International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 295-310
http://rewriting.loria.fr/rta/

SIMULATION IN THE CALL-BY-NEED LAMBDA-CALCULUS WITH LETREC

MANFRED SCHMIDT-SCHAUSS [1] AND DAVID SABEL [1] AND ELENA MACHKASOVA [2]

[1] Dept. Informatik und Mathematik, Inst. Informatik, Goethe-University, PoBox 11 19 32, D-60054 Frankfurt, Germany
E-mail address, M. Schmidt-Schauß: schauss@ki.informatik.uni-frankfurt.de
E-mail address, D. Sabel: sabel@ki.informatik.uni-frankfurt.de

[2] Division of Science and Mathematics, University of Minnesota, Morris, MN 56267-2134, U.S.A
E-mail address, E. Machkasova: elenam@morris.umn.edu

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

1. Introduction

Non-strict programming languages such as the core-language of Haskell can be modeled using call-by-need lambda calculi. Contextual semantics, based on an operational semantics, describes behavior of expressions in all possible contexts and can model the semantics of different variants of these calculi. Applicative bisimulation is a restricted form of contextual equivalence: if two closed expressions behave the same on all arguments, then they are bisimilar. It allows more convenient proofs of e.g. correctness of program transformations. Abramsky & Ong showed that applicative bisimulation is the same as contextual equivalence in a specific simple lazy lambda calculus [Abr90, Abr93], and Howe [How89, How96] proved that in classes of calculi applicative bisimulation is the same as contextual equivalence. This leads to the expectation that some form of applicative bisimulation may be used for calculi with Haskell's cyclic let(rec). Howe's method is applicable to calculi with non-recursive let even in the presence of non-determinism [Man10]. However, in the case of (cyclic) letrec

1998 ACM Subject Classification: F.4.2, F.3.2, F.3.3, F.4.1.
Key words and phrases: semantics, contextual equivalence, bisimulation, lambda calculus, call-by-need, letrec.

© M. Schmidt-Schauß, D. Sabel, and E. Machkasova
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.295

and non-determinism the method fails, as a recent counterexample shows [SS09a]. This raises a question: which call-by-need calculi with letrec permit applicative bisimilarity as a tool for proving contextual equality.

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

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

Related work on calculi with recursive bindings includes the following foundational papers. An early paper that proposes cyclic let-bindings (as graphs) is [Ari94], where reduction and confluence properties are discussed. [Ari95, Ari97, Mar98] 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 [Mar98] that call-by-name and call-by-need evaluation induce the same observational equivalences. Call-by-need lambda calculi with a recursive let that closely correspond to our calculus L_{need} are also presented in [Ari95, Ari97, Ari02]. In [Ari02] it is shown that there exist infinite normal forms and that the calculus satisfies a form of confluence. In this paper we show that the letrec calculus of [Ari97] is equivalent to L_{need} w.r.t. convergence and contextual equivalence (see Theorem 7.1) and that bisimulation for the letrec calculus of [Ari97] is equivalent to contextual equivalence. This supports our experience and view that contextual equivalence is a more central notion than a specific standard reduction.

Outline: In Sect. 3 we introduce the two letrec-calculi and recall results for Abramsky's lazy lambda calculus. In Sect. 4 and 5 the translations W and N are introduced and the full-abstraction results are obtained. In Sect. 6 we show that bisimulation and contextual equivalence are the same in the call-by-need calculus with letrec. In Sect. 7 we show that our result is transferable to the letrec-calculus of [Ari97]. Finally, we conclude in Sect. 8.

2. Common Notions and Notations for Calculi

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

Definition 2.1. An untyped deterministic *calculus* $\mathcal{D}$ is a four-tuple $(\mathcal{E}, \mathcal{C}, \rightarrow, \mathcal{W})$, where $\mathcal{E}$ are expressions, $\mathcal{C} : \mathcal{E} \rightarrow \mathcal{E}$ is a set of functions (which usually represents contexts), $\rightarrow$

is a small-step reduction relation (usually the normal-order reduction), which is a partial function on expressions, and $\mathcal{W} \subset \mathcal{E}$ is a set of *values* of the calculus.

For $C \in \mathcal{C}$ and an expression s, the functional application is denoted as $C[s]$. For contexts, this is the replacement of the hole of C by s. We also assume that the identity function Id is contained in $\mathcal{C}$ with $Id[s] = s$ for all expressions s.
The *transitive closure* of $\rightarrow$ is denoted as $\xrightarrow{+}$ and the *transitive and reflexive closure* of $\rightarrow$ is denoted as $\xrightarrow{*}$. Given an expression t, a sequence $t \rightarrow t_1 \rightarrow \ldots \rightarrow t_n$ is called a *reduction sequence*; it is called an *evaluation* if t_n is a value, i.e. $t_n \in W$. Then we say s *converges* and denote this as $s{\downarrow}t_n$ or as $s{\downarrow}$ if t_n is not important. If there is no t_n s.t. $s{\downarrow}t_n$ then s *diverges*, denoted as $s{\Uparrow}$. When dealing with multiple calculi, we often use the calculus name to mark its expressions and relations, e.g. $\xrightarrow{\mathcal{D}}$ denotes a reduction relation in $\mathcal{D}$.

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

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

$$\begin{array}{lll} s \leq_D t & \text{iff} & \forall C \in \mathcal{C} : \ C[s]{\downarrow}_D \Rightarrow C[t]{\downarrow}_D \\ s \sim_D t & \text{iff} & s \leq_D t \wedge t \leq_D s \end{array}$$

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

We are interested in translations between calculi that are faithful w.r.t. the corresponding contextual preorders. Recall that we developed such translations between calculi with contextual equivalences in [SS08b, SS09b]: A translation $\tau : (\mathcal{E}_1, \mathcal{C}_1, \rightarrow_1, \mathcal{W}_1) \rightarrow (\mathcal{E}_2, \mathcal{C}_2, \rightarrow_2, \mathcal{W}_2)$ is a mapping $\tau_E : \mathcal{E}_1 \rightarrow \mathcal{E}_2$ and a mapping $\tau_C : \mathcal{C}_1 \rightarrow \mathcal{C}_2$ such that $\tau_C(Id_1) = Id_2$. The following notions are defined:

- τ is *compositional* iff $\tau(C[e]) = \tau(C)[\tau(e)]$ for all C, e.
- τ is *convergence equivalent* iff $e{\downarrow}_1 \iff \tau(e){\downarrow}_2$ for all e.
- τ is *adequate* iff for all $e, e' \in \mathcal{E}_1$: $\tau(e) \sim_2 \tau(e') \implies e \sim_1 e'$.
- τ is *fully abstract* iff for all $e, e' \in \mathcal{E}_1$: $e \sim_1 e' \iff \tau(e) \sim_2 \tau(e')$.

From [SS08b, SS09b] it is known that a compositional and convergence equivalent translation is adequate.

3. Three Calculi

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

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

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

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

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

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

Figure 1: Reduction rules of L_{need}

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 $(\texttt{letrec}\ x_1 = s_1, \dots, x_{n-1} = s_{n-1}\ \texttt{in}\ s_n)$ are all expressions s_i with $1 \le i \le n$. Free and bound variables in expressions and α-renamings are defined as usual. The set of free variables in t is denoted as $FV(t)$. We use the distinct variable convention, i.e., all bound variables in expressions are assumed to be distinct, and free variables are distinct from bound variables. The reduction rules are assumed to implicitly α-rename bound variables in the result if necessary.

A *context* C is an expression from L_{need} extended by a symbol $[\cdot]$, the *hole*, such that $[\cdot]$ occurs exactly once (as subexpression) in C. Given a term t and a context C, we write $C[t]$ for the L_{need}-expression constructed from C by plugging t into the hole, i.e, by replacing $[\cdot]$ in C by t, where this replacement is meant syntactically, i.e., a variable capture is permitted.

Definition 3.1. The *reduction rules* for the calculus and language L_{need} are defined in Fig. 1, where the labels S, V are used for the exact definition of the normal-order reduction below. Several reduction rules are denoted by their name prefix, e.g. the union of (llet-in) and (llet-e) is called (llet). The union of (llet) and (lapp) is called (lll).

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

Example 3.2. For the expression $\texttt{letrec}\ x = x\ \texttt{in}\ x$ the labeling does not fail:

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

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

Figure 2: Labeling algorithm for L_{need}

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

Definition 3.3 (Normal Order Reduction of L_{need})**.** Let t be an expression. Then a single normal order reduction step $\xrightarrow{need}$ is defined as follows: first the labeling algorithm is applied to t. If the labeling algorithm terminates successfully, then one of the rules in Figure 1 is applied, if possible, where the labels S, V must match the labels in the expression t (again t may have more labels). The *normal order redex* is defined as the left-hand side of the applied reduction rule. The notation for a normal-order reduction that applies the rule a is $\xrightarrow{need,a}$, e.g. $\xrightarrow{need,lapp}$ applies the rule (*lapp*).

Definition 3.4. A *reduction context* R_{need} is any context, such that its hole is labeled with S or T by the labeling algorithm.

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

Example 3.5. We consider the expression t_1 := `letrec` $x = (y\ \lambda u.u), y = \lambda z.z$ `in` x. The labeling algorithm applied to t_1 yields (`letrec` $x = (y^V\ \lambda u.u)^V, y = (\lambda z.z)^S$ `in` $x^V)^V$. The only reduction rule that matches this labeling is the reduction rule (cp-e), i.e. $t_1 \xrightarrow{need}$ (`letrec` $x = ((\lambda z'.z')\ \lambda u.u), y = (\lambda z.z)$ `in` $x) = t_2$. The labeling of t_2 is (`letrec` $x = ((\lambda z'.z')^S\ \lambda u.u)^V, y = (\lambda z.z)$ `in` $x^V)^V$, which makes the reduction (lbeta) applicable, i.e. $t_2 \xrightarrow{need}$ (`letrec` $x =$ (`letrec` $z' = \lambda u.u$ `in` $z'), y = (\lambda z.z)$ `in` $x) = t_3$. The labeling of t_3 is (`letrec` $x =$ (`letrec` $z' = \lambda u.u$ `in` $z')^S, y = (\lambda z.z)$ `in` $x^V)^V$. Thus an (llet-e)-reduction is applicable to t_2, i.e. $t_3 \xrightarrow{L_{need}}$ (`letrec` $x = z', z' = \lambda u.u, y = (\lambda z.z)$ `in` $x) = t_4$. Application of the labeling algorithm to t_4 yields: (`letrec` $x = z'^S, z' = \lambda u.u, y = (\lambda z.z)$ `in` $x^V)^V$. Thus the normal order reduction is a (cp-in)-reduction, i.e. $t_4 \xrightarrow{L_{need}}$ (`letrec` $x = z', z' = \lambda u.u, y = (\lambda z.z)$ `in` $z') = t_5$ The labeling of t_5 is (`letrec` $x = z', z' = \lambda u.u^S, y = (\lambda z.z)$ `in` $z'^V)^V$. Again a (cp-e) reduction is applicable, i.e. $t_5 \to$ (`letrec` $x = z', z' = \lambda u.u, y = (\lambda z.z)$ `in` $\lambda u'.u') = t_6$ The labeling algorithm applied to t_6

yields $(\texttt{letrec}\ x = z', z' = \lambda u.u, y = (\lambda z.z)\ \texttt{in}\ \lambda u'.u'^S)^V$, but no reduction is applicable to t_6, since t_6 is a WHNF.

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

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

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

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

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

3.3. The Lazy Lambda Calculus

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

We repeat the definitions and the required properties of L_{lazy}, where proofs can be found in [How89, How96, Abr90, Abr93]. For basic definitions and confluence see e.g. [Bar84]. Since this calculus is well-studied and some properties are folklore, there are different and alternative proofs of the properties below. We require these properties in other sections and as properties of the target of translations, which allows us to lift the properties to the calculi L_{name} and L_{need}.

Definition 3.6 (Simulation in L_{lazy})**.** Let η be a binary relation on closed L_{lazy}-expressions. Then $s\ [\eta]_{lazy}\ t$ holds iff $s{\downarrow}\lambda x.s'$ implies $\big(\, t{\downarrow}\lambda x.t'$ and for all closed L_{lazy}-expressions r the relation $s'[r/x]\ \eta\ t'[r/x]$ holds$\big)$. The relation $\leq_{b,lazy}$ is defined as the greatest fixpoint of the operator $[\cdot]_{lazy}$.

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

There are several variants of behaviorally and contextually defined relations in L_{lazy}, that are all equivalent to contextual approximation.

Theorem 3.7. *In L_{lazy}, all the following relations are equivalent to contextual approximation $\leq_{lazy}$:*

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

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

Theorem 3.8. *Let s, t be L_{lazy}-expressions. If $s \xrightarrow{beta} t$, then $s \sim_{lazy} t$. For all L_{lazy}-expressions s, t: $\Omega \leq_{lazy} s$. If s, t are closed and $s\Uparrow$ and $t\Uparrow$, then $s \sim_{lazy} t$.*

Also the following can easily be derived from Theorem 3.7 and Theorem 3.8.

Proposition 3.9. *For open L_{lazy}-expressions s, t, where all free variables of s, t are in $\{x_1, \ldots, x_n\}$: $s \leq_{lazy} t \iff \lambda x_1, \ldots x_n.s \leq_{lazy} \lambda x_1, \ldots x_n.t$*

Proposition 3.10. *Given any two closed L_{lazy}-expressions s, t: for all closed L_{lazy}-abstractions r and also for $r = \Omega$ $s\ r \leq_{lazy} t\ r \iff s \leq_{lazy} t$.*

Proof. The if-direction follows from the congruence property. The only-if direction follows from Theorem 3.7. ∎

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

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

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

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

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

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

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

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

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

The final steps in the label computation are as follows:

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

For the general cases, we proceed as follows:

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

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

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

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

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

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

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

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

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

Now we show the other direction:

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

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

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

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

Proof. If $IT(s) \xrightarrow{tree} T$, then $IT(s) = \mathcal{R}[(\lambda x.t_1)\ t_2]$ where $\mathcal{R}$ is a reduction context and $T = \mathcal{R}[t_1[t_2/x]]$. Let p be the position of the hole of $\mathcal{R}$ in $IT(s)$. We first apply Lemma 4.6 to s and the tree context $\mathcal{R}[([\cdot]\ t_2)]$ and thus obtain a reduction $s \xrightarrow{name,*} s'$, such that $IT(s) = IT(s')$ and $s' = R[r]$ where $R = L[A[\cdot]]$ is a reduction context and $IT(L[r]) = (\lambda x.t_1)$, and

r is an abstraction. It is obvious that $IT(s')|_{p.2} = t_2$ and that $R = L[A'[[\cdot]\ r_2]]$. Thus $s' = L[A'[((\lambda x.r_1)\ r_2)]] \xrightarrow{name,beta} L[A'[r_1[r_2/x]] = s''$. Now one can verify that $IT(s'') = T$ must hold. ∎

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

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

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

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

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

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

5. Translation $N : L_{name} \to L_{lazy}$

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

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

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

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

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

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

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

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

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

- $N(\texttt{letrec}\ x_1 = s_1, \dots, x_n = s_n\ \texttt{in}\ r) = ((\lambda x_1. \dots x_n.(N(r)))\ U_1\ \dots\ U_n)$
  $$\begin{array}{lrcl} \text{where} & U_i & = & (\lambda x_1, \dots, x_n.x_i\ x_1 \dots x_n)\ X'_1\ \dots\ X'_n, \\ & X'_i & = & \lambda x_1 \dots x_n.F_i(x_1 x_1 \dots x_n) \dots (x_n x_1 \dots x_n), \\ & F_i & = & \lambda x_1, \dots, x_n.N(s_i). \end{array}$$
- $N(s_1\ s_2) = (N(s_1)\ N(s_2))$
- $N(\lambda x.s) = \lambda x.N(s)$
- $N(x) = x.$

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

Convergence equivalence of the translation N follows by inspecting the relation between L_{name}- and the translated L_{lazy}-reductions. The full proof can be found in [SS10]

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

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

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

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

Proof. Since N is convergence equivalent (Proposition 5.3) and compositional by Lemma 5.4, we derive that N is adequate (see [SS08b] and Section 2). ∎

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

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

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

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

Proof. We first show that for all expressions $s \in L_{name}$: $s \sim_{name} \iota(N(s))$. Since N is the identity mapping on `letrec`-free expressions of L_{name} and $N(s)$ is `letrec`-free, we have $N(\iota(N(s))) = N(s)$. Hence adequacy of N (Proposition 5.5) implies $s \sim_{name} \iota(N(s))$. Theorem 3.8 shows $N(s) \sim_{lazy} v$ and Proposition 5.5 show that $\iota(v) \sim_{name} \iota(N(s)) \sim_{name} s$. Finally, Theorem 4.10 shows the claim. ∎

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

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

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

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

6. On Simulation in L_{need}

First we show that finite simulation (see [SS08a]) is correct for L_{need}:

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

Proof. The $\Leftarrow$ direction is trivial. We show the nontrivial part. Assume that for all closed abstractions r and for $r = \Omega$: $s\ r \leq_{need} t\ r$. Then we transfer the problem to L_{lazy} as follows: $N(s)$ and $N(t)$ are closed expressions in L_{lazy}. Since the translation N is surjective, every closed L_{lazy}-expression is in the image of N. Thus for every closed L_{lazy}-expression r' that is an abstraction or Ω, there is some L_{need}-expression r, such that $N(r) = r'$. We have $N(s)\ r'{\downarrow} \implies N(t)\ r'{\downarrow}$, since $N(s\ r) = (N(s)\ N(r))$, and since N is fully abstract. We can apply Proposition 3.10 and obtain $N(s) \leq_{lazy} N(t)$. Now Theorem 5.9 shows $s \leq_{need} t$. ■

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

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

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

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

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

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

Proof. This follows from Lemma 6.2. The complete proof can be found in [SS10]. ■

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

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

Proof. Let $\leq_{need,0}$ the restriction of $\leq_{need}$ to closed expressions. It is easy to verify that $\leq_{need,0} \subseteq [\leq_{need,0}]_{need}$ and thus for closed expressions s, t: $s \leq_{need} t \implies s \leq_{b,need} t$. For the other direction let $s \leq_{b,need} t$. The criterion in Proposition 6.4 then implies that for all $n \geq 0 : s\ r_1\ \dots\ r_n \downarrow_{need} \implies t\ r_1\ \dots\ r_n \downarrow_{need}$, where r_i are closed letrec-free abstractions or Ω. Full-abstraction of $N \circ W$ (see Theorem 5.9) implies that $N(W(s\ r_1\ \dots\ r_n)) \downarrow_{lazy} \implies N(W(t\ r_1\ \dots\ r_n)) \downarrow_{lazy}$. Since N and W translate applications into applications, this also shows that $N(W(s))\ N(W(r_1))\ \dots\ N(W(r_n)) \downarrow_{lazy} \implies N(W(t))\ N(W(r_1))\ \dots\ N(W(r_n)) \downarrow_{lazy}$. Moreover, since every L_{lazy}-abstractions is an $N \circ W$-image of a letrec-free abstraction, we also conclude that $N(W(s)) \leq_{b,lazy,3} N(W(t))$. Now Theorem 3.7 and full abstraction of $N \circ W$ finally show $s \leq_{need} t$. ∎

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

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

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

For open expressions, we can lift the properties from L_{lazy}, which also follows from full abstraction of $N \circ W$ and from Lemma 3.9.

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

The results above imply the following theorem:

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

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

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

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

For the sake of completeness we show that our results are transferable to the call-by-need lambda calculus with letrec of [Ari97]. The syntax is identical to the calculus L_{need}, but the standard reduction strategy of [Ari97] differs from our normal order reduction. In particular [Ari97] do not provide a standard reduction strategy but an equational system from which we will derive a standard reduction.

We will show that the normal order reduction and the standard reduction corresponding to the equational system of [Ari97] are interchangeable and thus define the same notion of contextual equivalence. As a further result we show that bisimilarity can also be based on the strategy according to [Ari97] and coincides with contextual equivalence.

We recall the standard reduction strategy of [Ari97]. We will denote the notions related to Ariola & Felleisen's calculus with a prefix or mark "AF", if necessary. First we introduce AF-evaluation contexts R_{AF} that play a role similar to our reduction contexts:

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

In Figure 3 the standard reductions (abbreviated as AF-reduction) of [Ari97, Section 8] are shown where L is an $\mathcal{L}$-context as introduced in Sect. 3.2 and $R_{AF,i}, R'_{AF}, R''_{AF}$ are R_{AF}-contexts. The calculus of [Ari97] uses the notion of a black hole which represents a cyclic dependency of the form $\texttt{letrec}\ x_1 = R_{AF}[x_n], x_2 = R_{AF}[x_1], \ldots x_n = R_{AF}[x_1]$. In contrast to [Ari97], we do not consider a black hole to be an answer and therefore do not copy it in (deref) rules. This reflects the authors' intention, as shown by a similar copy restriction in [Ari94].

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

Figure 3: Reduction rules defining $\xrightarrow{AF}$

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

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

In [SS10] we prove:

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

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

It remains to show that $\leq_{b,AF}^{o} = \leq_{AF}$. As a first step we derive an alternative characterization of $\leq_{b,AF}$. The proof can be found in [SS10].

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

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

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

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

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

8. Conclusion

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

Acknowledgement

The authors thank the anonymous reviewers for their valuable comments.

References

[Abr90] S. Abramsky. The lazy lambda calculus. In D. A. Turner (ed.), *Research Topics in Functional Programming*, pp. 65–116. Addison-Wesley, 1990.

[Abr93] S. Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. *Inf. Comput.*, 105(2):159–267, 1993.

[Ari94] Z. M. Ariola and J. W. Klop. Cyclic Lambda Graph Rewriting. In *Proc. IEEE LICS*, pp. 416–425. IEEE Press, 1994.

[Ari95] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *POPL'95*, pp. 233–246. ACM Press, San Francisco, California, 1995.

[Ari97] Z. M. Ariola and M Felleisen. The call-by-need lambda calculus. *J. Funct. Programming*, 7(3):265–301, 1997.

[Ari02] Z. M. Ariola and S. Blom. Skew confluence and the lambda calculus with letrec. *Annals of Pure and Applied Logic*, 117:95–168, 2002.

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

[Fel91] M. Felleisen. On the expressive power of programming languages. *Science of Computer 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.

[How89] D. Howe. Equality in lazy computation systems. In *Proc. IEEE LICS*, 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 *Proc. IEEE LICS*, pp. 82–91. 1994.

[Man10] M. Mann and M. Schmidt-Schauß. Similarity implies equivalence in a class of non-deterministic call-by-need lambda calculi. *Information and Computation*, 208(3):276 – 291, 2010.

[Mar98] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *J. Funct. Programming*, 8:275–317, 1998.

[SS07] M. Schmidt-Schauß. Correctness of copy in calculi with letrec. In *Term Rewriting and Applications (RTA-18)*, *LNCS*, vol. 4533, pp. 329–343. Springer, 2007.

[SS08a] M. Schmidt-Schauß and E. Machkasova. A finite simulation method in a non-deterministic call-by-need calculus with letrec, constructors and case. In *Proc. of RTA 2008*, no. 5117 in LNCS, pp. 321–335. Springer-Verlag, 2008.

[SS08b] M. Schmidt-Schauß, J. Niehren, J. Schwinghammer, and D. Sabel. Adequacy of compositional translations for observational semantics. In *5th IFIP TCS 2008*, *IFIP*, vol. 273, pp. 521–535. Springer, 2008.

[SS09a] M. Schmidt-Schauß, E. Machkasova, and D. Sabel. Counterexamples to simulation in non-deterministic call-by-need lambda-calculi with letrec. Frank report 38, Inst. f. Informatik, Goethe-University, Frankfurt, 2009.

[SS09b] 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.

[SS10] M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Simulation in the call-by-need lambda-calculus with letrec. Frank report 40, Inst. f. Informatik, Goethe-University, Frankfurt, 2010.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc-nd/3.0/`.