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

Manfred Schmidt-Schauss[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}@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

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

1 Introduction and Related Work

The construction of compilers and the compilation of programs in higher level, expressive programming languages is an important process in computer science that is a highly sophisticated engineering task. Unfortunately there remains a gap between theory and practice. Usually compilers incorporating lots of complicated transformations and optimizations are built with only a partial knowledge about correctness issues. This gap increases with the number of features, such as higher-order functions, concurrency, store, and system- or user-interaction. The ability to reason about program equivalence in the presence of non-determinism opens a door to a rigorous handling of these features.
We study these issues using a call-by-need lambda-calculus L with data structures and non-determinism that in addition has letrec allowing cyclic binding dependencies. This may have applications for concurrent Haskell [Pey03,PGF96], and also for other functional programming languages [Han96].

Our language L comes with a rewrite semantics (a small-step semantics) that is more appropriate to investigate non-determinism than a big-step semantics, since it explicitly models interleaving and atomicity. On top of the operational semantics we define a contextual semantics with may-convergence, which is maximal, since all expressions that cannot be distinguished by observations are identified.

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

Unfortunately the approach in [Plo75] cannot be applied to systems with cyclic dependencies (such as `letrec`) or with non-determinism, since it requires confluence of transformations which fails in such systems (see [AW96]). Some alternative approaches include restrictions on cyclic substitution [AK97] or considering terms up to infinite unwindings of cycles [AB02].

Investigations of correctness (also called meaning-preservation) for a call-by-value system of mutually recursive components with applications to modules and linking were undertaken in [MT00,Mac02] where a proof method based on diagrams called *lift* and *project* was introduced. The diagram approach was later extended and generalized in [WDK03] in an abstract setting. Another approach based on multihole contexts was used for a call-by-name system of mutually recursive components in [Mac07]. However, the diagram-based and context-based approaches above require that all normal-order reductions preserve behavior of a term, which is not the case for (choice)-reductions. Contextual equivalence for non-deterministic call-by-need calculi was investigated in [KSS98,SSS07a,SSSS08] using the method of forking and commuting diagrams, and in [MSC99] using abstract machine-reductions.

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

In this paper we propose a method of finite simulation, i.e., a simulation with a finite depth, and prove its correctness by a new proof technique that uses approximations. The simulation method constructs a set of answer-terms for

a given expression. These sets are then compared for various closed expressions in order to show contextual equivalence. An answer is either an abstraction or an constructor-expression, built from constructors, Ω, and abstractions, such as partial lists. Consider the two (non-convertible) expressions s, t where $s =$ `repeat True` is a nonending list and t is recursively defined by $t =$ `choice` $\bot$ (`Cons True` t). The latter will evaluate, depending on the choices, to $\bot$, (`Cons True` $\bot$), (`Cons True` (`Cons True` $\bot$)), $\ldots$. The expressions s, t are equivalent w.r.t. observational equivalence based on may-convergence. Our simulation method permits to show their equivalence solely on the basis of the (approximative) answers that can be derived from each expression.
The proof of validity of the proposed method for our calculus L (see section 2) requires several steps. The first step is to investigate the correctness of several reductions and transformations in L. Note that the normal-order reduction in the language L treats chains of variable-variable bindings as transparent for several reductions. This is crucial for constructing correctness proofs which otherwise may not even be possible, since the measures for inductions are insufficient. A context lemma and standardization of reductions are proved. The second step is a transfer from L to the calculus L_S (see section 4), which has the same contextual equivalence as L, but simpler reduction rules, e.g. variable-variable bindings are now opaque. The third step (see Sections 5 – 7) is to define the computation of answers from a closed expression, and to prove criteria for contextual equivalence on the basis of the answer sets. We also provide a method to analyze contextual equivalence and preorder of answers. In particular, we show that abstractions can be compared based on applying them to all closed answers or to Ω. As an application of this technique, we show that `choice` (see Section 8) has useful algebraic properties, such as idempotency, commutativity and associativity, for all expressions, including open ones.
Missing proofs can be found in [SSM08].

2 The Calculus L

2.1 Syntax and Reductions of the Functional Core Language L

We define the calculus L consisting of a language $\mathcal{L}(\mathrm{L})$, its reduction rules, the normal order reduction strategy, and contextual equivalence. L is the calculus considered in [SSSS04] and an extension by `choice` of the one in [SSSS08]. The rules of the calculus limit copying of abstractions and prohibit copying of constructor expressions, thus limiting the level of complexity of proofs. There are finitely many types, and for every type T there are finitely many, say $\#(T)$, constants called constructors $c_{T,i}, i = 1, \ldots, \#(T)$, each with an arity $\mathrm{ar}(c_{T,i}) \geq 0$. The syntax for expressions E is as follows:

$$\begin{aligned} E ::= {} & V \mid (c\ E_1 \ldots E_{\mathrm{ar}(c)}) \mid (\mathtt{seq}\ E_1\ E_2) \mid (\mathtt{case}_T\ E\ \mathit{Alt}_1 \ldots \mathit{Alt}_{\#(T)}) \mid (E_1\ E_2) \\ & (\mathtt{choice}\ E_1\ E_2) \mid (\lambda\ V.E) \mid (\mathtt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \mathtt{in}\ E) \\ \mathit{Alt} ::= {} & (\mathit{Pat}\ \rightarrow\ E) \qquad \mathit{Pat} ::= (c\ V_1 \ldots V_{\mathrm{ar}(c)}) \end{aligned}$$

where E, E_i are expressions, V, V_i are variables, and c denotes a constructor. Expressions ($\texttt{case}_T \ldots$) have exactly one alternative for every constructor of type T. We assign the names *application*, *abstraction*, *constructor application*, `seq`-*expression*, `case`-*expression*, or `letrec`-*expression* to the expressions $(E_1\ E_2)$, $(\lambda V.E)$, $(c\ E_1 \ldots E_{\mathrm{ar}(c)})$, $(\texttt{seq}\ E_1\ E_2)$, $(\texttt{case}_T\ E\ Alt_1 \ldots Alt_{\#(T)})$, $(\texttt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \texttt{in}\ E)$, respectively. A *value* v is defined as either an abstraction or a constructor application (with any subexpressions).
We assume that variables V_i in `letrec`-bindings are all distinct, that the bindings can be interchanged, and that there is at least one binding. `letrec` is recursive, i.e., the scope of x_j in $(\texttt{letrec}\ x_1 = E_1, \ldots, x_j = E_j, \ldots\ \texttt{in}\ E)$ is E and all expressions E_i. Free and bound variables in expressions and α-renaming are defined using the usual conventions. The set of free variables in t is denoted as $FV(t)$. For simplicity we use the distinct variable convention, i.e., all bound variables in expressions are assumed to be distinct, and free variables are distinct from bound variables. The reduction rules are assumed to implicitly rename bound variables in the result by α-renaming if necessary. We will use some obvious abbreviations of the syntax. E.g. $\{x_i = x_{i+1}\}_{i=m}^{n}$ abbreviates $x_m = x_{m+1}, \ldots, x_n = x_{n+1}$.

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

Definition 2.2 (Reduction Rules of the Calculus L**).** *The (base) reduction rules for the calculus and language* L *are defined in figures 1 and 2, where the labels S, V are to be ignored in this subsection, but will be used in subsection 2.2. The abbreviation Env means a set of bindings. The reduction rules can be applied in any context. 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), (lcase), (lapp), (lseq) is called (lll).*
Reductions (and transformations) are denoted using an arrow with super and/or subscripts: e.g. $\xrightarrow{llet}$. Transitive closure of reductions is denoted by a $+$, reflexive transitive closure by a $*$. E.g. $\xrightarrow{*}$ is the reflexive, transitive closure of $\rightarrow$.

2.2 Normal Order Reduction and Contextual Equivalence

The normal order reduction strategy of the calculus L is a call-by-need strategy, which is a call-by-name strategy adapted to sharing. The labeling algorithm in figure 3 will detect the position to which a reduction rule is applied according to the normal order. It uses the labels: S (subterm), T (top term), V (visited), W (visited, no copy-target). For a term s the labeling algorithm starts with s^T, where no other subexpression in s is labeled, and exhaustively applies the rules in figure 3. The algorithm may terminate with a failure if a relabeling occurs, and

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

Fig. 1. Reduction rules, part a

otherwise with success, which indicates a potential normal-order redex, usually as the direct superterm of the S-marked subexpression.

Definition 2.3 (Normal Order Reduction of L**).** *Let t be an expression. Then a single normal order reduction step $\xrightarrow{no}$ is defined by first applying the labeling algorithm to t. If the labeling algorithm terminates successfully, then one of the rules in figures 1 and 2 has to be applied, if possible, where the labels S, V must match the labels in the expression t. The* normal order redex *is defined as the subexpression to which the reduction rule is applied.*

Definition 2.4. *A reduction context R is any context, such that its hole will be labeled with S or T by the labeling algorithm. A* surface context*, denoted as $\mathcal{S}$, is a context where the hole is not contained in an abstraction. An* application surface context*, denoted as $\mathcal{AS}$, is a surface context where the hole is neither contained in an abstraction nor in an alternative of a case-expression.*

Note that the normal order redex is unique, and that a normal-order reduction is unique with the only exception of (choice).
A *weak head normal form (WHNF)* is either a value v or an expression $(\texttt{letrec}\ Env\ \texttt{in}\ v)$, or $(\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ x_m)$.

Definition 2.5. *A normal order reduction sequence is called an* evaluation *if the last term is a WHNF. For a term t, we write $t\downarrow$ iff there is an evaluation starting*

(case-c)	$(\texttt{case}_T\ (c_i\ \overrightarrow{t})^S\ \ldots ((c_i\ \overrightarrow{y}) \to t) \ldots) \to (\texttt{letrec}\ \{y_i = t_i\}_{i=1}^n\ \texttt{in}\ t)$
	where $n = \mathrm{ar}(c_i) \geq 1$
(case-c)	$(\texttt{case}_T\ c_i^S\ \ldots\ (c_i \to t) \ldots) \to t\quad$ if $\mathrm{ar}(c_i) = 0$
(case-in)	$\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{t})^S, \{x_i = x_{i-1}\}_{i=2}^m, Env$
	$\texttt{in}\ C[\texttt{case}_T\ x_m^V\ \ldots ((c_i\ \overrightarrow{z}) \ldots \to t) \ldots]$
	$\to \texttt{letrec}\ x_1 = (c_i\ \overrightarrow{y}), \{y_i = t_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=2}^m, Env$
	$\texttt{in}\ C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^n\ \texttt{in}\ t)]$
	where $n = \mathrm{ar}(c_i) \geq 1$ and y_i are fresh variables
(case-in)	$\texttt{letrec}\ x_1 = c_i^S, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ C[\texttt{case}_T\ x_m^V\ \ldots\ (c_i \to t) \ldots]$
	$\to \texttt{letrec}\ x_1 = c_i, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ \ C[t]\quad$ if $\mathrm{ar}(c_i) = 0$
(case-e)	$\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{t})^S, \{x_i = x_{i-1}\}_{i=2}^m,$
	$u = C[\texttt{case}_T\ x_m^V\ \ldots ((c_i\ \overrightarrow{z}) \to r_1) \ldots], Env\quad \texttt{in}\ r_2$
	$\to \texttt{letrec}\ x_1 = (c_i\ \overrightarrow{y}), \{y_i = t_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=2}^m,$
	$u = C[(\texttt{letrec}\ z_1 = y_1, \ldots, z_n = y_n\ \texttt{in}\ r_1)], Env\quad \texttt{in}\ r_2$
	where $n = \mathrm{ar}(c_i) \geq 1$ and y_i are fresh variables
(case-e)	$\texttt{letrec}\ x_1 = c_i^S, \{x_i = x_{i-1}\}_{i=2}^m, u = C[\texttt{case}_T\ x_m^V\ \ldots\ (c_i \to r_1) \ldots], Env\quad \texttt{in}\ r_2$
	$\to \texttt{letrec}\ x_1 = c_i, \{x_i = x_{i-1}\}_{i=2}^m \ldots, u = C[r_1], Env\ \texttt{in}\ r_2$
	if $\mathrm{ar}(c_i) = 0$

Fig. 2. Reduction rules, part b

from t. We also say that t is converging *(or* terminating*). Otherwise, if there is no evaluation of t, we write $t\Uparrow$. A specific representative of non-converging expressions is $\Omega := (\lambda z.(z\ z))\ (\lambda x.(x\ x))$, i.e. $\Omega\Uparrow$. For consistency with our earlier work (e.g. [SSS07b]) the must-divergence notation $\Uparrow$ is used.*

As an example for normal-order reduction, some reductions of Ω: $(\lambda z.(z\ z))\ (\lambda x.(x\ x)) \xrightarrow{no,lbeta} (\texttt{letrec}\ z = \lambda x.(x\ x)\ \texttt{in}\ (z\ z)) \xrightarrow{no,cp} (\texttt{letrec}\ z = \lambda x.(x\ x)\ \texttt{in}\ ((\lambda x'.(x'\ x'))\ z)) \xrightarrow{no,lbeta} (\texttt{letrec}\ z = \lambda x.(x\ x)\ \texttt{in}\ (\texttt{letrec}\ x_1 = z\ \texttt{in}\ (x_1\ x_1))) \xrightarrow{no,llet} (\texttt{letrec}\ z = \lambda x.(x\ x), x_1 = z\ \texttt{in}\ (x_1\ x_1)) \longrightarrow \ldots$.

Definition 2.6 (contextual preorder and equivalence). *Let s, t be terms. Then:*

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

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

3 Correctness of Reductions and Transformations

Theorem 3.1. *All the reductions (viewed as transformations) in the base calculus* L *with the exception of (choice) maintain contextual equivalence, i.e., when-*

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

The labeling rules can be applied in any context

Fig. 3. Labeling algorithm for L

ever $t \xrightarrow{a} t'$, with $a \in \{cp, lll, case, seq, lbeta\}$, then $t \sim_c t'$. The same holds for all transformations in figure 4. Moreover, $s \xrightarrow{choice} t$ implies $t \leq_c s$.

We define non-reduction transformations in Figure 4. Some transformations have two or more forms, e.g. (ve1) and (ve2). The side condition for (abs2) guarantees finiteness of (abs2) sequences. The transformations are used later either for the pre-evaluation or to aid correctness proofs.

Proposition 3.2. *The expression Ω is the least element w.r.t. $\leq_c$, and for every closed expression s with $s\Uparrow$, the equation $s \sim_c \Omega$ holds.*

We summarize correctness of transformations and decreasing property of (choice) in the *standardization* result, which shows that reduction sequences can be standardized using normal-order reduction.

Theorem 3.3 (Standardization). *If $t \xrightarrow{*} t'$ where t' is a WHNF and the sequence $\xrightarrow{*}$ consists of any reduction from L in figures 1 and 2 and of transformation steps from figure 4, then $t\downarrow$.*

4 A Simpler Calculus

We define a simpler calculus L_S that is used to produce a set of values of any closed expression. It is formulated such that a so-called *pre-evaluation* can be defined and shown to be a correct tool to prove contextual preorder and contextual equivalence of expressions in almost the same way as the simulation method would do it. The calculus L_S does not use variable-binding chains for reduction steps, and permits also copying expressions of the form $(c\ x_1 \ldots x_n)$, where x_i are variables. Such expressions are called *cv-expression.*
The rules of the calculus L_S are defined in figure 6. We use labels S, T, V indicating the normal order redex The labeling algorithm in 5 starts with t^T, where no subexpression of t is labeled, and uses the rules exhaustively, which can be applied in any context.

(ve1)	$(\texttt{letrec}\ x = y, x_1 = t_1, \ldots, x_n = t_n\ \texttt{in}\ r) \rightarrow (\texttt{letrec}\ x_1 = t'_1, \ldots, x_n = t'_n\ \texttt{in}\ r')$
	where $t'_i = t_i[y/x], r' = r[y/x], n \geq 0$ and if $x \neq y$
(ve2)	$(\texttt{letrec}\ x = y\ \texttt{in}\ s) \rightarrow s[y/x]$ if $x \neq y$
(abs1)	$(\texttt{letrec}\ x = c\ \overrightarrow{t}, Env\ \ \texttt{in}\ s) \rightarrow (\texttt{letrec}\ x = c\ \overrightarrow{x}, \{x_i = t_i\}_{i=1}^{\mathrm{ar}(c)}, Env\ \ \texttt{in}\ s)$
	where $\mathrm{ar}(c) \geq 1$ and for $1 \leq i \leq \mathrm{ar}(c)$ $x_i \in FV(Env)$
(abs2)	$(c\, t_1 \ldots t_n) \rightarrow (\texttt{letrec}\ x_1 = t_1, \ldots, x_n = t_n\ \texttt{in}\ (c\, x_1 \ldots x_n))$
	where at least one of t_i is not a variable
(cpcx-in)	$(\texttt{letrec}\ x = c\ \overrightarrow{t}, Env\ \texttt{in}\ C[x])$
	$\rightarrow (\texttt{letrec}\ x = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^{\mathrm{ar}(c)}, Env\ \texttt{in}\ C[c\ \overrightarrow{y}])$
(cpcx-e)	$(\texttt{letrec}\ x = c\ \overrightarrow{t}, z = C[x], Env\ \texttt{in}\ t)$
	$\rightarrow (\texttt{letrec}\ x = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^{\mathrm{ar}(c)}, z = C[c\ \overrightarrow{y}], Env\ \texttt{in}\ t)$
(gc1)	$(\texttt{letrec}\ \{x_i = s_i\}_{i=1}^{n}, Env\ \texttt{in}\ t) \rightarrow (\texttt{letrec}\ Env\ \texttt{in}\ t)$
	if for all $i : x_i$ does not occur in Env nor in t
(gc2)	$(\texttt{letrec}\ \{x_i = s_i\}_{i=1}^{n}\ \texttt{in}\ t) \rightarrow t$ if for all $i : x_i$ does not occur in t
(ucp1)	$(\texttt{letrec}\ Env, x = t\ \texttt{in}\ S[x]) \rightarrow (\texttt{letrec}\ Env\ \texttt{in}\ S[t])$
(ucp2)	$(\texttt{letrec}\ Env, x = t, y = S[x]\ \texttt{in}\ r) \rightarrow (\texttt{letrec}\ Env, y = S[t]\ \texttt{in}\ r)$
(ucp3)	$(\texttt{letrec}\ x = t\ \texttt{in}\ S[x]) \rightarrow S[t]$
	where in the (ucp)-rules, x has at most one occurrence in $S[x]$ and no occurrence in Env, t, r; and S is a surface context
(cpbot1)	$(\texttt{letrec}\ x = \Omega, Env\ \texttt{in}\ C[x]) \rightarrow (\texttt{letrec}\ x = \Omega, Env\ \texttt{in}\ C[\Omega])$
(cpbot2)	$(\texttt{letrec}\ x = \Omega, y = C[x], Env\ \texttt{in}\ r) \rightarrow (\texttt{letrec}\ x = \Omega, y = C[\Omega], Env\ \texttt{in}\ r)$

Fig. 4. Transformations in L calculus

$$(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C[x^S]) \rightarrow (\texttt{letrec}\ x = s^S, Env\ \texttt{in}\ C[x^V])$$
$$(\texttt{letrec}\ x = s, y = C[x^S], Env\ \texttt{in}\ r) \rightarrow (\texttt{letrec}\ x = s^S, y = C[x^V], Env\ \texttt{in}\ r)$$
$$\text{if } C \neq [.]$$

The rules for $(\texttt{letrec}\ Env\ \texttt{in}\ t)^T$, $(s\ t)$, $(\texttt{seq}\ s\ t)$ and $(\texttt{case}\ s\ alts)$ are as for L

Fig. 5. Labeling rules of L_S

An L_S-*WHNF* is defined as v or $(\texttt{letrec}\ Env\ \texttt{in}\ v)$, where v is an abstraction or a cv-expression. It is easy to see that every L_S-WHNF is also an L-WHNF, and that for every L-WHNF t, there is an L_S-WHNF t' with: $t \xrightarrow{L_S,no,*} t'$ using only (*abs*), (*lll*), and (*cp*).
Using diagrams and an induction on the length of a reduction sequence, the equivalence is shown in [SSM08]:

Theorem 4.1. *Let s be an expression. Then $s{\downarrow}_{L_S} \Leftrightarrow s{\downarrow}_L$.*

5 Pre-Evaluation of Expressions

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

(cp-in)	$(\texttt{letrec}\ x = v^S, Env\ \ \texttt{in}\ C[x^V]) \to (\texttt{letrec}\ x = v, Env\ \ \texttt{in}\ C[v])$
	where v is an abstraction or a cv-expression
(cp-e)	$(\texttt{letrec}\ x = v^S, Env, y = C[x^V]\ \texttt{in}\ r) \to (\texttt{letrec}\ x = v, Env, y = C[v]\ \texttt{in}\ r)$
	where v is an abstraction or a cv-expression
(abs)	$(c\ t_1 \dots t_n)^{S \vee T} \to (\texttt{letrec}\ x_1 = t_1, \dots, x_n = t_n\ \texttt{in}\ (c\ x_1 \dots x_n))$
	if $(c\ t_1 \dots t_n)$ is not a cv-expression
(lbeta), (seq-c), (case), (choice), (lll) are as in L	

Fig. 6. Reduction rules of L_S

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

Definition 5.1. *A* pseudo-value *is an expression built from $\circledcirc$, constructors, and abstractions, and an* answer *is a pseudo-value not equal to $\circledcirc$.*

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

Example 5.2. The expression $(\texttt{letrec}\ x = (\texttt{Cons}\ \texttt{True}\ x)\ \texttt{in}\ x)$ has the following resulting answers: $(\texttt{Cons}\ \circledcirc\ \circledcirc)$, $(\texttt{Cons}\ \texttt{True}\ \circledcirc)$, $(\texttt{Cons}\ \circledcirc\ (\texttt{Cons}\ \texttt{True}\ \circledcirc))$, $(\texttt{Cons}\ \texttt{True}\ (\texttt{Cons}\ \texttt{True}\ \circledcirc))$, ...

The approximation reduction $\xrightarrow{A}$ is based upon L_S-reduction:

Definition 5.3. *Let s be a closed expression. We define the approximation reduction $\xrightarrow{A}$as follows:*
Then $s \xrightarrow{A} v$ holds for some closed answer v iff there is a reduction starting from $(\texttt{letrec}\ x = s\ \texttt{in}\ x)$ to v using the following intermediate steps.

1. *$(\texttt{letrec}\ x = s\ \texttt{in}\ x) \xrightarrow{*} s'$ using an L_S-evaluation to a WHNF s'. Continuing from s', we perform any number of L_S-reductions in application surface contexts (non-deterministically), where the target variables of (cp) are also in application surface contexts.*

2. *Perform any number of copy-reductions into the "in"-expression. Here the target variable of (cp) may be in any context C.*
3. *The last step is to remove the top-letrec-environment, and to replace all remaining let-bound variables in the "in"-expression by $\circledcirc$. The resulting expression is now either $\circledcirc$ or one of the desired answers v.*

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

Lemma 5.4. *Let s be a closed expression and $v \in ans(s)$. Then $v \leq_c s$.*

Proof. This follows from the correctness of the transformations proved in the previous sections, from decreasingness of (choice) (see Theorem 3.1) and from the fact that $\circledcirc \sim_c \Omega$ is the least element w.r.t. $\leq_c$ (see Proposition 3.2). □

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

Theorem 5.5. *Let R be a reduction context, s be a closed expression such that $R[s]\downarrow$. Then there is an answer v with $(\texttt{letrec}\ x = s\ \texttt{in}\ x) \xrightarrow{A} v$, such that $R[v]\downarrow$. Note that $s \sim_c (\texttt{letrec}\ x = s\ \texttt{in}\ x)$ (see Theorem 3.1).*

Proof. In the proof we always refer to the calculus L_S.
Let R be a reduction context and s be a closed expression. Let *Red* be a normal-order reduction of $R[(\texttt{letrec}\ x = s\ \texttt{in}\ x)] \xrightarrow{no} r_1 \ldots \xrightarrow{no} r_n$, where r_n is a WHNF, and n is the number of normal-order reductions. In every expression of *Red*, the bindings inherited from $x = s$ can be identified in every r_i by labeling them with †. Thus we label letrec-bound variables and the bound expression in surface positions that are derived from s. An important invariant is that for all †-labeled bindings $y_i = a_i$, and all free variables y in a_i, y is also a †-labeled variable, which follows by induction on the length of the reduction from the fact that s is closed. If a WHNF w of $R[(\texttt{letrec}\ x = s\ \texttt{in}\ x)]$ is reached, then from the WHNF we can gather all the †-labeled bindings in the top level letrec environment of w, and construct the expression $s' := (\texttt{letrec}\ Env\ \texttt{in}\ x)$, where we denote $x_1 = s_1, \ldots, x_m = s_m$ by *Env* and where $x_1 = x$ for convenience. Now we compute one possible answer v from s' as required by our claim as follows. We perform $n+1$ of the following macro-copy-steps within the environment *Env* into the "in"-expression:
One step consists of replacing all occurrences of x_i by s_i in the "in"-expression (initially x) for all $x_i = s_i$ in *Env* s.t. s_i is an abstraction or a cv-expression. We do this in parallel for every letrec-bound variable, which is the same as applying the substitution σ that is formed from *Env*. This is repeated $n+1$ times. The last step is to remove the top-environment, and to replace all letrec-bound variables in the in-expression by $\circledcirc$. This may produce either $\circledcirc$, or the desired answer v, and we have $s' \xrightarrow{A} v$ according to Definition 5.3. Since we assumed that a WHNF is reached, and s was in a reduction context before, it is not possible that only $\circledcirc$ is reached, since the initial variable x was in a reduction context and there must be at least one normal-order copy into x. Thus at least one of the macro-copy steps will replace x by a constructor-expression or an abstraction.

Now we have to show that $R[v]{\downarrow}$. We start by rearranging the normal-order reduction *Red* of $R[(\texttt{letrec}\ x = s\ \texttt{in}\ x)]$, such that all the reductions that are within the †-labeled *Env* are performed first, i.e., $R[(\texttt{letrec}\ x = s\ \texttt{in}\ x)] \xrightarrow{*} R[s'] \xrightarrow{no,*} r_n$. It is easy to see that the $R[(\texttt{letrec}\ x = s\ \texttt{in}\ x)] \xrightarrow{*} R[s']$ starts with an L_S-normal-order reduction to a WHNF, since x is "demanded" first. The subsequent reductions remain in application surface positions. The reduction $R[s'] \xrightarrow{no,*} r_n$ is normal-order, and has length at most n. The reduction sequence *Red* is a mixture of reduction steps within †-labeled components, or reduction steps that modify the non-†-labeled components. All reductions are in surface contexts. Hence the †-reductions can be shifted to the left over non-†-reductions, since they are independent.
Now we focus on $R[s'] \xrightarrow{no,*} r_n$ of length at most n. We have to show that for $s' \xrightarrow{*} v$, we also have $R[v] \xrightarrow{no,*} u$, where u is a WHNF. The term v and its descendents can be represented using $\phi_{\geq k}$, which is defined as follows: $\phi_{\geq k}(r)$ denotes r modified by the following operations: first k applications of σ are performed (the substitution corresponding to the s-environment *Env*), then any number of (cp)-steps using *Env* and variables in r as target variables, and as a final step $[\circledcirc/x_i]$-replacements in r for all let-bound variables in *Env*. Now we have to show that $(\phi_{\geq n+1} R[s']){\downarrow}$. The steps $\xleftarrow{no,a} . \xrightarrow{\phi_{\geq k}}$ can be switched, i.e. replaced by $\xrightarrow{\phi_{\geq k-1}} . \xleftarrow{\rho,*}$, where $\rho = \{(no,a),(abs),(lll),(cp),(cpbot),(ve)\}$. Using this commutation, it is easily shown by induction that, finally, we obtain a WHNF that is the result of a macro-copy reduction using $\phi_{\geq i}$, where $i \geq 1$. The argument now is that the replaced positions do not contribute to the WHNF w, hence it remains a WHNF after applying $\phi_{\geq 1}$. This means there is a reduction sequence $R[v] \xrightarrow{*} w'$, where w' is a WHNF. Finally, the standardization theorem 3.3 shows that $R[v]{\downarrow}$. □

6 Least Upper Bounds and Sets of Answers

Definition 6.1. *Let W be a set of expressions, and let t be an expression. Then t is a* lub of W *iff $\forall u \in W : u \leq_c t$, and for every s with $\forall u \in W : u \leq_c s$, it is $t \leq_c s$.*
The expression t is called a contextual lub (club) *of W, iff for all contexts C: $C[t]$ is a lub of $\{C[r] \mid r \in W\}$. The notation is $t \in club(W)$. An expression t is called a* linear club (lclub) *of W if the set W is a $\leq_c$-ascending chain of expressions. The notation is $t \in lclub(W)$. The set of all t such that $t \in lclub(A)$ for some $A \subseteq W$ is denoted as $sublclub(W)$.*

Example 6.2. The following $\leq_c$-ascending chain $\lambda x_1.\Omega$, $\lambda x_1, x_2.\Omega$, $\ldots \lambda x_1, \ldots, x_n.\Omega$ has YK as lclub, which is equivalent to the value $\lambda x.(Y\ K)$. The combinators are defined as $Y = \lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x))$, and $K = \lambda x, y.x$.

Easy arguments show that the following holds:

Lemma 6.3. *For any closed expression s: $s \sim_c \Omega$ iff $ans(s) = \emptyset$. Otherwise, if $s \not\sim_c \Omega$, then $s \in club(ans(s))$.*

This yields an immediate criterion for contextual preorder:

Corollary 6.4. *Let s, t be closed expressions. If for all $w \in ans(s)$ we also have $w \leq_c t$, then $s \leq_c t$.*

The following useful sufficient condition immediately follows from the corollary:

Theorem 6.5. *Let s, t be closed expressions. If for all $v \in ans(s)$ there is some $w \in sublclub(ans(t))$ with $v \leq_c w$, then $s \leq_c t$.*

Note that a simplistic subset-condition for answer-sets is insufficient for $s \leq_c t$:

Example 6.6. Let $s := \lambda x.Y\ K$, $f\ z :=$ `choice` z (`letrec` $u = f\ z$ `in` $\lambda x.u$) and $t := f\ \Omega$, where an explicit definition of f is $f = Y\ (\lambda g.\lambda z.$`choice` Ω (`letrec` $u = g\ z$ `in` $\lambda x.u$)). Then for every $v \in ans(t)$, we have $v <_c s$. However, it is easy to see that $s \sim_c t$, since $\lambda x.Y\ K$ is the club of the ascending chain of values in $ans(t)$.

The following is obvious using contexts:

Proposition 6.7. $(c\ s_1 \dots s_n) \leq_c (c\ t_1 \dots t_n) \Leftrightarrow s_i \leq_c t_i$ *for all i.*

7 Criteria for Abstractions

Besides the trivial method to compare two abstractions $\lambda x.s$ and $\lambda x.t$ by α-equivalence, perhaps combined with other correct transformations, we give a stronger condition for $\lambda x.s \leq_c \lambda x.t$ that is based on applying the abstractions to all possible pseudo-value arguments not using the criteria for all contexts. The following is proved in [SSM08].

Lemma 7.1. *[Context Lemma for Closing Reduction Contexts] Let s, t be expressions. Then $s \leq_c t$ iff for all reduction contexts R: if $R[s], R[t]$ are closed and $R[s]{\downarrow}$, then also $R[t]{\downarrow}$,*

In a *pseudo-value environment Env* every bound term is a (closed) pseudo-value.

Proposition 7.2. *Let s, t be two expressions. Then $s \leq_c t$ iff for all pseudo-value environments Env: if* (`letrec` *Env* `in` s), (`letrec` *Env* `in` t) *are closed then* (`letrec` *Env* `in` s) $\leq_c$ (`letrec` *Env* `in` t).

Proof. In order to show the non-trivial direction, we will use Lemma 7.1. Let R be a reduction context such that $R[s], R[t]$ are closed and such that $R[s]{\downarrow}$. It is no restriction to assume that $R[\cdot]$ is of the form (`letrec` Env_1, Env_2 `in` $R'[\cdot]$), where Env_1 binds all the variables in $FV(s,t)$, and (`letrec` Env_1 `in` $[\cdot]$) is closed. Since $s' :=$ (`letrec` Env_1, Env_2 `in` $R'[s]$)$\downarrow$, there is a normal-order reduction *Red* of

s'. In the same way as in the proof of Theorem 5.5, we can evaluate all bindings in Env_1 first, obtaining Env_1' such that $s'' := (\texttt{letrec}\ Env_1', Env_2\ \texttt{in}\ R'[s])\!\downarrow$. The environment Env_1' will be further modified into Env_1'' as follows: every binding $x = r$, where r is not an abstraction and not a cv-expression is changed into $x = \circledcirc$. Again we have $s^{(3)} := (\texttt{letrec}\ Env_1'', Env_2\ \texttt{in}\ R'[s])\!\downarrow$, since the $\circledcirc$-bindings do not influence the normal-order reduction. Let n be the length of a normal-order reduction of $s^{(3)}$. We further modify Env_1'' into $Env_1^{(3)}$ by applying the substitution σ corresponding to Env_1'' at least n times to the environment, and then replacing all remaining occurrences of variables by $\circledcirc$. Similar as in the proof of Theorem 5.5, we argue that $(\texttt{letrec}\ Env_1^{(3)}, Env_2\ \texttt{in}\ R'[s])\!\downarrow$. Using the knowledge about correct transformations, it can be proved using induction that $(\texttt{letrec}\ Env_1^{(3)}, Env_2\ \texttt{in}\ R'[s]) \sim_c (\texttt{letrec}\ Env_1^{(3)}, Env_2\ \texttt{in}\ R'[(\texttt{letrec}\ Env_1^{(3)}\ \texttt{in}\ s)])$, hence $(\texttt{letrec}\ Env_1^{(3)}, Env_2\ \texttt{in}\ R'[(\texttt{letrec}\ Env_1^{(3)}\ \texttt{in}\ s)])\!\downarrow$.
Now we argue the reverse way for t: By the assumption, we have $(\texttt{letrec}\ Env_1^{(3)}, Env_2\ \texttt{in}\ R'[(\texttt{letrec}\ Env_1^{(3)}\ \texttt{in}\ s)]) \leq_c (\texttt{letrec}\ Env_1^{(3)}, Env_2\ \texttt{in}\ R'[(\texttt{letrec}\ Env_1^{(3)}\ \texttt{in}\ t)])$, hence $(\texttt{letrec}\ Env_1^{(3)}, Env_2\ \texttt{in}\ R'[(\texttt{letrec}\ Env_1^{(3)}\ \texttt{in}\ t)])\!\downarrow$. The same argument as above shows that also $(\texttt{letrec}\ Env_1^{(3)}, Env_2\ \texttt{in}\ R'[t])\!\downarrow$. Since $\circledcirc \sim_c \bot$ is the $\leq_c$-least element, and (cp) does not change the $\sim_c$ equivalence class, we also have $(\texttt{letrec}\ Env_1'', Env_2\ \texttt{in}\ R'[t])\!\downarrow$. Since $(\texttt{letrec}\ Env_1'', Env_2\ \texttt{in}\ R'[t])$ can be reached from $(\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ R'[t])$ by reductions that only decrease by $\leq_c$ due to Theorem 3.1, we finally have $(\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ R'[t])\!\downarrow$. Since R was arbitrary, we can apply Lemma 7.1 and obtain that $s \leq_c t$ □

Theorem 7.3. *$\lambda x.s \leq_c \lambda x.t$ iff for all pseudo-values v: $(\lambda x.s)\ v \leq_c (\lambda x.t)\ v$.*

Proof. Follows from Proposition 7.2, since $(\lambda x.s)\ v \sim_c (\texttt{letrec}\ x = v\ \texttt{in}\ s)$ □

8 Finite Simulation Method and Examples

Now we have several criteria to prove $s \leq_c t$ for closed expressions s, t.

1. If $ans(s) \subseteq ans(t)$, then $s \leq_c t$.
2. If for every $v \in ans(s)$, there is some $w \in ans(t)$ with $v \leq_c w$, then $s \leq_c t$.
3. If for every $v \in ans(s)$, there is some $w \in ans(t)$ with $v \leq_c w$ or some $w \in sublclub(ans(t))$ with $v \leq_c w$, then $s \leq_c t$.
4. if $s = c\ s_1 \dots s_n$, $t = c\ t_1 \dots t_n$, and $s_i \leq_c t_i$ for all i, then $s \leq_c t$.
5. if $s = \lambda x.s'$, $t = \lambda x.t'$, and for all pseudo-values v: $s\ v \leq_c t\ v$, then $s \leq_c t$.

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

1. Compute the answer-sets $ans(s)$ and $ans(t)$.

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

If answers-sets are finite, the recursion depth is bounded, and all the involved tests are decidable, the procedure becomes effective. Proving $s \sim t$ for expressions s, t can be done by checking $s \leq_c t$ and $t \leq_c s$.

Example 8.1. Let $s := repeat$ `True`, $t := Y\ (\lambda a.$`choice` Ω (`Cons True` a) where $repeat := Y\ (\lambda r.\lambda x.$`Cons` $x\ (r\ x))$. Then s can be reduced to the answers (`Cons True` (`Cons True` $(\dots($`Cons True` $\Omega))))$ and t can be reduced to the same answers, where we use $\circledcirc \sim_c \Omega$. This implies that $s \sim_c t$.

Example 8.2. Finite simulation can distinguish expressions that differ only by sharing: Let $s :=$ (`letrec` $x =$ `choice True False in` $\lambda y.x$) and let $t = \lambda y.$(`letrec` $x =$ `choice True False in` x). These expressions are contextually different, using the context $C[\cdot] :=$ (`letrec` $z = [\cdot]$ `in if` $(z\ \bot)$ `then` (`if` $(z\ \bot)$ `then True else` $\bot$) `else True`). The answer-sets are: $ans(t) = \{t\}$, $ans(s) = \{\lambda y.$`True`$, \lambda y.$`False`$\}$.

Example 8.3. We are able to prove idempotency, commutativity and associativity for `choice` as a binary operator or all expressions, and hence these identities can be used as program transformation using Proposition 7.2: For instance, for commutativity: every pseudo-value environment Env that closes s and t, we consider (`letrec` Env `in choice` $s\ t$) and (`letrec` Env `in choice` $t\ s$). The first step of the approximation is the choice-reduction. Then the right and left hand side have the same set of answers, hence they are equivalent, which follows from Corollary 6.4. The same can be done for the other identities.

9 Conclusion and Further Research

We have shown that in a call-by-need non-deterministic lambda calculus with letrec, where the proof method of Howe fails to prove correctness of co-inductive simulation, the correctness of finite simulation can be established as a tool with almost the same practical power. Further research is to adapt and extend the methods to an appropriately defined simulation, and to investigate an extension of the tools and methods to a combination of may- and must-convergence.

Acknowledgements
We thank David Sabel for reading several versions of the paper.

References

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

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

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

AW96. Z. M. Ariola and J. W.Klop. Equational term graph rewriting. *Fundamentae Informaticae*, 26(3,4):207–240, 1996.

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

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

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

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

KSS98. A. Kutzner and M. Schmidt-Schauß. A nondeterministic call-by-need lambda calculus. In *ICFP 1998*, pages 324–335. ACM Press, 1998.

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

Mac07. E. Machkasova. Computational soundness of a call by name calculus of recursively-scoped records. In *7th WRS*, ENTCS, 2007.

Man04. M. Mann. Congruence of bisimulation in a non-deterministic call-by-need lambda calculus. In *SOS'04*, BRICS NS-04-1, pages 20–38, 2004.

MSC99. A. K. D. Moran, D. Sands, and M. Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, volume 1594 of *LNCS*, pages 85–102. Springer-Verlag, 1999.

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

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

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

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

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

SSM08. M. Schmidt-Schauß and E. Machkasova. A finite simulation method in a non-deterministic call-by-need calculus with letrec, constructors and case. Frank 32, Inst. f. Informatik, J.W.Goethe-University, Frankfurt, 2008.

SSS07a. D. Sabel and M. Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Math. Structures Comput. Sci.*, 2007. accepted for publication.

SSS07b. M. Schmidt-Schauß and D. Sabel. On generic context lemmas for lambda calculi with sharing. Frank 27, Inst. Informatik, J.W.G-Univ., Frankfurt, 2007.

SSSS04. M. Schmidt-Schauß, M. Schütz, and D. Sabel. On the safety of Nöcker's strictness analysis. Frank 19, Inst. Informatik, J.W.G-Univ., Frankfurt, 2004.

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

WDK03. J. B. Wells, D.Plump, and F. Kamareddine. Diagrams for meaning preservation. In *RTA*, volume 2706 of *LNCS*, pages 88–106, 2003.