above show that the embedding is not conservative, since s, t are contextually equal in $L_{lcc,\neg seq}$, but different in L_{lcc}.

Proposition 8.5. *The calculi $L_{lcc,\neg seq}$ and L_{lcc} are not isomorphic* w.r.t. *compositional translations that keep types and constructors.*

Proof. We show that in $L_{lcc,\neg seq}$ there is no context C with $a{\downarrow} \iff C[a] \sim \texttt{True}$: Suppose there is such a C. Then $C[x]$ cannot reduce to `True` for all x, hence the reduction of $C[x]$ will use the argument x. The first use in a normal order reduction of $C[x]$ may be a case analysis of x: in this case $C[x]$ does not converge for abstractions. If the first use is an application of x to some expression, then $C[x]$ does not converge for $x = \texttt{True}$. Since the translation τ is assumed to be compositional and convergence equivalent, the translation keeps the formula $a{\downarrow} \iff C[a] \sim \texttt{True}$.

However, in L_{lcc} the context $C[\cdot] = \texttt{seq}\ [\cdot]\ \texttt{True}$ is the requested one, which cannot be retranslated into $L_{lcc,\neg seq}$. Hence there is no compositional and convergence equivalent translation that is an isomorphism.

The extension by case and constructors of the lazy lambda calculus is also nontrivial:

Proposition 8.6. *The calculus $L_{lcc,\neg seq}$ is not isomorphic to Abramsky's lazy lambda calculus, if there is a nontrivial type T in $L_{lcc,\neg seq}$.*

Proof. The closed term $\texttt{Top} := Y\ K$ is a greatest element in the lazy lambda calculus of Abramsky, and unique up to $\sim$. However, the calculus $L_{lcc,\neg seq}$ does not have a greatest element: Suppose a is a greatest element. Then the context $\texttt{case}_T\ [\cdot]\ \ldots$ shows that a must be of the form $c\ a_1 \ldots a_n$, where c is a constructor of type T. Since also $a \geq Y\ K$, we would derive $Y\ K \sim Y\ K\ \bot \leq a\ \bot \sim \bot$ which is a contradiction, since $Y\ K \not\sim \bot$.

Remark 8.7. It is an open issue whether the embedding of Abramsky's lazy lambda calculus into $L_{lcc,\neg seq}$ is conservative or not. It is also open whether the embedding of Abramsky's lazy lambda calculus into an extension with `seq` is conservative or not. These issues are related to the open question to determine a (unique) form of infinite tree representation like the Böhm-trees for the expressions of Abramsky's lazy lambda calculus *w.r.t.* contextual equivalence (see also Problem # 18 in the TLCA list of open problems [TLC10].)

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

In this section we introduce a variant of the calculus L_{LR}. The call-by-need calculus L_{need} has the same syntax as the calculus L_{LR} (the expression $\mathbb{E}_{\mathcal{L}}$), but uses a simplified form of reduction rules. The goal of this section is to show that L_{need} and L_{LR} are equivalent, *i.e.* all of our results also apply to the calculus L_{need}.

$(\texttt{letrec}\ Env\ \texttt{in}\ s)^T$	$\rightsquigarrow (\texttt{letrec}\ Env\ \texttt{in}\ s^S)^V$
$(s\ t)^{S\vee T}$	$\rightsquigarrow (s^S\ t)^V$
$(\texttt{seq}\ s\ t)^{S\vee T}$	$\rightsquigarrow (\texttt{seq}\ s^S\ t)^V$
$(\texttt{case}_T\ s\ \texttt{of}\ \text{alts})^{\mathrm{S\vee T}}$	$\rightsquigarrow (\texttt{case}_T\ s^S\ \texttt{of}\ \text{alts})^{\mathrm{V}}$
$(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C[x^S])$	$\rightsquigarrow (\texttt{letrec}\ x = s^S, Env\ \texttt{in}\ C[x^V])$ if s was not labeled
$(\texttt{letrec}\ x = s, y = C[x^S], Env\ \texttt{in}\ t)$	$\rightsquigarrow (\texttt{letrec}\ x = s^S, y = C[x^V], Env\ \texttt{in}\ t)$ if s was not labeled and if $C[x] \neq x$

Fig. 7. The rules of the L_{need}-labeling algorithm

The labeling algorithm in Fig. 7 will detect the position to which a reduction rule will be applied according to normal order. It uses three labels: S, T, V, where T means reduction of the top term, S means reduction of a subterm, and V labels already visited subexpressions, and $S \vee T$ matches T as well as S. The algorithm does not look into S-labeled letrec-expressions. We also denote the fresh V only in the result of the unwind-steps, and do not indicate the already existing V-labels. There will be at most one S-label at positions during the execution of the labeling. For an expression s the labeling algorithm starts with s^T, where no subexpression in s is labeled. We assume that the expression that gets a new label was not labeled before. Then this algorithm terminates, *e.g.* for $(\texttt{letrec}\ x = x\ \texttt{in}\ x)^T$ it will stop with $(\texttt{letrec}\ x = x^S\ \texttt{in}\ x^V)^V$. For the normal-order reduction rules, we call an expression of the form $(c\ x_1 \ldots x_n)$ a cv-expression if all x_i are variables.

Definition 9.1 (Normal Order Reduction of L_{need}). *A normal order reduction for L_{need} is defined as the reduction at the position of the final label S, or one position higher up, or copying the term from the final position to the position before, as indicated in Fig. 8. A normal-order reduction step is denoted as $\xrightarrow{need}$.*

Note that normal order reduction is unique. Note also that L_{need} and L_{LR} have the reduction rules rules (lbeta), (case-c), (seq-c), (llet-in), (llet-e), (lapp), (lcase), and (lseq) in common, but in difference to L_{LR}, the calculus L_{need} copies variables and constructor applications (using the rule (cpcv-in) and (cpcv-e)). As a consequence in L_{need} it is not necessary to follow variable-to-variable chains in `letrec`-environments. For instance, for the expression $\texttt{letrec}\ x = \texttt{Cons True Nil}, y = x, z = y\ \texttt{in}\ \texttt{case}_{List}\ z\ \texttt{of}\ (\texttt{Cons}\ u\ v\ \rightarrow\ v)\ (\texttt{Nil} \rightarrow \texttt{Nil})$ the normal order reduction in L_{need} proceeds as follows:

(lbeta)	$C[((\lambda x.s)^S\ t)] \to C[\texttt{letrec}\ x = t\ \texttt{in}\ s]$
(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)	$\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s)^S\ \texttt{in}\ t \to \texttt{letrec}\ Env_1, Env_2, x = s\ \texttt{in}\ t$
(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)	$C[(\texttt{case}_T\ (c_i\ \overrightarrow{s})^S \texttt{of} \ldots ((c_i\ \overrightarrow{y}) \to t_i) \ldots)] \to C[(\texttt{letrec}\ \{y_i = s_i\}_{i=1}^{\mathrm{ar}(c_i)}\ \texttt{in}\ t_i)]$ 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 $\mathrm{ar}(c_i) = 0$
(cpcv-in)	$(\texttt{letrec}\ x = s^S, Env\ \ \texttt{in}\ C[x^V]) \to (\texttt{letrec}\ x = s, Env\ \ \texttt{in}\ C[s])$ where s is an abstraction or a variable or a cv-expression
(cpcv-e)	$(\texttt{letrec}\ x = s^S, Env, y = C[x^V]\ \texttt{in}\ t) \to (\texttt{letrec}\ x = s, Env, y = C[s]\ \texttt{in}\ t)$ where s is an abstraction or a variable or a cv-expression
(abs)	$(\texttt{letrec}\ x = (c\ s_1 \ldots s_n)^S, Env\ \texttt{in}\ t) \to (\texttt{letrec}\ x = (c\ x_1 \ldots x_n), x_1 = s_1, \ldots, x_n = s_n, Env\ \texttt{in}\ t)$

Fig. 8. Normal-Order Reduction Rules of L_{need}

$$
\begin{array}{ll}
 & \texttt{letrec}\ x = \texttt{Cons True Nil}, y = x, z = y\ \texttt{in} \\
 & \quad \texttt{case}_{List}\ z\ \texttt{of}\ (\texttt{Cons}\ u\ v\ \to\ v)\ (\texttt{Nil} \to \texttt{Nil}) \\
\xrightarrow{need,\mathrm{cpcv}} & \texttt{letrec}\ x = \texttt{Cons True Nil}, y = x, z = y\ \texttt{in} \\
 & \quad \texttt{case}_{List}\ y\ \texttt{of}\ (\texttt{Cons}\ u\ v\ \to\ v)\ (\texttt{Nil} \to \texttt{Nil}) \\
\xrightarrow{need,\mathrm{cpcv}} & \texttt{letrec}\ x = \texttt{Cons True Nil}, y = x, z = y\ \texttt{in} \\
 & \quad \texttt{case}_{List}\ x\ \texttt{of}\ (\texttt{Cons}\ u\ v\ \to\ v)\ (\texttt{Nil} \to \texttt{Nil}) \\
\xrightarrow{need,\mathrm{abs}} & \texttt{letrec}\ x = \texttt{Cons}\ w\ w', w = \texttt{True}, w' = \texttt{Nil}, y = x, z = y\ \texttt{in} \\
 & \quad \texttt{case}_{List}\ y\ \texttt{of}\ (\texttt{Cons}\ u\ v\ \to\ v)\ (\texttt{Nil} \to \texttt{Nil}) \\
\xrightarrow{need,\mathrm{cpcv}} & \texttt{letrec}\ x = \texttt{Cons}\ w\ w', w = \texttt{True}, w' = \texttt{Nil}, y = x, z = y\ \texttt{in} \\
 & \quad \texttt{case}_{List}\ (\texttt{Cons}\ w\ w')\ \texttt{of}\ (\texttt{Cons}\ u\ v\ \to\ v)\ (\texttt{Nil} \to \texttt{Nil}) \\
\xrightarrow{need,\mathrm{case\text{-}c}} & \texttt{letrec}\ x = \texttt{Cons}\ w\ w', w = \texttt{True}, w' = \texttt{Nil}, y = x, z = y\ \texttt{in} \\
 & \quad (\texttt{letrec}\ u = w, v = w'\ \texttt{in}\ v) \\
\xrightarrow{need,\mathrm{llet\text{-}in}} & \texttt{letrec}\ x = \texttt{Cons}\ w\ w', w = \texttt{True}, w' = \texttt{Nil}, y = x, z = y, u = w, v = w'\ \texttt{in}\ v \\
\xrightarrow{need,\mathrm{cpcv}} & \texttt{letrec}\ x = \texttt{Cons}\ w\ w', w = \texttt{True}, w' = \texttt{Nil}, y = x, z = y, u = w, v = w'\ \texttt{in}\ w' \\
\xrightarrow{need,\mathrm{cpcv}} & \texttt{letrec}\ x = \texttt{Cons}\ w\ w', w = \texttt{True}, w' = \texttt{Nil}, y = x, z = y, u = w, v = w'\ \texttt{in True}
\end{array}
$$

An L_{need}-WHNF is defined as an expression of the form (`letrec` Env `in` v) where v is an abstraction or a constructor application. Convergence of an expression s is defined as $s{\downarrow_{need}}$ iff there is an L_{need}-normal order reduction sequence $s \xrightarrow{need} t$, where t is an L_{need}-WHNF.

Concluding, the calculus L_{need} is defined by the tuple $(\mathbb{E}_{\mathcal{L}}, \mathbb{C}_{\mathcal{L}}, \xrightarrow{need}, \mathbb{A}_{need})$ where $\mathbb{A}_{need}$ is the set of L_{need}-WHNFs.

9.1 Equivalence of Tree-Convergence and L_{need}-Convergence

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

Lemma 9.2. *Let s, t be finite expressions and $s \to t$ by a rule (abs), (cpcv-in), (cpcv-e), or (lll). Then $IT(s) = IT(t)$.*

Lemma 9.3. *Let s be a finite expression. If s is an L_{need}-WHNF then $IT(s)$ is an L_{tree}-answer. If $IT(s)$ is an L_{tree}-answer, then $s{\downarrow_{need}}$.*

Proof. If s is an L_{need}-WHNF, then obviously, $IT(s)$ is an L_{tree}-answer. The other direction follows analogous to the proof of Lemma 5.17, where the only difference is the case for $s = \texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{n}, Env\ \texttt{in}\ x_n$, where v is an abstraction or a constructor application: Then n $(need, \text{cpcv})$-reductions are required to obtain a WHNF, and in case of a constructor application an additional $(need, \text{abs})$-reduction must be performed first (if the arity of the corresponding constructor is greater than 0). □

Lemma 9.4. *Let $s \in \mathbb{E}_{\mathcal{L}}$ such that $s \xrightarrow{need,a} t$. If the reduction a is (*cpcv-in*), (*cpcv-e*), (*abs*), or (*lll*) then $IT(s) = IT(t)$. If the reduction a is (*lbeta*), (*case-c*), (*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.*

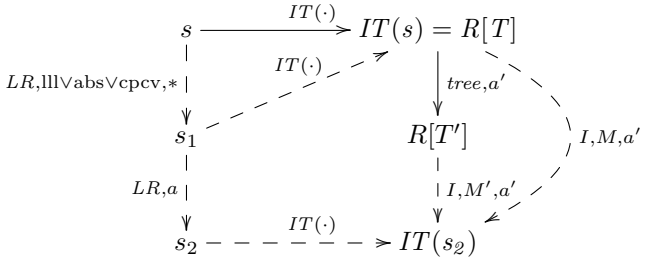
Proof. The first part is obvious. The second part can be proved by labeling every redex in $IT(s)$ that is derived from the redex $s \xrightarrow{need} t$ by $IT(.)$. This results in the set M for $IT(s)$. There will be at least one position in M that is a normal order redex of $IT(s)$.

Proposition 9.5. *Let s be a finite expression such that $s{\downarrow_{need}}$. Then $IT(s){\downarrow_{tree}}$.*

Proof. We assume that $s \xrightarrow{need,*} t$ where t is an L_{need}-WHNF. Using Lemma 9.4, we see that there is a finite sequence of reductions $IT(s) \xrightarrow{I,*} IT(t)$. Lemma 9.3 shows that $IT(t)$ is an L_{tree}-WHNF. Now Proposition 5.15 shows that $IT(s){\downarrow_{tree}}$.

We will now show that s converges whenever $IT(s)$ converges

Lemma 9.6. *Let* $IT(s) = R[T] \xrightarrow{tree,a'} R[T']$ *for some reduction context* R. *Then* $s \xrightarrow{need,\mathrm{lll}\vee\mathrm{abs}\vee\mathrm{cpcv},*} s_1 \xrightarrow{need,a} s_2$ *with* $R[T'] \xrightarrow{I,M} IT(s_2)$ *where* $(a', a) \in \{(\mathrm{betaTr}, \mathrm{lbeta}), (\mathrm{caseTr}, \mathrm{case}), (\mathrm{seqTr}, \mathrm{seq})\}$.



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)-, (abs)-, and (cpcv)-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)-reductions moves the environment Env to the top of the expression, where perhaps it is joined with a top-level environment of C. Let $s \xrightarrow{\mathrm{lll},*} s'$. Lemma 9.2 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 $s \xrightarrow{\mathrm{lll},*} s_1$ 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 9.2). 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 $\texttt{letrec}\ x_1 = v, x_2 = x_1, \ldots, x_m = x_{m-1}, \ldots R'[x_m] \ldots$, where v is a constructor application or an abstraction. For the latter case a sequence of $(need, \mathrm{abs} \vee \mathrm{cpcv}, *)$-reductions is necessary before the corresponding reduction rule can be applied. Again Lemma 9.2 assures that the infinite tree remains unchanged. After applying the corresponding reduction rule, *i.e.* $s_1 \xrightarrow{need,a} s_2$, the normal order reduction may have changed infinitely many positions of $IT(s_2)$, while $R[T] \xrightarrow{tree,a'} R[T']$ does not change all these position, but nevertheless Lemma 9.4 shows that there is a reduction $R[T] \xrightarrow{I,M,a'} IT(s_2)$, and Lemma 5.11 shows that also $R[T'] \xrightarrow{I,M',a'} IT(s_2)$ for some M'.

Proposition 9.7. *Let* s *be an expression such that* $IT(s)\!\downarrow_{tree}$. *Then* $s\!\downarrow_{need}$.

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 9.3. In the general case, let $T \xrightarrow{tree,a'} T'$ be a *tree*-reduction. Lemma 9.6 shows that there are expressions s', s'' with $s \xrightarrow{need,\mathrm{lll}\vee\mathrm{abs}\vee\mathrm{cpcv},*} s' \xrightarrow{need,a} s''$, 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 9.5 and 9.7 imply the theorem

Theorem 9.8. *Let s be an $\mathbb{E}_{\mathcal{L}}$-expression. Then $s{\downarrow_{need}}$ if and only if $IT(s){\downarrow_{tree}}$.*

9.2 Similarity in L_{need}

Since L_{need} and L_{LR} use the same language a direct consequence of Theorem 9.8 and Proposition 5.19 is that the contextual equivalences of both calculi coincide. Main Theorem 7.9 implies that both relations are equivalent to similarity.

Theorem 9.9. $\leq_{LR} = \leq_{need} = \leq^o_{b,LR}$

Also the inductive variant of behavioral preorder can be used for L_{need}:

Proposition 9.10. *In L_{need}, for closed $\mathbb{E}_{\mathcal{L}}$-expressions s, t the statement $s \leq_{need} t$ is equivalent to the conjunction of the following conditions for s, t: For all $n \geq 0$, and for all contexts Q_i from $\mathcal{Q}_{CE}$ (see Section 4.2.2), $Q_1[\ldots Q_n[s]\ldots]{\downarrow_{need}} \implies Q_1[\ldots Q_n[t]\ldots]{\downarrow_{need}}$.*

Proof. This follows from Proposition 7.5, and Theorems 9.8 and 9.9.

10 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

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*, pp. 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*, pp. 233–246, 1995. ACM.

AK94. Z. M. Ariola and J. W. Klop. Cyclic Lambda Graph Rewriting. In *LICS 1994*, pp. 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, 1984.

Fel91. M. Felleisen. On the expressive power of programming languages. *Sci. Comput. Programming*, 17(1–3):35–75, 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*, pp. 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*, pp. 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*, pp. 141–152, 2006. ACM.

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

Mil80. R. Milner. *A Calculus of Communicating Systems*, *LNCS* 92. 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*, pp. 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. http://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*, *Cambridge Tracts Theoret. Comput. Sci.* 52, chapter 5, pp. 197–232. Cambridge University Press, 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*, *LNCS* 4533, pp. 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*, LNCS 5117, pp. 321–335. Springer-Verlag, 2008.

SSNSS08. M. Schmidt-Schauß, J. Niehren, J. Schwinghammer, and D. Sabel. Adequacy of compositional translations for observational semantics. In *5th IFIP TCS 2008*, *IFIP* 273, pp. 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/.

SSS10. M. Schmidt-Schauß and D. Sabel. On generic context lemmas for higher-order calculi with sharing. *Theoret. Comput. Sci.*, 411(11-13):1521 – 1541, 2010.

SSS11a. D. Sabel and M. Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In *PPDP 2011*, pp. 101–112, New York, NY, USA, July 2011. ACM.

SSS11b. D. Sabel and M. Schmidt-Schauß. On conservativity of Concurrent Haskell. Frank report 47, Institut f. Informatik, Goethe-University, Frankfurt, 2011. http://www.ki.informatik.uni-frankfurt.de/papers/frank/.

SSSM10. M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Simulation in the call-by-need lambda-calculus with letrec. In *RTA 2010*, *LIPIcs* 6, pp. 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.

TLC10. TLCA list of open problems, http://tlca.di.unito.it/opltlca/, 2010.

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