A Complete Proof of Nöcker's Strictness Analysis*

Manfred Schmidt-Schauß[1], Marko Schütz[2], and David Sabel[1]

[1] Institut für Informatik, Johann Wolfgang Goethe-Universität,
Postfach 11 19 32, D-60054 Frankfurt, Germany,
schauss@ki.informatik.uni-frankfurt.de
[2] Dept. of Mathematics and Computing Science,
University of the South Pacific, Suva, Fiji Islands

* Under consideration for publication in a journal.

Technical Report Frank-20
Research group for Artificial Intelligence and Software Technology,
Institut für Informatik,
J.W.Goethe-Universität Frankfurt,

07. Apr. 2005

Abstract. This paper proves correctness of Nöcker's method of strictness analysis, implemented in the Clean compiler, which is an effective way for strictness analysis in lazy functional languages based on their operational semantics. We improve upon the work of Clark, Hankin and Hunt did on the correctness of the abstract reduction rules. Our method fully considers the cycle detection rules, which are the main strength of Nöcker's strictness analysis.
Our algorithm SAL is a reformulation of Nöcker's strictness analysis algorithm in a higher-order call-by-need lambda-calculus with `case`, constructors, `letrec`, and `seq`, extended by set constants like `Top` or Inf, denoting sets of expressions. It is also possible to define new set constants by recursive equations with a greatest fixpoint semantics. The operational semantics is a small-step semantics. Equality of expressions is defined by a contextual semantics that observes termination of expressions. Basically, SAL is a non-termination checker. The proof of its correctness and hence of Nöcker's strictness analysis is based mainly on an exact analysis of the lengths of normal order reduction sequences. The main measure being the number of "essential" reductions in a normal order reduction sequence.
Our tools and results provide new insights into call-by-need lambda-calculi, the role of sharing in functional programming languages, and into strictness analysis in general. The correctness result provides a foundation for Nöcker's strictness analysis in Clean, and also for its use in Haskell.

Table of Contents

1 Introduction

Strictness analysis is an essential phase when compiling programs in lazy functional languages such as Haskell [Jon03] and Clean [PvE03]. Conservative parallel evaluation and many optimizations become possible only with the information gained in strictness analysis. There are different methods: e.g. ad-hoc strictness optimizations in compilation schemes, strictness analysis based on abstract interpretation, use of type systems, and strictness analysis based on operational semantics.

A very effective way for strictness analysis in functional languages are algorithms based on the operational semantics. Nöcker's strictness analysis for Clean (see [NÖ0,Nöc93]) is a prominent example. In their paper [CHH00] Clark, Hankin and

Hunt show correctness of the part of the algorithm that pushes the abstract values through the program using the operational semantics. However, this is only part of the correctness. The cycle-detection rules are not considered. But these are the very rules that account for much of the strength of Nöcker's algorithm.
This paper extends the ideas on reduction length of normal order reductions for a proof of correctness of a strictness analysis algorithm in [SSPS95]. Essentially, this paper is a modified version of the proof in [SSSS04], which used a non-deterministic lambda-calculus, where non-determinism was exploited for representing sets of expressions. The report [SSSS04] had to use the conjecture that simulation implies contextual equivalence in the non-deterministic calculus, which is not necessary for the proof presented in this paper.
We will reformulate Nöcker's strictness analysis algorithm in a higher-order call-by-need lambda-calculus with `case`, constructors, `letrec`, constructors, and `seq`, extended by abstract constants representing sets of expressions. Graph reduction is modeled by `letrec`, which can also describe recursive definitions and can explicitly treat the sharing inherent in lazy functional languages.
A large part of the proof is to exhibit the properties of the reduction rules and extra reductions, in particular their influence on the length of normal order reduction sequences of some expression, where the main length measure only takes essential steps into account. The call-by-need calculus using `letrec` has a rather well-behaved length-measure for reductions, the reason appears to be the exact treatment of sharing: Moreover, the length-measure is robust w.r.t. changing the order of reduction, e.g. using strictness of expressions, and also invariant w.r.t. simplification rules and rules that only rearrange the let-structure of expressions. This robustness and the nice behavior of the variants of reduction length are consequences of the exact treatment of sharing in the calculus. The final induction in the correctness proof will be on the "essential" length of normal order reduction sequences.
Let us consider two example applications of the analysis algorithm.
An expression `f` is called *strict* in argument i for arity n, iff the evaluation starting with `f` $t_1 \ldots t_{i-1} \perp t_{i+1} \ldots, t_n$ will never yield a weak head normal form, where $\perp$ represents terms without WHNF.
The first example is the combinator K with definition `K x y = x`, which is strict in its first argument (for arity 2). This will be detected by Nöcker's method as follows. With $\top$ representing every closed term, `K` $\perp$ $\top$ reduces to $\perp$ indicating that `K` is indeed strict in its first argument.
A nontrivial example (see also 2) is `length` with the following definition:

```
length  =  letrec len  = \ lst a ->  case lst of
                          Nil  -> a;
                          y:ys -> len  ys (a+1)
           in len
```

Reducing (`length` $\top$ $\perp$) using the rules of the calculus results either in $\perp$ or in an expression that is essentially the same as `length` $\top$ $\perp$. Since the same

expression is generated, and at least one (essential) normal order reduction was necessary, the strictness analysis algorithm concludes that the expression loops. Summarizing, the answer will be that `length` is strict in its second argument.
Our proof justifies this reasoning by loop-detection, even in connection with abstract set constants like $\top$ or Inf. However, our syntax is slightly different from Nöcker's since in our syntax there is a global `letrec` environment including all relevant function definitions; in addition our syntax can also express equality of set constants by sharing (see Remark 4).
A description of the structure of the paper is in section 3.

2 Related Work

Strictness analysis has been approached from many different perspectives. These can roughly be characterized as based on abstract interpretation (e.g. [BHA85,AH87,Bur91,CC77,Myc81,Wad87]), projections (e.g. [WH87,Pat96,LPJ95]), non-standard type systems (e.g. [KM89,Jen98,GNN98,CDG02]) or abstract reduction [Nöc92]. We will be concerned with the latter and will only briefly comment on the other approaches. For a detailed comparison of many of these approaches we refer to [Pap98,Pap00].
In [Nöc92,Nöc93] Nöcker described a strictness analysis based on abstracting the operational semantics of a non-strict functional programming language. This strictness analysis is very appealing, both intuitively and pragmatically, but it has proven theoretically challenging.
The key concept is to add new names for abstract constants, such as $\bot$ for all terms without WHNF, or $\top$ for all expressions, and to add appropriate (abstract) reduction rules capturing their semantics. This analysis was implemented at least twice: once by Nöcker in C for Concurrent Clean [NSvP91] and once by Schütz in Haskell [Sch94]. As of Concurrent Clean version 2.1 Nöcker's C-implementation is still in use in the compiler. The analysis is not very expensive to implement, runs quickly without large memory requirements and obtains good results.
Its drawback seems to be the slow progress in its theoretical foundation. Nöcker [Nöc92] himself proved correctness of the analysis for orthogonal term rewriting systems only. In [SSPS95] we showed correctness of the analysis for a supercombinator-based functional core language. In that exposition a treatment of sharing and `letrec` was missing. Then Clark, Hankin and Hunt [CHH00] proved correctness of a significant subset of the analysis, but did not consider the loop-detection rules. Since the loop-detection rules may well be the most important aspect of strictness analysis by abstract reduction this paper provides a formal account of the analysis using a language with explicit sharing and proving correctness for all of the rules of Nöcker's algorithm.
Moran and Sands in [MS99] developed tools for the detailed analysis of reduction lengths, unfortunately these cannot be used here, since only certain essential normal order reductions are relevant and also counting the number of `letrec`-

shufflings is not appropriate for the proof of correctness of Nöcker's strictness analysis.
Strictness analysis has numerous applications: optimizing the compilation of expressions, detecting possibilities for conservative parallel evaluation, and checking preconditions for correct application of transformations in the compilation process of lazy functional programming languages (see [San95,PJS94]).
This paper contributes to increase the applicability and trustworthiness of Nöcker's method and to provide foundations for its application e.g. in Haskell.

3 Overview

The goal of this paper is a reformulation and the proof of correctness of Nöcker-type strictness analysis for non-strict functional programming languages. Since the actions of the algorithm rely on the small-step operational semantics of a functional core language, LR, we use the operational semantics and an equality based on contextual preorder. An appropriate tool for proving the correctness of the cycle detection rules of Nöcker's algorithm is the "essential" length of a normal order reduction. The domains commonly used in the literature on denotational semantics do not provide such an operational measure. We could have developed appropriate tools based on a non-standard denotational semantics, but felt the operational approach to be more intuitive.
The paper has three main parts, where almost all proofs in the first two parts are shifted into the appendix:

1. A description of the language and the normal order reduction (sections 4 – 6).
2. A detailed analysis of the properties of contextual equivalence and of the length of normal order reduction sequences (sections 7 – 11).
3. A description of the strictness analysis algorithm, its data structures and a proof of its correctness (sections 12 – 15).

The first part is concerned with describing the calculus for a call-by-need functional core language LR using sharing and with investigating equivalences and variants of lengths of normal order reduction sequences. Set constants like $\top$ or *Inf* are permitted in the extended core language LR_U.
The core language provides `letrec`, the usual primitives like a weakly typed `case`, constructors, lambda, application, and `seq`. The latter is included, since programs in Clean or Haskell often use an equivalent primitive which would otherwise not be representable in the core language. The core language and its analysis is borrowed mainly from [SS03]. The `case`-primitive is slightly changed insofar as it is weakly typed. It has to be complemented by the addition of a `seq` in order to have the same expressiveness. The typing makes the language more similar in behavior to a typed functional programming language (see Example 7).

Contrary to Moran, Sands and Carlsson [MSC99] in applications we allow arguments other than variables. The results in this paper show this restriction on the term structure to be irrelevant for the main length measure.

The reduction rules for the language LR are defined for any matching subexpression. The normal order reduction is then defined as a specific strategy uniquely determining the next sub-expression for reduction. We define contextual equivalence as usual where the only observation is successful termination of the evaluation of an expression.

In the second part we show that contextual equivalence is stable w.r.t. all reduction rules. In this we employ a context lemma and the computing of overlappings of rules leading to complete sets of commuting and forking diagrams. Our main tool is induction. The measure is essentially the length of normal order reduction sequences. For technical reasons we need to provide several measures each counting a specific set of reduction rules occurring in the normal order reduction sequence. We then study how these measures are affected by reduction steps. A further base is a theorem on the correctness of copying parts of concrete terms.

In the third part we define the algorithm SAL as a reformulation of Nöcker's algorithm. It uses previously computed strictness knowledge about functions and strictness of built-in functions. The main data structure is a directed graph of abstract expressions, where the directed edges correspond to reductions or to cycle-checks. An expression in the graph represents a set of terms in the concrete core language, and a reduction either modifies the abstract expression or makes a case distinction on a set constant. Set constants may be $\top$, the set of all closed expressions, or *Inf*, the set of all expressions evaluating to infinite lists or lists without tails. It is also possible to define new set constants by recursive equations. This leads to a concise representation of unions, and it indicates that a directed graph is more appropriate to check the termination conditions.

The conditions on successful termination of SAL give new insights into the nature of the algorithm. In fact SAL is a non-termination checker for an infinite set of concretizations described by an abstract expression. The proof justifies the intuition that certain reductions (normal order and reductions at strict position) make progress, whereas this is not true for several other reductions.

The correctness proof of SAL (Theorem 8 and Corollaries 3,4) relies on arguments on the number of "essential" normal order reduction steps of expressions after reductions and transformations.

4 Syntax of the Functional Core Language LR

Our language, LR, the language of concrete terms, has the following syntax:
There are finitely many constants, called constructors. The set of constructors is partitioned into (nonempty) types. For every type T we denote the constructors as $c_{T,i}, i = 1, \ldots, |T|$. Every constructor has an arity $\mathrm{ar}(c_{T,i}) \geq 0$.

The syntax for expressions E, case alternatives Alt and patterns Pat is as follows:

$$\begin{aligned} E &::= V \mid (c\ E_1 \ldots E_{\mathrm{ar}(c)}) \mid (\texttt{seq}\ E_1\ E_2) \mid (\texttt{case}_T\ E\ Alt_1 \ldots Alt_{|T|}) \mid (E_1\ E_2) \\ &\quad (\lambda\ V.E) \mid (\texttt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \texttt{in}\ E) \\ Alt &::= (Pat\ \rightarrow\ E) \\ Pat &::= (c\ V_1 \ldots V_{\mathrm{ar}(c)}) \end{aligned}$$

where E, E_i are expressions, V, V_i are variables and where c denotes a constructor. Within each individual pattern variables are not repeated. In a `case`-expression of the form $(\texttt{case}_T \ldots)$, for every constructor $c_{T,i}, i = 1, \ldots, |T|$ of type T, there is exactly one alternative with a pattern of the form $(c_{T,i}\ y_1 \ldots y_n)$.
We assign the names *constructor application*, `seq`-*expression*, `case`-*expression*, *application*, *abstraction*, or `letrec`-*expression* to the expressions $(c\ E_1 \ldots E_{\mathrm{ar}(c)})$, $(\texttt{seq}\ E_1\ E_2)$, $(\texttt{case}_T\ E\ Alt_1 \ldots Alt_N)$, $(E_1\ E_2)$, $(\lambda V.E)$, $(\texttt{letrec}\ V_1 = E_1, \ldots V_n = E_n\ \texttt{in}\ E)$, respectively.
The constructs `case`, `seq` and the constructors $c_{T,i}$ can only occur in special syntactic constructions. Thus expressions where `case`, `seq` or a constructor is applied to the wrong number of arguments are not allowed.
The structure `letrec` obeys the following conditions: The variables V_i in the bindings are all distinct. We also assume that the bindings in `letrec` are commutative, i.e. `letrec`s with interchanged bindings are assumed to be syntactically equivalent. `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. This fixes the notions of closed, open expressions and α-renamings. Free and bound variables in expressions are defined using the usual conventions. Variable binding primitives are λ, `letrec`, patterns, and the scope of variables bound in a `letrec` are all the expressions occurring in it. The set of free variables in an expression 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 to obey this convention. Note that this is only necessary for the copy rule (cp). We follow the convention by omitting parentheses in nested applications: $(s_1 \ldots s_n)$ denotes $(\ldots (s_1\ s_2) \ldots s_n)$.
The set of closed LR-expressions is denoted as L^0.
To abbreviate the notation, we will sometimes use $(\texttt{case}_T\ E\ alts)$ instead of $(\texttt{case}_T\ E\ alt_1 \ldots alt_{|T|})$. Sometimes we abbreviate the notation of `letrec`-expression $(\texttt{letrec}\ x_1 = E_1, \ldots x_n = E_n\ \texttt{in}\ E)$, as $(\texttt{letrec}\ Env\ \texttt{in}\ E)$, where $Env \equiv \{x_1 = E_1, \ldots x_n = E_n\}$. This will also be used freely for parts of the bindings. We also use the notation $\{x_{g(i)} = s_{h(i)}\}_{i=m}^{n}$ for the chain $x_{g(m)} = s_{h(m)}, x_{g(m+1)} = s_{h(m+1)}, \ldots x_{g(n)} = s_{h(n)}$ of bindings, e.g., $\{x_i = s_{i-1}\}_{i=m}^{n}$ means the bindings $x_m = s_{m-1}, x_{m+1} = s_m, \ldots x_n = s_{n-1}$. We assume that `letrec`-expressions have at least one binding. The set of bound variables in an environment Env is denoted as $LV(Env)$. In examples we will use : as an infix binary list-constructor, and `Nil` as the constant constructor for lists. We will write $(c_i\ \overrightarrow{z})$ as shorthand for the constructor application $(c_i\ z_1\ \ldots\ z_{\mathrm{ar}(c_i)})$.

In the following we define different context classes and contexts. To visually distinguish context classes from individual contexts, we use different text styles.

Definition 1. *The class $\mathcal{C}$ of all* contexts *is defined as follows.*

$$\begin{aligned}\mathcal{C} ::= {} & [\cdot] \mid (\mathcal{C}\ E) \mid (E\ \mathcal{C}) \mid\ (\texttt{seq}\ E\ \mathcal{C}) \mid (\texttt{seq}\ \mathcal{C}\ E) \mid \lambda x.\mathcal{C} \\ & \mid (\texttt{case}_T\ \mathcal{C}\ alts) \mid (\texttt{case}_T\ E\ alt_1, \ldots, (Pat\ \rightarrow\ \mathcal{C}), \ldots, alt_n) \\ & \mid (c\ E_1 \ldots E_{i-1}\ \mathcal{C}\ E_{i+1} \ldots E_{\mathrm{ar}(c)}) \\ & \mid (\texttt{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ \mathcal{C}) \\ & \mid (\texttt{letrec}\ \{x_j = E_j\}_{j=1}^{i-1}, x_i = \mathcal{C}, \{x_j = E_j\}_{j=i+1}^{n}\ \texttt{in}\ E)\end{aligned}$$

Definition 2. *The following special context classes are defined:* reduction contexts*, $\mathcal{R}$, and* weak reduction contexts*, $\mathcal{R}^-$, the latter has no* `letrec`*-expressions above the hole. The former achieves nesting by referencing bound variables from inside weak reduction contexts.*

$$\begin{aligned}\mathcal{R}^- ::= {} & [\cdot] \mid (\mathcal{R}^-\ E) \mid (\texttt{case}_T\ \mathcal{R}^-\ alts) \mid (\texttt{seq}\ \mathcal{R}^-\ E) \\ \mathcal{R}\ ::= {} & \mathcal{R}^- \mid (\texttt{letrec}\ Env\ \texttt{in}\ \mathcal{R}^-) \mid \\ & (\texttt{letrec}\ x_1 = \mathcal{R}_1^-, x_2 = \mathcal{R}_2^-[x_1], \ldots, x_j = \mathcal{R}_j^-[x_{j-1}], Env\ \texttt{in}\ \mathcal{R}^-[x_j]) \\ & \quad \textit{where } j \geq 1 \textit{ and } \mathcal{R}^-, \mathcal{R}_i^-, i = 1, \ldots, j \textit{ are weak reduction contexts}\end{aligned}$$

For a term t with $t \equiv R^-[t_0]$, we say R^- is maximal (for t)*, iff there is no larger weak reduction context with this property. For a term t with $t \equiv C[t_0]$, we say C is a* maximal reduction context *iff C is either*

- *a maximal weak reduction context, or*
- *of the form $(\texttt{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ R^-)$ where R^- is a maximal weak reduction context and $t_0 \neq x_j$ for all $j = 1, \ldots, n$, or*
- *of the form $(\texttt{letrec}\ x_1 = R_1^-, x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], \ldots\ \texttt{in}\ R^-[x_j])$, where $R_i^-, i = 1, \ldots, j$ are weak reduction contexts and R_1^- is a maximal weak reduction context for $R_1^-[t_0]$, and the number, j, of involved bindings is maximal. (Other bindings may, of course, be present.)*

Searching for a maximal reduction context can be seen as an algorithm walking over the term structure. In implementations of functional programming this is usually called "unwind" (see also section 5).
For example the maximal reduction context of $(\texttt{letrec}\ x_2 = \lambda x.x, x_1 = x_2\ x_1\ \texttt{in}\ x_1)$ is $(\texttt{letrec}\ x_2 = [\cdot], x_1 = x_2\ x_1\ \texttt{in}\ x_1)$, in contrast to the non-maximal reduction context $(\texttt{letrec}\ x_2 = \lambda x.x, x_1 = x_2\ x_1\ \texttt{in}\ [\cdot])$.

Definition 3. *We define* surface contexts*, meaning that the hole is not in the body of an abstraction. Let $\mathcal{S}$ be the context class of* surface contexts *defined as follows:*

$$\begin{aligned}\mathcal{S} ::= {} & [\cdot] \mid (\mathcal{S}\ E) \mid (E\ \mathcal{S}) \mid\ (\texttt{seq}\ E\ \mathcal{S}) \mid (\texttt{seq}\ \mathcal{S}\ E) \\ & \mid (\texttt{case}_T\ \mathcal{S}\ alts) \mid (\texttt{case}_T\ E\ alt_1, \ldots, (Pat\ \rightarrow\ \mathcal{S}), \ldots, alt_n) \\ & \mid (c\ E_1 \ldots E_{i-1}\ \mathcal{S}\ E_{i+1} \ldots E_{\mathrm{ar}(c)}) \\ & \mid (\texttt{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ \mathcal{S}) \\ & \mid (\texttt{letrec}\ \{x_j = E_j\}_{j=1}^{i-1}, x_i = \mathcal{S}, \{x_j = E_j\}_{j=i+1}^{n}\ \texttt{in}\ E)\end{aligned}$$

Note that every reduction context is also a surface context.

Definition 4. *We define* application surface contexts*, meaning that the hole is not in the body of an abstraction and not in the alternatives of a case. Let $\mathcal{AS}$ be the context class of* application surface contexts *defined as follows:*

$$\begin{aligned}\mathcal{AS} ::= \ & [\cdot] \mid (\mathcal{AS}\ E) \mid (E\ \mathcal{AS}) \mid \ (\texttt{seq}\ E\ \mathcal{AS}) \mid (\texttt{seq}\ \mathcal{AS}\ E) \\ & \mid (\texttt{case}_T\ \mathcal{AS}\ alts) \mid (c\ E_1 \ldots E_{i-1}\ \mathcal{AS}\ E_{i+1} \ldots E_{\mathrm{ar}(c)}) \\ & \mid (\texttt{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ \mathcal{AS}) \\ & \mid (\texttt{letrec}\ \{x_j = E_j\}_{j=1}^{i-1}, x_i = \mathcal{AS}, \{x_j = E_j\}_{j=i+1}^{n}\ \texttt{in}\ E)\end{aligned}$$

Definition 5. *Let D be a context, the* main depth *of D is the depth of the hole in D. With $D_{(i)}$ we denote a context of main depth i.*

Definition 6. *Let $\mathcal{AS}^{-}_{(1)}$ be the context class of* weak application surface contexts of main depth 1*, which are application surface contexts with a hole not below a* `letrec`*:*

$$\begin{aligned}\mathcal{AS}^{-}_{(1)} ::= \ & ([\cdot]\ E) \mid (E\ [\cdot]) \mid (c\ E_1 \ldots E_{i-1}\ [\cdot]\ E_{i+1} \ldots E_{\mathrm{ar}(c)}) \\ & \mid (\texttt{case}_T\ [\cdot]\ alts) \mid (\texttt{seq}\ [\cdot]\ E) \mid (\texttt{seq}\ E\ [\cdot])\end{aligned}$$

Sometimes we will also use *multicontexts*, which are like contexts, but have several holes $\cdot_i$, and every hole occurs exactly once in the term. We write a multicontext as $C[\cdot_1, \ldots, \cdot_n]$, and if the terms s_i for $i = 1, \ldots, n$ are plugged into the holes $\cdot_i$, then we denote the resulting term as $C[s_1, \ldots, s_n]$.

Definition 7. *A* value *is either an abstraction, or a constructor application. We denote values by the letters v, w.*

Definition 8. *The (base) reduction rules for the language* LR *are defined in figures 1 and 2. The union of (llet-in) and (llet-e) is called (llet), the union of (case-c), (case-in), (case-e) is called (case), the union of (seq-c), (seq-in), (seq-e) is called (seq), the union of (cp-in) and (cp-e) is called (cp), and the union of (llet), (lcase), (lapp) (lseq), is called (lll).*
The specializations of (seq), (case), (cp) where the C-context is restricted to a surface context, is denoted as (seqS), (caseS), (cpS).
Reductions are denoted using an arrow with super and/or subscripts: e.g. $\xrightarrow{llet}$. To explicitly state the context in which a particular reduction is executed we annotate the reduction arrow with the context in which the reduction takes place. If no confusion arises, we omit the context at the arrow.
The redex *of a reduction is the term as given on the left side of a reduction rule. We will also speak of the* inner redex*, which is the modified* `case`*-expression for (case)-reductions, the modified* `seq`*-expression for (seq)-reductions, and the variable position which is replaced by a (cp). Otherwise it is the same as the redex.*
Transitive closure of reductions is denoted by a $+$, reflexive transitive closure by a $*$. E.g. $\xrightarrow{*}$ is the reflexive, transitive closure of $\rightarrow$. If necessary, we attach more information to the arrow.

(lbeta)	$((\lambda x.s)\ r) \to (\texttt{letrec}\ x = r\ \texttt{in}\ s)$
(cp-in)	$(\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \ \texttt{in}\ C[x_m]) \to (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \ \texttt{in}\ C[v])$ where v is an abstraction
(cp-e)	$(\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[x_m]\ \texttt{in}\ r) \to (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[v]\ \texttt{in}\ r)$ where v is an abstraction
(llet-in)	$(\texttt{letrec}\ \{x_i = s_i\}_{i=1}^{n}\ \texttt{in}\ (\texttt{letrec}\ \{y_i = t_i\}_{i=1}^{m}\ \texttt{in}\ r)) \to (\texttt{letrec}\ \{x_i = s_i\}_{i=1}^{n}, \{y_i = t_i\}_{i=1}^{m}\ \texttt{in}\ r)$
(llet-e)	$(\texttt{letrec}\ x_1 = s_1, \ldots, x_i = (\texttt{letrec}\ \{y_i = t_i\}_{i=1}^{m}\ \texttt{in}\ s_i), \ldots, x_n = s_n\ \texttt{in}\ r) \to (\texttt{letrec}\ \{x_i = s_i\}_{i=1}^{n}, \{y_i = t_i\}_{i=1}^{m}\ \texttt{in}\ r)$
(lapp)	$((\texttt{letrec}\ Env\ \texttt{in}\ t)\ x) \to (\texttt{letrec}\ Env\ \texttt{in}\ (t\ x))$
(lcase)	$(\texttt{case}_T\ (\texttt{letrec}\ Env\ \texttt{in}\ t)\ alts) \to (\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{case}_T\ t\ alts))$
(seq-c)	$(\texttt{seq}\ v\ t) \to t$ if v is a value
(seq-in)	$(\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[(\texttt{seq}\ x_m\ t)]) \to (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[t])$ if v is a value
(seq-e)	$(\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[(\texttt{seq}\ x_m\ t)]\ \texttt{in}\ r) \to (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[t]\ \texttt{in}\ r)$ if v is a value
(lseq)	$(\texttt{seq}\ (\texttt{letrec}\ Env\ \texttt{in}\ s)\ t) \to (\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{seq}\ s\ t))$

Fig. 1. Reduction rules, part a

Note that the reduction rules generate only syntactically correct expressions, since contexts are appropriately defined.

Remark 1. The case-rule looks a bit weird, but it is carefully designed and we made all possibilities explicit. If a `case`-expression of the form `case_T x ...` is to be evaluated, then the case and constructor application must cooperate. It is not permitted to copy every constructor application into the position of x. A possibility is to use a rule (abs) that abstracts the terms in the constructor application. However, including the rule (abs) into the calculus provides a very hard obstacle in proving that all reductions are correct program transformations. The current definition of (case) is borrowed from FUNDIO [SS03].

5 Normal Order Reduction

First we will informally describe how the position of the normal order redex can be reached by using a labeling algorithm. Then we will rigidly define the normal order reduction in definition 9.
The following labeling algorithm will detect the position to which a reduction rule will be applied according to normal order. It uses three labels: $e0, e1, e$, where $e0$ means evaluation of the top term, $e1$ means evaluation of a subterm, and e matches $e0$ as well as $e1$. The algorithm does not look into $e1$ labeled letrec-expressions. For a term s the labeling algorithm starts with s^{e0}

(case-c) $(\texttt{case}_T\ (c_i\ \overrightarrow{t})\ \ldots((c_i\ \overrightarrow{y}) \to t)\ldots) \to (\texttt{letrec}\ \{y_i = t_i\}_{i=1}^n\ \texttt{in}\ t)$
where $n = \mathrm{ar}(c_i) \geq 1$

(case-c) $(\texttt{case}_T\ c_i\ \ldots\ (c_i \to t)\ldots) \to t$ if $\mathrm{ar}(c_i) = 0$

(case-in) $\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^m, Env$
$\texttt{in}\ C[\texttt{case}_T\ x_m\ \ldots((c_i\ \overrightarrow{z})\ldots \to t)\ldots]$
$\to \texttt{letrec}\ x_1 = (c_i\ \overrightarrow{y}), \{y_i = t_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=2}^m, Env$
$\texttt{in}\ C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^n\ \texttt{in}\ t)]$
where $n = \mathrm{ar}(c_i) \geq 1$ and y_i are fresh variables

(case-in) $\texttt{letrec}\ x_1 = c_i, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ C[\texttt{case}_T\ x_m\ \ldots\ (c_i \to t)\ldots]$
$\to \texttt{letrec}\ x_1 = c_i, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ C[t]$
if $\mathrm{ar}(c_i) = 0$

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

(case-e) $\texttt{letrec}\ x_1 = c_i, \{x_i = x_{i-1}\}_{i=2}^m, u = C[\texttt{case}_T\ x_m\ \ldots\ (c_i \to r_1)\ldots], Env$
$\texttt{in}\ r_2$
$\to \texttt{letrec}\ x_1 = c_i, \{x_i = x_{i-1}\}_{i=2}^m \ldots, u = C[r_1], Env\ \texttt{in}\ r_2$
if $\mathrm{ar}(c_i) = 0$

Fig. 2. Reduction rules, part b

The labeling algorithm:

$$
\begin{array}{ll}
(\texttt{letrec}\ Env\ \texttt{in}\ t)^{e0} & \to (\texttt{letrec}\ Env\ \texttt{in}\ t^{e1}) \\
(s\ t)^{e} & \to (s^{e1}\ t) \\
(\texttt{seq}\ s\ t)^{e} & \to (\texttt{seq}\ s^{e1}\ t) \\
(\texttt{case}_T\ s\ alts)^{e} & \to (\texttt{case}_T\ s^{e1}\ alts) \\
(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C[x^{e1}]) & \to (\texttt{letrec}\ x = s^{e1}, Env\ \texttt{in}\ C[x]) \\
(\texttt{letrec}\ x = s, y = C[x^{e1}], Env\ \texttt{in}\ t) & \to (\texttt{letrec}\ x = s^{e1}, y = C[x], Env\ \texttt{in}\ t)
\end{array}
$$

If the labeling algorithm terminates, i.e. it is no longer possible to apply a rule, then the normal order redex may only be the marked subterm or its direct superterm. It is possible that there is no normal order reduction: In this case either the evaluation is already finished, or it can be viewed as a kind of dynamically detected error. If the labeling algorithm does not terminate (e.g. due to mutually recursive `letrec`-bindings), then there is no normal order redex and hence no normal order reduction.

Definition 9. *Let t be an expression. Let R be the maximal reduction context such that $t \equiv R[t']$ for some t'. The normal order reduction $\xrightarrow{n}$ is defined by one of the following cases:*

1. *t' is a* `letrec`*-expression* ($\texttt{letrec}\ Env_1\ \texttt{in}\ t''$)*, and R is not trivial. Then there are 5 cases, where R_0 is a reduction context:*
   (a) $R = R_0[(\texttt{seq}\ [\cdot]\ r)]$. *Reduce* $(\texttt{seq}\ t'\ r)$ *using (lseq).*
   (b) $R = R_0[([\cdot]\ x)]$. *Reduce* $(t'\ x)$ *using (lapp).*
   (c) $R = R_0[(\texttt{case}_T\ [\cdot]\ alts)]$. *Reduce* $(\texttt{case}_T\ t'\ alts)$ *using (lcase).*
   (d) $R = (\texttt{letrec}\ Env_2\ \texttt{in}\ [\cdot])$. *Reduce t using (llet-in) by flattening t' resulting in* $(\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ t'')$.
   (e) $R = (\texttt{letrec}\ x = [\cdot], Env_2\ \texttt{in}\ t''')$. *Reduce t using (llet-e) by flattening t' resulting in* $(\texttt{letrec}\ x = t'', Env_1, Env_2\ \texttt{in}\ t''')$.
2. *t' is a value. There are the following cases:*
   (a) $R = R_0[\texttt{case}_T\ [\cdot]\ \ldots]$, $t' \equiv (c_T\ \ldots)$, *i.e. the top constructor of t' belongs to type T. Then apply (case-c) to* $(\texttt{case}_T\ t'\ \ldots)$,
   (b)
   $$R = \texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^{m}, Env \\ \texttt{in}\ R_0^-[\texttt{case}_T\ x_m\ (c_{T,j}\ \overrightarrow{y} \to r)\ alts],$$
   $t' = c_{T,j}\ \overrightarrow{t}$, *Then apply (case-in) resulting in*
   $$\texttt{letrec}\ x_1 = c_{T,j}\ \overrightarrow{z}, \{x_i = x_{i-1}\}_{i=2}^{m}, \{z_i = t_i\}_{i=1}^{n}, Env \\ \texttt{in}\ R_0^-[(\texttt{letrec}\ \{y_i = z_i\}_{i=1}^{n}\ \texttt{in}\ r)]$$
   (c) $R = \texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ R_0^-[\texttt{case}_T\ x_m\ (c_{T,j} \to r)\ alts]$, $t' = c_{T,j}$. *Then apply (case-in) resulting in* $\texttt{letrec}\ x_1 = c_{T,j}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ R_0^-[r]$
   (d) $R = \texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^{m}, Env,\ y = R_0^-[\texttt{case}_T\ x_m\ (c_{T,j}\ \overrightarrow{y} \to r)\ alts]\ \texttt{in}\ r'$,
   $t' = c_{T,j}\ \overrightarrow{t}$, *and y is in a reduction context. Then apply (case-e) resulting in*
   $$\texttt{letrec}\ x_1 = c_{T,j}\ \overrightarrow{z}, \{x_i = x_{i-1}\}_{i=2}^{m}, \{z_i = t_i\}_{i=1}^{n}, Env, \\ y = R_0^-[(\texttt{letrec}\ \{y_i = z_i\}_{i=1}^{n}\ \texttt{in}\ r)] \\ \texttt{in}\ r'$$
   (e) $R = \texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^{m}, Env,\ y = R_0^-[\texttt{case}_T\ x_m\ (c_{T,j} \to r)\ alts]\ \texttt{in}\ r'$,
   $t' = c_{T,j}$, *and y is in a reduction context.*
   Then apply (case-e) resulting in
   $\texttt{letrec}\ x_1 = c_{T,j}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = R_0^-[r]\ \texttt{in}\ r'$.
   (f) $R = R_0[([\cdot]\ s)]$ *where R_0 is a reduction context and t' is an abstraction. Then apply (lbeta) to* $(t'\ s)$.
   (g) $R = (\texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ R_0^-[x_m])$ *where R_0^- is a weak reduction context and t' is an abstraction. Then apply (cp-in) and copy t' to the indicated position, resulting in* $(\texttt{letrec}\ x_1 = [t'], \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ R_0^-[t'])$.

(h) $R = (\texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^m, Env, y = R_0^-[x_m]\ \texttt{in}\ r)$ *where* R_0^- *is a weak reduction context,* y *is in a reduction context and* t' *is an abstraction. Then apply (cp-e) resulting in* $(\texttt{letrec}\ x_1 = [t'], \{x_i = x_{i-1}\}_{i=2}^m, Env, y = R_0^-[t']\ \texttt{in}\ r)$.
(i) $R = R_0[(\texttt{seq}\ [\cdot]\ r)]$. *Then apply (seq-c) to* $(\texttt{seq}\ t'\ r)$ *resulting in* r.
(j) $R = (\texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ R_0^-[(\texttt{seq}\ x_m\ r)])$, *and* t' *is a constructor application. Then apply (seq-in) resulting in* $(\texttt{letrec}\ x_1 = t', \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ R_0^-[r])$.
(k) $R = (\texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^m, Env, y = R_0^-[(\texttt{seq}\ x_m\ r)]\ \texttt{in}\ r')$ *where* y *is in a reduction context, and* t' *is a constructor application. Then apply (seq-e) resulting in* $(\texttt{letrec}\ x_1 = t', \{x_i = x_{i-1}\}_{i=2}^m, Env, y = R_0^-[r]\ \texttt{in}\ r')$.

The normal order redex *is defined as the subexpression to which the reduction rule is applied. This includes the* `letrec`*-expression that is mentioned in the reduction rules, for example in (cp-e).*

The normal order reduction implies that `seq` behaves like a function strict in its first argument, and that the `case`-construct is strict in its first argument. I.e., these rules can only be applied if the corresponding argument is a value.
A central notion is that of weak head normal form.

Definition 10. *A* weak head normal form (WHNF) *is one of the cases:*

- *A value* v.
- *A term of the form* $(\texttt{letrec}\ Env\ \texttt{in}\ v)$, *where* v *is a value.*
- *A term of the form* $(\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ x_m)$

If the value v *is an abstraction, we call it a* functional WHNF (FWHNF).

Lemma 1. *For every term* t*: if* t *has a normal order redex, then the redex and the normal order reduction are unique.*

Definition 11. *For a term* t*, we write* $t{\Downarrow}$ *iff there is a normal order reduction sequence to WHNF starting from* t*. Otherwise, we write* $t{\Uparrow}$*. If* $t{\Downarrow}$*, we say that* t *is* terminating.
If an expression t *is terminating, the normal order reduction to WHNF is denoted as* $nor(t)$.
For a term t*, we write* $t{\Uparrow\Uparrow}$*, if* t *has no normal order reduction to a WHNF, and no normal order reduction to a term of the form* $R[x]$ *where* x *is a free variable in* $R[x]$*, and* R *is a reduction context. A term* t *with* $t{\Uparrow\Uparrow}$ *is also called* bot-term*, and a specific representative is* Ω*, which can be defined as*

$$\Omega := (\lambda z.(z\ z))\ (\lambda x.(x\ x)).$$

Note that there are useful open terms t that might not have any normal order reduction to a WHNF, e.g. x is such a term.
Note also that there are (closed) terms t that are neither WHNFs nor have a normal order redex. For example $(\texttt{case}_T(\lambda x.x)\ alts)$ or `((cons 1 2) 3)`, where

`cons` is a constructor of arity 2. These terms are bot-terms and could be considered as violating type conditions.
Consider the closed "cyclic term" ($\texttt{letrec}\ x = x\ \texttt{in}\ x$). The maximal reduction context for this term is ($\texttt{letrec}\ x = [\cdot]\ \texttt{in}\ x$). Obviously, there is no normal order reduction defined for this term.
A term that has a non-terminating normal order reduction is $(\lambda z.(z\ \ z))\ \ (\lambda x.(x\ \ x))$ the start of the infinite reduction being $(\lambda z.(z\ z))\ (\lambda x.(x\ x)) \xrightarrow{n,lbeta} (\texttt{letrec}\ z = \lambda x.x\ x\ \texttt{in}\ (z\ z)) \xrightarrow{n,cp} (\texttt{letrec}\ z = \lambda x.x\ \ x\ \texttt{in}\ ((\lambda x.x\ \ x)\ \ z)) \xrightarrow{n,lbeta} (\texttt{letrec}\ z\ =\ \lambda x.x\ \ x\ \texttt{in}\ (\texttt{letrec}\ x_1\ =\ z\ \texttt{in}\ (x_1\ x_1))) \xrightarrow{n,llet} (\texttt{letrec}\ z = \lambda x.x\ x, x_1 = z\ \texttt{in}\ (x_1\ x_1))$.

6 Contextual Equivalence

We define contextual equivalence w.r.t. terminating normal order reduction sequences.

Definition 12 (contextual preorder and equivalence). *Let s, t be terms. Then:*

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

Note that we permit contexts such that $C[s]$ may be an open term. In appendix H we show that $s \leq_c t$ is equivalent to:

$$\forall C[\cdot]: \ C[s], C[t] \text{ are closed} \Rightarrow \Big(C[s]{\Downarrow} \Rightarrow C[t]{\Downarrow}\Big)$$

A *precongruence* $\leq_c$ is a preorder on expressions, such that $s \leq_c t \Rightarrow C[s] \leq_c C[t]$ for all contexts C. A *congruence* is a precongruence that is also an equivalence relation.

Proposition 1. *$\leq_c$ is a precongruence, and $\sim_c$ is a congruence.*

Proof. Let $s \leq_c t, t \leq_c r$, let C be a context such that $C[s]{\Downarrow}$. Then $C[t]{\Downarrow}$. Since $t \leq_c r$, we have also $C[r]{\Downarrow}$. Hence $s \leq_c r$.
To show the congruence property, let $s \leq_c t$ and let C be a context. To show $C[s] \leq_c C[t]$, let D be a further context. If $D[C[s]]{\Downarrow}$, we can use the context DC for $s \leq_c t$, and see that $D[C[t]]{\Downarrow}$. This shows $C[s] \leq_c C[t]$.

We define strictness of functions and expressions consistent with the notions from denotational semantics.

Definition 13. *An expression s is* strict, *iff $(s\ \Omega) \sim_c \Omega$.*
An expression s is strict in the i^{th} argument for arity n, *iff $1 \leq i \leq n$ and for all closed expressions $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots t_n$: $(s\ t_1 \ldots t_{i-1}\ \Omega\ t_{i+1} \ldots t_n) \sim_c \Omega$.*
An expression s is strict in the subexpression s_0, *iff for the term s' that is constructed from s by replacing s_0 by Ω, we have $s' \sim_c \Omega$. Here we mean by subexpression also the position within the superterm.*

Knowing strictness of functions and strict subterms of terms helps to rearrange evaluation and is thus of importance for optimizations and parallelization of non-strict programs.

7 Context Lemma

The Context Lemma restricts the criterion for contextual equivalence to reduction contexts. This restriction is of great value in proving the conservation of contextual equivalence by certain reductions, since there is no need to introduce parallel reductions like Barendregt's 1-reduction [Bar84]. Its proof can be found in appendix B.

Lemma 2. *Let s, t be terms. If for all reduction contexts R: $(R[s]\Downarrow \Rightarrow R[t]\Downarrow)$, then $\forall C : (C[s]\Downarrow \Rightarrow C[t]\Downarrow)$; I.e. $s \leq_c t$.*

8 Extra Reduction Rules and Equivalence of Reductions

The following lemma shows that `letrec`s in reduction contexts can immediately be moved to the top level environment.

Lemma 3. *Let $t = (\texttt{letrec}\ Env\ \texttt{in}\ t')$ be an expression, and R be a reduction context. Then*

1. *If $R = (\texttt{letrec}\ Env_R\ \texttt{in}\ R')$, where R' is a weak reduction context, then $R[(\texttt{letrec}\ Env\ \texttt{in}\ t')] \xrightarrow{n,lll,+} (\texttt{letrec}\ Env_R, Env\ \texttt{in}\ R'[t'])$.*
2. *If $R = (\texttt{letrec}\ Env_R, x = R'\ \texttt{in}\ r)$, where R' is a weak reduction context, then $R[(\texttt{letrec}\ Env\ \texttt{in}\ t')] \xrightarrow{n,lll,+} (\texttt{letrec}\ Env_R, Env, x = R'[t']\ \texttt{in}\ r)$, and $(\texttt{letrec}\ Env_R, Env, x = R'[\cdot]\ \texttt{in}\ r)$ is a reduction context.*
3. *If R is not a `letrec`-expression, i.e. R is a weak reduction context, then $R[(\texttt{letrec}\ Env\ \texttt{in}\ t')] \xrightarrow{n,lll,*} (\texttt{letrec}\ Env\ \texttt{in}\ R[t'])$, and $(\texttt{letrec}\ Env\ \texttt{in}\ R[\cdot])$ is a reduction context.*

Proof. This follows by induction on the number of reductions, using the definition of reduction context and weak reduction context and the (lll)-reductions.

Definition 14. *We define (n,mll) to stand for the $\xrightarrow{n,lll,*}$-reduction which shifts `letrec`-environments in reduction contexts to the top of the expression as in Lemma 3.*

8.1 Extra Reduction Rules

Definition 15. *We define further transformation rules in figure 3. The union of (gc1) and (gc2) is called (gc), the union of (cpx-in) and (cpx-e) is called (cpx), the union of (cpcx-in) and (cpcx-e) is denoted as (cpcx).*
A constructor application of the form $(c\ x_1 \ldots x_n)$ is called a cx-expression.

We require three specialized reduction rules: (case-cx) *is like (case) with the difference, that if the constructor application is a cx-expression, then the rule has no effect on the binding. The extra reduction rule (cpcxnoa) can be seen as an abbreviation of a (cpcx) with subsequent (cpx) and (gc)-reductions.*
Note that the (useless) reduction $\texttt{letrec}\ x = x\ \texttt{in}\ t \to \texttt{letrec}\ x = x\ \texttt{in}\ t$ *is not allowed as an instance of the (cpx)-rule. Note also that the reduction (lwas) includes the reductions (lapp), (lcase), (lseq).*

Definition 16. *For a given term* t*, the measure* $\mu_{lll}(t)$ *is a pair* $(\mu_1(t), \mu_2(t))$*, ordered lexicographically. The measure* $\mu_1(t)$ *is the number of* `letrec`*-subexpressions in* t*, and* $\mu_2(t)$ *is the sum of* $\texttt{lrdepth}(C)$ *for all* `letrec`*-subexpressions* s *with* $t \equiv C[s]$*, where* `lrdepth` *is defined as follows:*

$$\begin{aligned}\texttt{lrdepth}([\cdot]) &= 0\\ \texttt{lrdepth}(C_{(1)}[C'[]]) &= \begin{cases} 1 + \texttt{lrdepth}(C'[]) & \text{if } C_{(1)} \text{ is not a } \texttt{letrec}\\ \texttt{lrdepth}(C'[]) & \text{if } C_{(1)} \text{ is a } \texttt{letrec}\end{cases}\end{aligned}$$

The following termination property of (lll) is required in later proofs.

Proposition 2. *The reduction (lll) is terminating, I.e. there are no infinite reductions sequences consisting only of (lll)-reductions.*

Proof. This holds, since $t_1 \xrightarrow{lll} t_2$ implies $\mu_{lll}(t_1) > \mu_{lll}(t_2)$, and the ordering induced by the measure is well-founded.

8.2 Equivalence of Reductions

In the appendix (section C.8 of appendix C) we prove the following two theorems:

Theorem 1. *All the reductions in the base calculus maintain contextual equivalence. I.e. whenever* $t \xrightarrow{a} t'$*, with* $a \in \{cp, lll, case, seq, lbeta\}$*, then* $t \sim_c t'$*.*

Theorem 2. *The reductions (ucp), (cpx), (cpax), (gc), (lwas), (cpcx), (abs), (abse), (xch), (cpcxnoa) and(case-cx) maintain contextual equivalence. I.e. whenever* $t \xrightarrow{a} t'$*, with* $a \in \{ucp, cpx, cpax, gc, lwas, cpcx, abs, abse, xch, cpcxnoa, case\text{-}cx\}$*, then* $t \sim_c t'$*.*

Proposition 3. *If* $t\Uparrow$ *and* $t \xrightarrow{a} t'$*, where* a *is any reduction (lll), (seq), (lbeta), (case), (gc) (cpx), (cpax), (ucp), (lwas), then also* $t'\Uparrow$*.*

Proof. This follows from the contextual equivalences (see Theorems 1 and 2).

Theorem 3 (Standardization). *Let* t *be a term such that* $t \xrightarrow{*} t'$*, where* t' *is a WHNF, and the reductions are base reductions or extra reductions. Then* $t\Downarrow$*.*

Proof. This follows from Theorems 1 and 2.

(gc1)	$(\texttt{letrec}\ \{x_i = s_i\}_{i=1}^n, Env\ \texttt{in}\ t) \to (\texttt{letrec}\ Env\ \texttt{in}\ t)$ if for all $i : x_i$ does not occur in Env nor in t
(gc2)	$(\texttt{letrec}\ \{x_i = s_i\}_{i=1}^n\ \texttt{in}\ t) \to t$ if for all $i : x_i$ does not occur in t
(cpx-in)	$(\texttt{letrec}\ x = y, Env\ \texttt{in}\ C[x]) \to (\texttt{letrec}\ x = y, Env\ \texttt{in}\ C[y])$ where y is a variable and $x \neq y$
(cpx-e)	$(\texttt{letrec}\ x = y, z = C[x], Env\ \texttt{in}\ t) \to (\texttt{letrec}\ x = y, z = C[y], Env\ \texttt{in}\ t)$ where y is a variable and $x \neq y$
(cpax)	$(\texttt{letrec}\ x = y, Env\ \texttt{in}\ s) \to (\texttt{letrec}\ x = y, Env[y/x]\ \texttt{in}\ s[y/x])$ where y is a variable, $x \neq y$ and $y \in FV(s, Env)$
(cpcx-in)	$(\texttt{letrec}\ x = c\ \overrightarrow{t}, Env\ \texttt{in}\ C[x]) \to (\texttt{letrec}\ x = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^{\mathrm{ar}(c)}, Env\ \texttt{in}\ C[c\ \overrightarrow{y}])$
(cpcx-e)	$(\texttt{letrec}\ x = c\ \overrightarrow{t}, z = C[x], Env\ \texttt{in}\ t) \to (\texttt{letrec}\ x = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^{\mathrm{ar}(c)}, z = C[c\ \overrightarrow{y}], Env\ \texttt{in}\ t)$
(abs)	$(\texttt{letrec}\ x = c\ \overrightarrow{t}, Env\ \ \texttt{in}\ s) \to (\texttt{letrec}\ x = c\ \overrightarrow{x}, \{x_i = t_i\}_{i=1}^{\mathrm{ar}(c)}, Env\ \ \texttt{in}\ s)$ where $\mathrm{ar}(c) \geq 1$
(abse)	$(c\ \overrightarrow{t}) \to (\texttt{letrec}\ \{x_i = t_i\}_{i=1}^{\mathrm{ar}(c)}\ \texttt{in}\ c\ \overrightarrow{x})$ where $\mathrm{ar}(c) \geq 1$
(xch)	$(\texttt{letrec}\ x = t, y = x, Env\ \texttt{in}\ r) \ \to\ (\texttt{letrec}\ y = t, x = y, Env\ \texttt{in}\ r)$
(ucp1)	$(\texttt{letrec}\ Env, x = t\ \texttt{in}\ S[x]) \to (\texttt{letrec}\ Env\ \texttt{in}\ S[t])$
(ucp2)	$(\texttt{letrec}\ Env, x = t, y = S[x]\ \texttt{in}\ r) \to (\texttt{letrec}\ Env, y = S[t]\ \texttt{in}\ r)$
(ucp3)	$(\texttt{letrec}\ x = t\ \texttt{in}\ S[x]) \to S[t]$ 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
(lwas)	$AS^-_{(1)}[(\texttt{letrec}\ Env\ \texttt{in}\ s)] \to (\texttt{letrec}\ Env\ \texttt{in}\ AS^-_{(1)}[s])$ where $AS^-_{(1)}$ is a weak application surface context of main depth 1 (see Definition 6)
(cpcxnoa)	$(\texttt{letrec}\ x = c\ x_1 \ldots x_m, Env\ \texttt{in}\ C[x]) \to (\texttt{letrec}\ x = c\ x_1 \ldots x_m, Env\ \texttt{in}\ C[c\ x_1 \ldots x_m])$
(case-cx)	$(\texttt{letrec}\ x = (c_{T,j}\ x_1 \ldots x_n), Env\ \texttt{in}\ C[\texttt{case}_T\ x\ ((c_{T,j}\ y_1 \ldots y_n) \to s)\ alts]) \to \texttt{letrec}\ x = (c_{T,j}\ x_1 \ldots x_n), Env\ \texttt{in}\ C[(\texttt{letrec}\ y_1 = x_1, \ldots, y_n = x_n\ \texttt{in}\ s)]$
(case-cx)	$\texttt{letrec}\ x = (c_{T,j}\ x_1 \ldots x_n), Env, y = C[\texttt{case}_T\ x\ ((c_{T,j}\ y_1 \ldots y_n) \to s)\ alts]\ \texttt{in}\ r \to\ \texttt{letrec}\ x = (c\ x_1 \ldots x_n), Env, y = C[(\texttt{letrec}\ y_1 = x_1, \ldots, y_n = x_n\ \texttt{in}\ s)]\ \texttt{in}\ r$
(case-cx)	like (case) in all other cases

Fig. 3. Extra Reduction Rules

9 A Convergent Rewrite System of Simplifications

Definition 17. *As* simplification rules *we will use (lwas), (llet), (gc), (cpax).*

Note that the rule (lwas) includes (lseq), (lcase), (lapp), but not (llet). The simplification rules (lwas), (llet), (gc), (cpax) maintain contextual equivalence, which is proved in Theorem 1 and Theorem 2. For definitions of confluence and local confluence see e.g. [BN98].
In appendix G we prove the following result:

Theorem 4. *The set of reductions (lwas), (llet), (gc), (cpax) is confluent (up to α-renaming) and terminating.*

Proposition 4. *The simplification rules, if applied exhaustively, produce a normal form with the following properties:*

- *There are no unnecessary bindings.*
- *The* `letrec`*-environments are joined at the top of the term, at the top in the body of abstractions, and at the top in the alternatives of cases.*

10 Length of Normal Order Reduction

We develop properties of different lengths measures of normal order reduction sequences, which are required in the proof of correctness of SAL (see section 15).

Definition 18. *Let t be a closed term with $t{\Downarrow}$ and $Red := nor(t)$. Then*

1. $\mathrm{rl}\sharp\sharp(Red)$ *is defined to be the number of (case), (lbeta), (seq)-reductions in Red.*
2. $\mathrm{rl}\sharp(Red)$ *is defined to be the number of (case), (lbeta), (seq), and (cp)-reductions in Red.*
3. $\mathrm{rl}\flat(Red)$ *is defined to be the number of (lll)-reductions in Red.*
4. $\mathrm{rl}(Red) := \mathrm{rl}\sharp(Red) + \mathrm{rl}\flat(Red)$*. I.e., the number of all normal order reduction steps.*

Definition 19. *Let t be a closed concrete term with $t{\Downarrow}$, and let $Red := nor(t)$. Then*

1. $\mathrm{rl}\sharp\sharp(t) := \mathrm{rl}\sharp\sharp(Red)$
2. $\mathrm{rl}\sharp(t) := \mathrm{rl}\sharp(Red)$
3. $\mathrm{rl}\flat(t) := \mathrm{rl}\flat(Red)$
4. $\mathrm{rl}(t) := \mathrm{rl}\sharp(t) + \mathrm{rl}\flat(t)$.

The main measure in this paper will be $\mathrm{rl}\sharp\sharp(\cdot)$.

Proposition 5. *Let t be a closed expression with $t{\Downarrow}$ and $Red_1 := nor(t)$.*

1. *If $Red_1 = \xrightarrow{n,a} \cdot\ Red_2$ with $a \in \{case, seq, lbeta, cp, lll\}$, then $\mathrm{rl}(Red_1) = \mathrm{rl}(Red_2) + 1$.*

2. *If* $Red_1 = \xrightarrow{n,a} \cdot Red_2$ *with* $a \in \{case, seq, lbeta, cp\}$, *then* $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2) + 1$.
3. *If* $Red_1 = \xrightarrow{n,a} \cdot Red_2$ *with* $a \in \{case, seq, lbeta\}$, *then* $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2) + 1$.

Proof. Trivial.

Theorem 5. *Let* t_1, s_1 *be closed and terminating concrete expressions with* $t_1\Downarrow$ *and* $t_1 \to s_1$. *Then* $s_1\Downarrow$ *and the following holds:*

1. *If* $t_1 \xrightarrow{a} s_1$ *with* $a \in \{case, seq, lbeta, cp\}$, *then* $\mathrm{rl}(t_1) \geq \mathrm{rl}(s_1)$, $\mathrm{rl}\sharp(t_1) \geq \mathrm{rl}\sharp(s_1)$ *and* $\mathrm{rl}\sharp\sharp(t_1) \geq \mathrm{rl}\sharp\sharp(s_1)$.
2. *If* $t_1 \xrightarrow{\mathcal{S},a} s_1$ *with* $a \in \{caseS, seqS, lbeta, cpS\}$, *then* $\mathrm{rl}\sharp(t_1) \geq \mathrm{rl}\sharp(s_1) \geq \mathrm{rl}\sharp(t_1) - 1$ *and* $\mathrm{rl}\sharp\sharp(t_1) \geq \mathrm{rl}\sharp\sharp(s_1) \geq \mathrm{rl}\sharp\sharp(t_1) - 1$. *For* $a = cpS$, *the equation* $\mathrm{rl}\sharp\sharp(t_1) = \mathrm{rl}\sharp\sharp(s_1)$ *holds.*
3. *If* $t_1 \xrightarrow{a} s_1$ *with* $a \in \{lll, gc\}$, *then* $\mathrm{rl}(t_1) \geq \mathrm{rl}(s_1)$, $\mathrm{rl}\sharp(t_1) = \mathrm{rl}\sharp(s_1)$ *and* $\mathrm{rl}\sharp\sharp(t_1) = \mathrm{rl}\sharp\sharp(s_1)$. *For* $a = gc1$ *in addition* $\mathrm{rl}(t_1) = \mathrm{rl}(s_1)$ *holds.*
4. *If* $t_1 \xrightarrow{a} s_1$ *with* $a \in \{cpx, cpax, xch, cpcx, abs\}$, *then* $\mathrm{rl}(t_1) = \mathrm{rl}(s_1)$, $\mathrm{rl}\sharp(t_1) = \mathrm{rl}\sharp(s_1)$ *and* $\mathrm{rl}\sharp\sharp(t_1) = \mathrm{rl}\sharp\sharp(s_1)$.
5. *If* $t_1 \xrightarrow{ucp} s_1$, *then* $\mathrm{rl}(t_1) \geq \mathrm{rl}(s_1)$, $\mathrm{rl}\sharp(t_1) \geq \mathrm{rl}\sharp(s_1)$ *and* $\mathrm{rl}\sharp\sharp(t_1) = \mathrm{rl}\sharp\sharp(s_1)$.
6. *If* $t_1 \xrightarrow{lwas} s_1$, *then* $\mathrm{rl}\sharp\sharp(t_1) = \mathrm{rl}\sharp\sharp(s_1)$.

Proof. Details of the proof are in appendix F.

The next proposition shows that modifying the normal order reduction sequence to exploit strictness will not increase the number of (case)-, (cp)-, (seq)-, and (lbeta)-reductions required to reach a WHNF. Its proof is done in the appendix, see section F.9.

Proposition 6. *Let* t_1 *be a closed concrete (*LR*-)expression with* $t_1\Downarrow$ *and let* $t_1 = S[t_0]$, *where* t_0 *is a strict subterm of* t_1, S *is a surface context, and* t_0 *is an inner b-redex for* $b \in \{(caseS), (seqS), (lbeta), (cpS)\}$. *Let* $t_1 \xrightarrow{\mathcal{S},b} s_1$, *where the subexpression* t_0 *is reduced using the b-reduction.*
Then $\mathrm{rl}\sharp(t_1) = 1 + \mathrm{rl}\sharp(s_1)$.
If $b \neq (cpt)$, *then* $\mathrm{rl}\sharp\sharp(t_1) = 1 + \mathrm{rl}\sharp\sharp(s_1)$ *and if* $b = (cpt)$, *then* $\mathrm{rl}\sharp\sharp(t_1) = \mathrm{rl}\sharp\sharp(s_1)$.

10.1 Local Evaluation and Deep Subterms

We introduce deep and strict subterms, which have a reduction length strictly smaller than that of the top term.

Definition 20. *Let* $t = (\texttt{letrec}\ Env\ \texttt{in}\ t')$ *be a concrete expression, and let* $x \in LV(Env)$. *Then the* local evaluation *of* x *is defined as the reduction sequence of* t, *which corresponds to the normal order reduction sequence of* $(\texttt{letrec}\ Env\ \texttt{in}\ x)$, *only considering reductions that make modifications in* Env, *i.e. a possibly occurring last (n,cp) that replaces* x *in the normal order reduction sequence is omitted in the local evaluation.*

If the normal order reduction sequence of (`letrec` *Env* `in` x) *terminates with a WHNF, then the length corresponding to* $\mathrm{rl}\sharp(\cdot)$ *of a local evaluation is denoted as* $\mathrm{rl}\sharp_{loc}$(`letrec` *Env* `in` x).

Proposition 7. *Let* $t_1 =$ (`letrec` *Env* `in` t_1') *be a closed concrete* LR*-expression with* $t_1{\Downarrow}$. *Let* $x \in LV(Env)$ *where the binding is* $x = t_x$, *and* t_x *is a strict subexpression in* t_1.
Then $\mathrm{rl}\sharp(t_1) \geq \mathrm{rl}\sharp_{loc}$(`letrec` *Env* `in` x) *and* $\mathrm{rl}\sharp\sharp(t_1) \geq \mathrm{rl}\sharp\sharp$(`letrec` *Env* `in` x).

Proof. The proof is in appendix F.10.

Definition 21. *Let* $t =$ (`letrec` *Env* `in` t_1). *The subterm* t_x *in the binding* $x = t_x$ *in Env is called a* deep subterm *in* t *provided* t_1 *is either an application, a* `seq`*-expression, a* `case`*-expression, or a variable, say* x_1. *In the latter case a part of the environment must be of the form* $\{x_n = t_2, x_{n-1} = x_n, \ldots, x_1 = x_2\}$, *where* t_2 *is an application, a* `seq`*-expression, or a* `case`*-expression. Furthermore,* $x \in LV(Env)$ *such that* $x \notin \{x_1, \ldots, x_n\}$.

The next proposition shows that for deep and strict subexpressions, the number $\mathrm{rl}\sharp\sharp(\cdot)$ of local normal order reductions to a WHNF is less than the corresponding number for the top term. The proof is in appendix F.10.

Proposition 8. *Let* $t_1 =$ (`letrec` *Env* `in` t_1') *be a closed concrete* LR*-expression with* $t_1{\Downarrow}$. *Let* $x \in LV(Env)$ *be a variable with binding* $x = t_x$, *for a strict and deep subterm* t_x *in* t_1, *where* t_x *is not a* `letrec`*-expression.*
Then $\mathrm{rl}\sharp\sharp(t_1) > \mathrm{rl}\sharp\sharp$(`letrec` *Env* `in` x).

11 Concrete Subterms and Environments

The goal of this section is to show that in certain situations, concrete subterms or even concrete parts of `let`-environments can be copied without consequence for the $\sim_c$-equivalence of expressions.
In the appendix A we prove the following theorem.

Theorem 6. *Let* $t =$ (`letrec` *Env* `in` $(c\ t_1 \ldots t_{\mathrm{ar}(c)})$) *be a closed expression, where* $Env = \{y_i = s_i\}_{i=1}^n$, *and let* $t_j' :=$ (`letrec` Env_j `in` t_j'') *for* $j = 1, \ldots, \mathrm{ar}(c)$, *where* Env_j *and* t_j'' *is Env and* t_j, *respectively, renamed by* $\rho_j := \{y_i \to y_{j,i} \mid i = 1, \ldots, n\}$.
Then for all j*: the expressions* t_j' *are closed and* $t \sim_c (c\ t_1'\ \ldots\ t_{\mathrm{ar}(c)}')$

Corollary 1. *Let* t *be a closed expression, which has a CWHNF. Then there is a constructor* c, *and for* $j = 1, \ldots, \mathrm{ar}(c)$ *there are closed terms* t_j *with* $t \sim_c (c\ t_1\ \ldots\ t_{\mathrm{ar}(c)})$.

Proof. t has a CWHNF (`letrec` *Env* `in` $(c\ t_1\ \ldots, t_{\mathrm{ar}(c)})$), such that $t \sim_c$ (`letrec` *Env* `in` $(c\ t_1\ \ldots, t_{\mathrm{ar}(c)})$). Theorem 6 shows that there are closed expressions t_i' for $i = 1, \ldots, \mathrm{ar}(c)$ such that $t \sim_c (c\ t_1'\ \ldots\ t_{\mathrm{ar}(c)}')$

12 Abstract Terms

Abstract terms are expressions from LR extended with set constants used in the formulation of strictness properties and strictness analysis algorithms, like $\bot$, $\top$, *Inf* etc. These constants were already used in [Nöc92,Nöc93,vEGHN93,Sch00]. The notation for sets of terms was coined "demands" in [Sch00]. Note that we only use set constants, up to *Fun*, where the semantics are sets of expressions that are closed w.r.t. $\leq_c$.

12.1 Set Constants

Let $\mathcal{U} = \{\bot, Fun\} \cup \{U_1, \ldots, U_K\}$ be a finite set of names of set constants. The set constants U_i are called *proper* set constants. For every proper set constant U_i there is a defining rule

$$(Eq_i): \quad U_i = \{\bot\} \cup r_{i,1} \cup \ldots \cup r_{i,n_i}$$

where r_i may be *Fun* or an expression $(c\ u'_1 \ldots u'_{\mathrm{ar}(c)})$, where u'_j are proper set constants or $\bot$.
With $rhs_{Eq}(U_i)$ we denote the right hand side of Eq_i. The restriction excludes *Fun* in expressions $(c\ u'_1 \ldots u'_{\mathrm{ar}(c)})$ in the right hand side, however, it is possible to define a set constant $Fun_\bot := \{\bot\} \cup Fun$, which corresponds to a lifted *Fun*, and use it in other right hand sides.
A mapping ψ from set constants to subsets of L^0, the set of closed concrete expressions, is called a *sc-interpretation* if it satisfies the following conditions:

1. $\psi(\bot) = \{t \mid t \in L^0 \text{ and } t \sim_c \Omega\}$
2. $\psi(Fun) = \{t \mid t \text{ is a closed FWHNF}\}$
3. $t \in \psi(U_i), t \sim_c t' \Rightarrow t' \in \psi(U_i)$.

For sc-interpretations ψ_1, ψ_2, we write $\psi_1 \leq \psi_2$, iff for all $i : \psi_1(U_i) \subseteq \psi_2(U_i)$.
We define an extension ψ^e for sc-interpretations ψ as follows:

$$\psi^e(\{\bot\} \cup r_1 \cup \cdots \cup r_n) := \psi(\bot) \ \cup \ \bigcup \{\psi(Fun) \mid r_j = Fun\}$$
$$\cup \bigcup \{c_j\ t_{j,1}\ \ldots\ t_{j,ar(c_j)} \mid r_j = c_j\ u_{j,1}\ \ldots\ u_{j,ar(c_j)} \wedge t_{j,l} \in \psi(u_{j,l})\}$$

which is obviously monotone. The equations Eq_i for the set constants define an operator Ψ on sc-interpretations as follows: $\Psi(\psi) := \psi^e \circ rhs_{Eq}$. The operator Ψ is monotone and has a greatest fixed point ψ^*.

Definition 22. *For every set constant $u \in \mathcal{U}$ we define a semantics $\gamma(u) \subseteq L^0$:*

$$\gamma(u) := \ \psi^*(u) \text{ for the greatest fixpoint } \psi^* \text{ of } \Psi.$$

Remark 2. The greatest fixpoint ψ^* of Ψ can be computed as follows. Let ψ_0 be the sc-interpretation with $\psi_0(U_i) = L^0$ for $i = 1, \ldots, K$. With $\psi_j := \Psi^j(\psi_0)$, the j-fold application of Ψ, for every $i = 1, \ldots, K$, the equation $\psi^*(U_i) = \bigcap_j \psi_j(U_i)$ holds. This representation of the fixpoint allows co-induction proofs in the style of induction proofs.

In the following we assume that the set constants $\top$ and *Inf* are proper set constants and defined with the defining equations:

$$\begin{aligned}\top &:= \{\bot\} \cup \mathit{Fun} \cup (c_1\ \top \ldots \top) \cup \ldots \cup (c_N\ \top\ \ldots\ \top)\\ &\quad \text{where } c_1, \ldots, c_N \text{ are all constructors}\\ \mathit{Inf} &:= \{\bot\} \cup (\top : \mathit{Inf})\\ &\quad \text{where ``}:\text{'' is the binary constructor for lists}\end{aligned}$$

Lemma 4. *Let t_1, t_2 be closed expressions, which have CWHNFs. Then $t_1 \leq_c t_2$ iff there is a constructor c, and for $j = 1, \ldots, \mathrm{ar}(c)$ there are closed terms $t_{1,j}, t_{2,j}$ such that $t_{1,j} \leq_c t_{2,j}$, $t_1 \sim_c (c\ t_{1,1}\ \ldots\ t_{1,\mathrm{ar}(c)})$ and $t_2 \sim_c (c\ t_{2,1}\ \ldots\ t_{2,\mathrm{ar}(c)})$.*

Proof. The if-direction is obvious.
To prove the other direction, let $t_1 \leq_c t_2$. Then Corollary 1 shows that there are closed expressions $t_{1,j}, t_{2,j}$ for $j = 1, \ldots, \mathrm{ar}(c)$, such that $(c\ t_{1,1}\ \ldots\ t_{1,\mathrm{ar}(c)}) \sim_c t_1 \leq_c t_2 \sim_c (c\ t_{2,1}\ \ldots\ t_{2,\mathrm{ar}(c)})$. Using contexts $C_i := (\mathtt{case}_T[\cdot]\ (c\ x_1\ \ldots x_{\mathrm{ar}(c)}) \to x_i\ \ldots)$ for $i = 1, \ldots, n$, it is easy to see that for $j = 1 \ldots, \mathrm{ar}(c)$: $t_{1,j} \leq_c t_{2,j}$.

Lemma 5. *For every proper set constant u: $\gamma(u)$ is down-closed. I.e. $t_1 \leq_c t_2 \in \gamma(u) \Rightarrow t_1 \in \gamma(u)$.*

Proof. We use Remark 2 and show by induction that for every i the sets $\psi_i(U_k)$ for all k are down-closed. The base case is $\psi_0(U_k) = L^0$, for all k, which is trivially down-closed.
Let i be a positive integer. We use as induction hypothesis that $\psi_{i-1}(U_k)$ is down-closed for all k. Let t_1, t_2 be closed expressions with $t_1 \leq_c t_2 \in \psi_i(U_k)$.
If $t_1 \sim_c \Omega$, then the lemma holds since every defining equation contains a component $\bot$.
If t_1 has an FWHNF, then t_2 must also have an FWHNF, and hence *Fun* is a component of the defining equation for U_k, and since iterations of Ψ cannot remove any expressions with FWHNFs, $t_1 \in \psi_i(U_k)$.
If t_1 has a CWHNF for constructor c, then by Lemma 4 there are closed terms $t_{1,j}, t_{2,j}$ with $t_1 \sim_c (c\ t_{1,1}\ \ldots\ t_{1,\mathrm{arc}})$, $t_2 \sim_c (c\ t_{2,1}\ \ldots\ t_{2,\mathrm{arc}})$ and $t_{1,j} \leq_c t_{2,j}$. In the defining equation for U_k, there is a component $(c\ u_{k,1}\ \ldots u_{k,\mathrm{ar}(c)})$ on the right hand side with $t_2 \in \psi_{i-1}(c\ u_{k,1}\ \ldots u_{k,\mathrm{ar}(c)})$.
The definition of ψ implies that $t_{2,j} \in \psi_{i-1}(u_{k,j})$ for all j. Using the induction hypothesis and the fact that only proper set constants and $\bot$ are possible as $u_{k,j}$, it is obvious that $\psi_{i-1}(u_{k,j})$ is down-closed and hence $t_{1,j} \in \psi_{i-1}(u_{k,j})$ for all j. This implies $t_1 \in \psi_i(U_k)$.
We have shown that for all k and all i : $\psi_i(U_k)$ is down-closed. Since $\gamma(U_k) = \bigcap_i \psi_i(U_k)$, we obtain that $\gamma(U_k)$ is down-closed, too.

Lemma 6. *The equation $\gamma(\top) = L^0$ holds.*

Proof. This follows using the same methods as in the proof of Lemma 5, since in the defining equation for $\top$ all used proper set constants are equal to $\top$.

Lemma 7. – *Every expression $t \in \gamma(\top)$ is either in $\gamma(\bot)$ or $\gamma(Fun)$, or contextually equivalent to a term of the form $(c\ t_1 \ldots t_n)$ where c is a constructor and t_i are closed expressions.*
– *Every expression $t \in Inf$ is either in $\gamma(\bot)$ or is contextually equivalent to a term of the form $t_1 : t_2$, where t_1, t_2 are closed and $t_2 \in \gamma(Inf)$.*

Remark 3. The definition of abstract constants does not cover the non-down-closed demands in [Sch00] like Fin, the abstract constant representing all finite lists.

12.2 Terms Including Set Constants: ac-Labeled Terms

We define structured terms including set constant, which are used to represent sets of concrete expressions.

Definition 23. *We extend the language* LR *by the set constants in $\mathcal{U}$ giving the language* LR_U*, where set constants can also be used as subexpressions.*
A closed LR_U*-term t is called an* ac-labeled term*, if it is of the form* $(\texttt{letrec}\ Env_{ac}, Env_{up}\ \texttt{in}\ r)$*, where the environment for abstract (set) constants is $Env_{ac} = \{x_1 = u_1, \ldots, x_n = u_n\}$, where u_i are set constants, and where the variables $LV(Env_{ac})$ are exactly the variables that are labelled "ac". The ac-labeled variables will be called* ac-variables*. The variables in $LV(Env_{up})$ are unlabeled and called* up-variables*. Set constants are only permitted in the ac-environment.*

Definition 24. *We also want to use strictness of built-in functions and the results of previous analyses. Therefore we assume that there is already a finite family of finite sets of concrete closed expressions (functions) $SF^{n,i}$ for $i, n \in \mathbb{N}$ with $1 \leq i \leq n$, such that every expression $f \in SF^{n,i}$ is known to be strict in its i^{th} argument for arity n. These functions are assumed to be defined via a binding $x = f$ in the top level* `letrec`*, where the variable x is in the upper part. We will use the definition of strictness for concrete expressions as well as for abstract expressions.*

Definition 25. *We define ACL-reductions, i.e. reductions on ac-labeled expressions.*

1. ACL-contexts: *For ac-labeled terms, we also speak of contexts, reduction contexts, surface contexts, etc., where the definition is the same, however, the set constants are treated like free variables.*
2. ACL-reductions:
   – ACL-base reduction *For ac-labeled terms, the reductions from Definition 8 that are used for concrete expressions are also defined, where set constants are treated like free variables. Every reduction has its inner redex and modifies the upper part.*
   – ACL-seq-Fun-reduction *The following reduction rule will be used on ac-labeled expression:*

$$\textit{(seq-Fun)}\ (\texttt{seq}\ x\ t) \to t \qquad \textit{if } x \textit{ is bound to } Fun.$$

- ACL-extra reductions *For ac-labeled terms, the reductions from Definition 15 that are used for concrete expressions are also defined, where set constants are treated like free variables. Every reduction has its inner redex and the modification in the upper part.*
  An unusual case occurs if (gc) eliminates bindings of set constants.
  There are the following restrictions and exceptions:
  The set constants must not be copied, i.e., neither the rule (cpx) nor (ucp) are allowed to copy a set constant to another position. The rule (xch) is only allowed if it does not change the ac-labeling.

3. ACL-normal order reductions: *Normal-order reduction is also defined accordingly. The rule (seq-Fun) is also permitted in ACL-normal order reductions. Note that an ACL-normal order reduction may get stuck if a set constant is in a reduction context.*
4. ACL-bot-reductions: *The bot-reduction rules are shown in figure 4.*

(beta-bot)	$D[(x\ y) \to D[x]$ if x ist bound to $\bot$ in D.
(hole)	$(\texttt{letrec}\ y = \bot, x = x, Env\ \texttt{in}\ r)$ $\to (\texttt{letrec}\ y = \bot, Env[y/x]\ \texttt{in}\ r[y/x])$
(case-bot)	$D[(\texttt{case}_T\ x\ \ldots((c_i\ y_1 \ldots y_n)\ \texttt{->}\ t)\ldots)] \to \quad D[x]$ if x is bound to $\bot$ in D
(app-bot)	$D[(v\ t)] \ \to\ D[x]$ if v is a constructor application and x is bound to $\bot$ in D
(case-untyped)	$D[(\texttt{case}_T\ s\ alts)] \to \quad D[x]$ where x is bound to $\bot$ in D. If s is an abstraction or a constructor application whose top-constructor does not belong to the type T; or s is a variable that is bound in D to a constructor application whose top-constructor does not belong to the type T
(seq-bot)	$D[(\texttt{seq}\ x\ t)] \to \quad D[x]$ if x is bound to $\bot$ in D
(strict-bot)	$D[(f\ x_1 \ldots x_n)] \to \quad D[x_i]$ if f is strict in its i^{th} argument for arity n and x_i is bound to $\bot$ in D
(case-Fun-bot)	$D[(\texttt{case}_T\ x\ \ldots)] \to \quad D[y]$ if x is bound to *Fun*, where y is bound to $\bot$ in D
(merge-bot)	$(\texttt{letrec}\ x = \bot, y = \bot, Env\ \texttt{in}\ r) \to \quad (\texttt{letrec}\ x = \bot, Env[x/y]\ \texttt{in}\ r[x/y])$

Fig. 4. Reduction rules for $\bot$

For an abstract term t, let $\mathrm{simp}(t)$ *be the result of exhaustively applying simplification rules and bot-reduction rules, where as a final rule, we also allow* $(\texttt{letrec}\ x = \bot\ \texttt{in}\ x) \ \to\ \bot$ *in order to avoid clumsy notation.*

Proposition 9. *The application of simplification and bot-reduction rules terminates.*

Proof. The simplification rules terminate, and do not increase the size of a term, and the bot-reduction rules strictly reduce the size of a term.

Later, for the construction of expressions for subcases, there will be the following kinds of modifications on ac-labeled expressions. For the exact conditions, see Definition 33.

Definition 26. *1.* uu-modification: *A binding $x = u$ in the ac-part is modified into $x = u'$ where u' is another set constant.*
2. ucx-modification: *A binding $x = u$ in the ac-part is modified into $x = (c\ x_1 \dots x_{\mathrm{ar}(c)})$, where $x_i, i = 1, \dots, \mathrm{ar}(c)$ are new ac-variables, and bindings $x_i = u_i, i = 1, \dots, \mathrm{ar}(c)$ are added to the ac-part, where u_i are set constants. Moreover, the ac-label is removed from x, thus the binding $x = (c\ x_1\ \dots\ x_{\mathrm{ar}(c)})$ is shifted into the upper part.*
3. generalisation: $(\texttt{letrec}\ Env\ \texttt{in}\ C[s]) \to (\texttt{letrec}\ x = \top; Env\ \texttt{in}\ C[x])$.

Lemma 8. *Let t be an ac-labeled term. Reduction according to Definition 25 and modifications according to Definition 26 transform an ac-labeled closed expression into an ac-labeled closed expression.*

12.3 Concretizations of ac-Labeled Terms

We define concretizations s of ac-labeled terms t as concrete terms, where the relationship can be informally described as follows. There is an upper part in s, t that must be syntactically identical, and the bindings of the free variables must be such that the semantic content of the variables from s has to be contained in the corresponding variable of t, e.g. $(\texttt{letrec}\ y = 2, x = 1 : x\ \texttt{in}\ y : x)$ is a concretization of $(\texttt{letrec}\ \mathrm{top} = \top, \mathrm{inf} = \mathit{Inf}\ \texttt{in}\ \mathrm{top} : \mathrm{inf})$, where the variable renaming is $\{x \to \mathrm{inf}, y \to \mathrm{top}\}$. The situation is in general a bit more complicated due to sharing in the concretization.
For an ac-labeled term t, not every closed expressions $s \leq_c t$, will qualify as a concretization of t. However, it will turn out, that the set of concretizations is sufficient for our correctness proof.

Definition 27. *Let $t = (\texttt{letrec}\ Env_{t,ac}, Env_{t,up}\ \texttt{in}\ t')$ be an ac-labeled closed abstract term, where $Env_{t,ac} = \{y_1 = u_1, \dots, y_k = u_k\}$ and u_i are set constants. Then the set of* concretizations $\gamma(t)$ *is defined as follows:*
Let $s = (\texttt{letrec}\ Env_s\ \texttt{in}\ s')$ be a closed concrete term. Then $s \in \gamma(t)$ iff the following holds:

- *There is a split of the environment Env_s into $Env_s = Env_{s,sh} \cup Env_{s,ac} \cup Env_{s,up}$ with $|LV(Env_{s,ac})| = k$, $FV(Env_{s,sh} \cup Env_{s,ac}) = \emptyset$ and $FV(Env_{s,up}, s') \subseteq LV(Env_{s,ac})$.*

- *The expressions* $s_1 := (\texttt{letrec}\ Env_{s,up}\ \texttt{in}\ s')$ *and* $t_1 := (\texttt{letrec}\ Env_{t,up}\ \texttt{in}\ t')$ *are equal up to a renaming of free variables. I.e., for* $LV(Env_{s,ac}) = \{x_1, \ldots, x_k\}$, $LV(t_1) = \{y_1, \ldots, y_k\}$ *appropriately ordered and for* ρ *defined by* $\rho(x_i) = y_i$ *for* $i = 1, \ldots, k$, *the equation* $\rho(s_1) = t_1$ *holds.*
- *For every* i: $(\texttt{letrec}\ Env_{s,sh}, Env_{s,ac}\ \texttt{in}\ x_i) \in \gamma(u_i)$,

Example 1. We want to show that $s \in \gamma(t)$ where $s = (\texttt{letrec}\ x_1 = r, x_2 = (\texttt{repeat}\ x_1)\ \texttt{in}\ x_2)$ and $t = (\texttt{letrec}\ \text{inf} = \mathit{Inf}\ \texttt{in}\ \text{inf})$ and r is a closed expression. For convenience, we omit the definition of `repeat` as `repeat = {\x -> x: (repeat x)}` in the respective upper environments.
The environments are: $Env_{t,ac} = \{\text{inf} = \mathit{Inf}\}$, $Env_{s,ac} = \{x_2 = (\texttt{repeat}\ x_1)\}$, $Env_{s,sh} = \{x_1 = r\}$. We only have to check that $(\texttt{letrec}\ x_1 = r, x_2 = (\texttt{repeat}\ x_1)\ \texttt{in}\ x_2) \in \gamma(\mathit{Inf})$. Correctness of reduction and Theorem 6 yield:

$$
\begin{array}{ll}
 & (\texttt{letrec}\ x_1 = r, x_2 = (\texttt{repeat}\ x_1)\ \texttt{in}\ x_2) \\
\sim_c & (\texttt{letrec}\ x_1 = r, x_2 = ((\texttt{letrec}\ x = x_1\ \texttt{in}\ x : \texttt{repeat}\ x))\ \texttt{in}\ x_2) \\
\sim_c & (\texttt{letrec}\ x_1 = r, x = x_1, x_2 = (x : \texttt{repeat}\ x)\ \texttt{in}\ x_2) \\
\sim_c & (\texttt{letrec}\ x_1 = r, x_2 = x_1 : \texttt{repeat}\ x_1\ \texttt{in}\ x_2) \\
\sim_c & (\texttt{letrec}\ x_1 = r, x_3 = \texttt{repeat}\ x_1\ \texttt{in}\ x_1 : x_3) \\
\sim_c & r\ :\ (\texttt{letrec}\ x_1 = r, x_3 = \texttt{repeat}\ x_1\ \texttt{in}\ x_3)
\end{array}
$$

Now the membership check means to test $r \in \gamma(\top)$, which holds, and to check $(\texttt{letrec}\ x_1 = r, x_3 = \texttt{repeat}\ x_1\ \texttt{in}\ x_3) \in \gamma(\mathit{Inf})$, which can now be proved using coinduction.

Proposition 10. *Let* $s \in \gamma(t)$, $t \to t'$ *by a bot-reduction as defined in Figure 4, and* $s{\Downarrow}$. *Then there exists a* s' *with* $s' \in \gamma(t')$, $s'{\Downarrow}$ *and* $\mathrm{rl}\sharp\sharp(s) = \mathrm{rl}\sharp\sharp(s')$.

Proof. This follows from Proposition 31, where for the application of the Proposition, the set constant $\bot$ in has to be replaced by the bot-term Ω.

13 The Calculus for Abstract Terms

13.1 Strict Functions and Positions

Definition 28. *Assume given the sets* $SF^{n,i}$ *the* SF-strict *contexts* $\mathcal{SFS}(t)$ *of a term* t *are defined as follows:*

1. *If* $t \equiv R[s]$ *then* $R \in \mathcal{SFS}(t)$.
2. *If* $t \equiv S[f\ t_1 \ldots t_n]$, $S \in \mathcal{SFS}(t)$ *and* $f \in SF^{n,i}$, *then* $S[f\ t_1 \ldots t_{i-1}[\cdot]t_{i+1} \ldots t_n] \in \mathcal{SFS}(t)$
3. *If* $t \equiv S[x]$, $S \in \mathcal{SFS}(t)$ *and* $t \equiv C[(\texttt{letrec}\ x = s, Env\ \texttt{in}\ t_0)]$ *then* $C[(\texttt{letrec}\ x = [\cdot], Env\ \texttt{in}\ t_0)] \in \mathcal{SFS}(t)$.
4. *If* $S \in \mathcal{SFS}(t)$ *and there is a term* t_0 *such that* $S[R^-_{(1)}[t_0]] \equiv t$ *then* $S[R^-_{(1)}[\cdot]] \in \mathcal{SFS}(t)$.

5. *If* $S \in \mathcal{SFS}(t)$ *and* $t \equiv S[(\texttt{letrec}\ Env\ \texttt{in}\ t_0)]$ *then* $S[(\texttt{letrec}\ Env\ \texttt{in}\ [\cdot])] \in \mathcal{SFS}(t)$.

Definition 29. *For ac-labeled expressions, we will use the term* strict position reduction *(*sp-reduction*), if the reduction is a base-reduction or an (ACL-seq-Fun)-reduction, and the inner redex of the reduction is in an SF-strict context. This reduction may not be unique, since there may be several SF-strict positions where an sp-reduction may be possible. We will use this notion also for reductions on concrete expressions if no confusion arises.*

Note that a subterm in an SF-strict context is also a strict subterm.

Definition 30. *The* proper *SF-strict contexts are defined as follows.*
Assume given the sets $SF^{n,i}$, *let* $t = (\texttt{letrec}\ x = t_x, Env\ \texttt{in}\ t_0)$, *and let* $(\texttt{letrec}\ x = [\cdot], Env\ \texttt{in}\ t_0) \in \mathcal{SFS}(t)$, *and* t_x *is a deep subterm of* t. *Then* $(\texttt{letrec}\ x = [\cdot], Env\ \texttt{in}\ t_0) \in \mathcal{PSFS}(t)$.

Lemma 9. *The* SF-strict *contexts and* proper SF-strict *contexts are surface-contexts.*

13.2 Inheritance of Concretizations

Lemma 10. *Let* t *be an ac-labeled expression,* $s \in \gamma(t)$, $s{\Downarrow}$ *and* $t \to t'$, *where the reduction is within the upper part and a reduction from Definition 25. Then there is* $s' \in \gamma(t')$, *such that* $s'{\Downarrow}, s \sim_c s'$ *and* $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s')$.

Proof. Let $s \in \gamma(t)$ with $s{\Downarrow}$.

1. Let the reduction be a base-reduction, or a non-exceptional ACL-extra-reduction, i.e. not the (gc)-reduction on bindings of set-constants. If the reduction is in the upper part of t, then the same reduction is possible on s giving s'. Since the effect of the reductions is the same, and since the expressions in s at the position of the set constants are the same as before the reduction, we see that $s' \in \gamma(t')$. Theorems 1 and 2 show that contextual equivalence holds, and Theorem 5 shows the claim on the reduction length.
2. Let the reduction be a (gc) that eliminates bindings of set constants. Then we cannot use (gc) on s, hence other arguments are required. We show that $s \in \gamma(t')$. Let $t = (\texttt{letrec}\ Env_{t,ac}, Env_{t,up}\ \texttt{in}\ t_0), s = (\texttt{letrec}\ Env_{s,sh}, Env_{s,ac}, Env_{s,up}\ \texttt{in}\ s_0)$, and let $Env'_{t,ac}$ be the reduced environment. For convenience assume that $LV(Env_{t,ac}) = LV(Env_{s,ac})$ has the bindings $x_i = u_i$, for $i = 1, \ldots, n$, and the name correspondence is according to the definition of concretization. Let $Env_{s,ac} = Env'_{s,ac} \cup Env''_{s,ac}$ with $LV(Env'_{s,ac}) = LV(Env'_{t,ac})$, and $Env'_{s,sh} = Env_{s,sh} \cup Env''_{s,ac}$. The condition $FV(Env_{s,up}, s_0) \subseteq LV(Env'_{s,ac})$ holds, since (gc) is applicable to t.
   From $x_i \in LV(Env'_{s,ac})$ for all i ,and from $s \in \gamma(t)$ we derive $(\texttt{letrec}\ Env_{s,sh}, Env'_{s,ac}, Env''_{s,ac}\ \texttt{in}\ x_i) \in \gamma(u_i)$, hence $s \in \gamma(t')$.

3. Let the reduction be a (seq-Fun)-reduction. It is sufficient to consider the case: $t := (\texttt{letrec}\ x = \mathit{Fun}, \ldots\ \texttt{in}\ C[(\texttt{seq}\ x\ r)]) \to (\texttt{letrec}\ x = \mathit{Fun}, \ldots\ \texttt{in}\ C[r])$. Let $s \in \gamma(t)$. Then $s = (\texttt{letrec}\ \mathit{Env}_{s,sh}, \mathit{Env}_{s,ac}, \mathit{Env}_{s,up}\ \texttt{in}\ C[(\texttt{seq}\ x\ r)])$, where $x \in LV(\mathit{Env}_{s,ac})$, and $(\texttt{letrec}\ \mathit{Env}_{s,sh}, \mathit{Env}_{s,ac}\ \texttt{in}\ x) \in \gamma(\mathit{Fun})$. The definition of $\gamma(\mathit{Fun})$ implies that there is a reduction sequence of $(\texttt{letrec}\ \mathit{Env}_{s,sh}, \mathit{Env}_{s,ac}\ \texttt{in}\ x)$ to a WHNF. The same reductions can be made on s giving $s'' = (\texttt{letrec}\ \mathit{Env}''_{s,sh}, \mathit{Env}''_{s,ac}, \mathit{Env}_{s,up}\ \texttt{in}\ C[(\texttt{seq}\ x\ r)])$, such that x is bound to an abstraction in $\mathit{Env}''_{s,ac}$. Moreover, $s \sim_c s''$, and $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s'')$. Now a (seq)-reduction is possible: $s'' = (\texttt{letrec}\ \ldots\ \texttt{in}\ C[(\texttt{seq}\ x\ r)]) \to (\texttt{letrec}\ \ldots\ \texttt{in}\ C[r]) =: s'$. We have $\mathrm{rl}\sharp\sharp(s'') \geq \mathrm{rl}\sharp\sharp(s')$. To check the conditions that $s' \in \gamma(t')$, the only missing part is $(\texttt{letrec}\ \mathit{Env}''_{s,sh}, \mathit{Env}''_{s,ac}\ \texttt{in}\ x_i) \in \gamma(u_i)$ for other set constants. This follows from Theorem 1, and since $\gamma(u_i)$ is closed w.r.t. $\sim_c$. We obtain $s' \in \gamma(t')$. □

Proposition 11. *Let t be an ac-labeled expression. If $t \to t'$ by an sp-reduction, but not a (cp), and $s \in \gamma(t)$, then there is an expression $s' \in \gamma(t')$ with $s \sim_c s'$ and $\mathrm{rl}\sharp\sharp(s) > \mathrm{rl}\sharp\sharp(s')$.*

Proof. We have only to argue that there is an expression $s' \in \gamma(t')$ with $\mathrm{rl}\sharp\sharp(s) > \mathrm{rl}\sharp\sharp(s')$. Using Proposition 6 and Lemma 9 in the proof of Lemma 10 shows the claim for sp-reductions that are not a (cp) and not a (seq-Fun)-reduction.
In the case of a (seq-Fun)-reduction, we extend the arguments from the proof of Lemma 9. The missing part is that the redex $(\texttt{seq}\ x\ r)$ in s'' remains a strict subterm after the reduction sequence $s \xrightarrow{*} s''$. This holds, since there are no modifications in Env_{up} and $C[(\texttt{seq}\ x\ r)]$. Now Proposition 6 shows the claim on the length.

Lemma 11. *Let $t = (\texttt{letrec}\ \mathit{Env}_{t,ac}, \mathit{Env}_{t,up}\ \texttt{in}\ t_{in})$ be an ac-labeled expression, $s \in \gamma(t)$ with $s{\Downarrow}$, let $x_1 = u_1, \ldots, x_N = u_N$ be the ac-bindings, $x_i = u_i$ be a fixed binding in $\mathit{Env}_{t,ac}$ and let $u_i = \{\bot\} \cup r_1 \cup \ldots \cup r_k$ be the defining equation for u_i. Let t_j for $j = 1, \ldots, k$ be the ucx-modification (see Definition 26) according to r_j. I.e. if $j = 0$, then t_0 is constructed from t by replacing u_i by $\bot$; if $r_j = \mathit{Fun}$, then t_j is constructed from t by replacing $x_i = u_i$ by $x_i = \mathit{Fun}$, and if $r_j = (c\ u_{i,1}\ \ldots\ u_{i,\mathrm{ar}(c)})$, then t_j is constructed from t by replacing $x_i = u_i$ by $x_i = (c\ x_{i,1} \ldots x_{i,\mathrm{ar}(c)})$, moving this binding into the upper part and adding the bindings $x_h = u_{i,h}$ for $h = 1, \ldots, \mathrm{ar}(c)$ to the ac-part.*
Then there is a $j \in \{0, \ldots, k\}$ and an $s' \in \gamma(t_j)$, such that $s \sim_c s'$ and $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s')$.

Proof. Let $s \in \gamma(t)$. Then s can be represented as $s := (\texttt{letrec}\ \mathit{Env}_{s,sh}, \mathit{Env}_{s,ac}, \mathit{Env}_{s,up}\ \texttt{in}\ s_{in})$. With $\overline{s} := (\texttt{letrec}\ \mathit{Env}_{s,sh}, \mathit{Env}_{s,ac}\ \texttt{in}\ x_i)$, we have in particular $\overline{s} \in \gamma(u_i)$.
If $\overline{s}{\Uparrow}$, then we let $s' := s, j = 0$, and have $s \in \gamma(t_0)$.
Assume $\overline{s}{\Downarrow}$. Then there is a normal-order reduction of $\overline{s}$ to a WHNF. If it is a FWHNF, then the same reductions can be made on s with the exception of the last (cp), resulting in $s' = (\texttt{letrec}\ \mathit{Env}'_{s,sh}, \mathit{Env}'_{s,ac}, \mathit{Env}_{s,up}\ \texttt{in}\ s_{in})$.

We have $s \sim_c s'$ and $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s')$ by Theorem 1 and 5. It is obvious that $r_1 = \mathit{Fun}$, and furthermore $s' \in \gamma(t_1)$. If the WHNF of $\overline{s}$ is a CWHNF, then let c be the corresponding constructor. The normal order reduction reduces this term to $\overline{s}'' := (\texttt{letrec}\ \mathit{Env}'_{s,sh}, \mathit{Env}'_{s,ac}\ \texttt{in}\ x_i)$, such that x_i is bound to an expression $(c\ a_1 \ldots a_{\mathrm{ar}(c)})$. The same reductions can be performed for s and produce an expression $s'' := (\texttt{letrec}\ \mathit{Env}'_{s,sh}, \mathit{Env}'_{s,ac}, \mathit{Env}_{s,up}\ \texttt{in}\ s_{in})$ with $s \sim_c s''$ and $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s'')$ by Theorem 1 and 5. There is a reduction sequence $\overline{s}'' \xrightarrow{(cpcx,cpx,gc)^*} \overline{s}'$, where the environment contains the bindings $x_i = (c\ x_{i,1} \ldots x_{i,\mathrm{ar}(c)}), x_{i,1} = a_1, \ldots, x_{i,\mathrm{ar}(c)} = a_{i,\mathrm{ar}(c)}$. The same reductions performed for s'' yield: $s'' \xrightarrow{(cpcx,cpx,gc)^*} s'$, where $s \sim_c s'$ and $\mathrm{rl}\sharp\sharp(s'') = \mathrm{rl}\sharp\sharp(s')$ by Theorem 5.
It remains to show that $s' \in \gamma(t_j)$ for some j. For any ac-variable $x_h \neq x_i$, the membership $(\texttt{letrec}\ \mathit{Env}'_{s,sh}, \mathit{Env}'_{s,ac}\ \texttt{in}\ x_h) \in \gamma(u_h)$ holds also in s', since (cpcx), (cpx), (gc) are correct program transformations. Since $\gamma(u_i) = \{\bot\} \cup \gamma(\mathit{Fun}) \cup \bigcup_{h=1,\ldots,k} \psi^*(r_h)$, there is some j such that $\overline{s}' \in \{(c\ b_1 \ldots b_{\mathrm{ar}(c)}) \mid b_h \in \gamma(u_{i,h}), h = 1, \ldots, \mathrm{ar}(c)\}$. For the thus chosen t_j, we show that $s' \in \gamma(t_j)$:
The modifications of s to generate s' are in the environments $\mathit{Env}_{s,sh}, \mathit{Env}_{s,ac}$, with the exception of the bindings $x_i = (c\ x_{i,1} \ldots x_{i,\mathrm{ar}(c)}), x_{i,1} = a_1, \ldots, x_{i,\mathrm{ar}(c)} = a_{i,\mathrm{ar}(c)}$, where the first binding is shifted into the upper part, and the other bindings are shifted into the ac-part of s'. This corresponds to the environment of t_j, hence the renaming condition is satisfied. We show that $(\texttt{letrec}\ \mathit{Env}'_{s,sh}, \mathit{Env}'_{s,ac}\ \texttt{in}\ x_{i,h}) \in \gamma(u_{i,h})$ for $h = 1, \ldots, \mathrm{ar}(c)$. This follows from Theorem 6, and since we have only used correct program transformations. The membership $(\texttt{letrec}\ \mathit{Env}_{s,sh}, \mathit{Env}_{s,ac}\ \texttt{in}\ x_h) \in \gamma(u_h)$ for $h \neq i$ follows since only correct program transformations are used.

13.3 Subset Relationship for Abstract Terms

The subset-relations w.r.t. concretization between two ac-labeled abstract terms is defined as follows:

Definition 31. *Let t_1, t_2 be two ac-labeled closed abstract terms.*
Then $t_1 \subseteq_\gamma t_2$ iff $\gamma(t_1) \subseteq \gamma(t_2)$.

A sufficient condition for $\subseteq_\gamma$ is given in the following lemma:

Lemma 12. *Let $t_1 = (\texttt{letrec}\ \mathit{Env}_1\ \texttt{in}\ t'_1)$ and $t_2 = (\texttt{letrec}\ \mathit{Env}_2\ \texttt{in}\ t'_2)$ be two closed abstract terms that are ac-labeled. Then $t_1 \subseteq_\gamma t_2$ if the following holds:*

- *The environment Env_1 is split into $\mathit{Env}_1 = \mathit{Env}_{1,ac} \uplus \mathit{Env}_{1,up}$, where $\mathit{Env}_{1,ac}$ is the ac-labeled part which is of the form $\{x_{1,1} = u_{1,1}, \ldots, x_{1,k} = u_{1,k}\}$.*
- *The environment Env_2 is split into $\mathit{Env}_2 = \mathit{Env}_{2,ac} \uplus \mathit{Env}_{2,up}$, where $\mathit{Env}_{2,ac}$ is the ac-labeled part, which is of the form $\{x_{2,1} = u_{2,1}, \ldots, x_{2,k} = u_{2,k}\}$.*
- *For $r_1 = (\texttt{letrec}\ \mathit{Env}_{1,up}\ \texttt{in}\ t'_1)$ and $r_2 = (\texttt{letrec}\ \mathit{Env}_{2,up}\ \texttt{in}\ t'_2)$, the equation $\rho(r_1) = r_2$ must hold, where $\rho(x_{1,i}) = x_{2,i}$ for $i = 1, \ldots, k$.*
- *For every i: $\gamma(u_{1,i}) \subseteq \gamma(u_{2,i})$.*

Proof. The conditions can directly be matched with the conditions in Definition 27.

We do not give an algorithm for detecting $\gamma(u_{1,i}) \subseteq \gamma(u_{2,i})$ based on the defining rules, since this is beyond the scope of this paper, however, we are sure that the relation is decidable. We will only use the relations $\bot \subseteq \gamma(u)$ and $\gamma(u) \subseteq \gamma(\top)$.

14 The Algorithm SAL

We present the algorithm SAL (strictness analyzer for a lazy functional language), which is a reformulation of the algorithm Nöcker implemented for Clean. The core is a method to detect non-termination of concretizations of abstract terms.
Intuitively, strictness of a function f is detected if the normal order reduction of $(f\ \bot)$ in the abstract language can only yield $\bot$ or nontermination. This may be represented by the set constant $\bot$, or by a proof that normal order reduction will not terminate. The calculus is also applicable for detecting more general forms of strictness. For example strictness in the i^{th} argument of an abstraction f can be detected by feeding $(f\ \top\ \ldots \top\ \bot\ \top\ \ldots \top)$ into the analyzer. By providing other set constants apart from $\top$ and $\bot$, even more complicated analyses are possible like a test for tail-strictness, or strictness under certain conditions.
Reduction of expressions $(\mathtt{case}_T\ \top\ \ldots)$ will require a case analysis, which is in [Nöc93] as a propagation of unions, whereas our calculus uses the equivalent method of generating a directed graph, in which the union of cases is represented by forking.

Definition 32. *The algorithm SAL uses the sets $SF^{n,i}$ of functions (see also Definition 24 and subsection 13.1), which are already known or shown to be strict in the i^{th} argument for arity n.*
The data structure for the algorithm SAL is a directed graph, where the nodes are labeled by ac-labeled abstract terms that are simplified. The edges may be labeled or not. The algorithm SAL starts with a directed graph consisting only of one node labeled with the simplified initial abstract term. This term must be ac-labeled.
Given a directed graph D, a new directed graph D' is constructed by using some rule from the definition 33 below. For every node added, we assume that the simplification rules (i.e. (lwas), (llet), (gc), (cpax)) and the bot-reduction rules (see Figure 4 and Definition 25) have been applied exhaustively.
The algorithm stops successfully, if all leaves are labeled with $\bot$, i.e. if every non-$\bot$ node has an outgoing edge.
If some rule generates a cycle in the graph such that no edge in the cycle has a label, then fail.

Definition 33. *The non-deterministic construction rules of SAL are:*

nred *Let a leaf L be labeled with t, and let $t \xrightarrow{a} t'$ be an sp-reduction. Let $t'' := \mathrm{simp}(t')$. Then generate a new node L' labeled with t'' and add a directed edge from L to L'. If a is (case), (lbeta), (seq), or (seq-Fun), then the edge is to be labeled with the kind of reduction.*

ired *Let a leaf L be labeled with t and let $t \xrightarrow{a} t'$ by a non-sp-reduction, which may be a (case), (seq), (lbeta), (cp), or (seq-Fun). Let $t'' := \mathrm{simp}(t')$. Then generate a new node L' labeled with t'' and add an unlabeled directed edge from L to L'.*

scsplit *Let a leaf L be labeled with t, and assume t has an occurrence of a proper set constant u. Let the defining equation for the set constant u have the right hand side $\{\bot\} \cup r_1 \cup \ldots \cup r_k$. Then generate new expressions $t_j, j = 0, \ldots, k$ from t, as follows*

- *For $j = 0$, let t_0 be generated from t by replacing u by $\bot$ in the ac-environment.*
- *If $r_j = Fun$, then t_j is generated by replacing u by Fun in the ac-environment.*
- *If $r_j = (c\ u_{j,1}\ \ldots\ u_{j,\mathrm{ar}(c)})$, then let t_j be the expression generated from t by replacing u by $(c\ x_1 \ldots x_{\mathrm{ar}(c_j)})$, where $x_i, i = 1, \ldots, \mathrm{ar}(c_j)$ are new variables. For $x_i, i = 1, \ldots, \mathrm{ar}(c)$ add $x_i = u_{j,i}$ to the ac-environment.*

For $j = 0, \ldots, k$ first simplify t_j, i.e. let $t'_j := \mathrm{simp}(t_j)$ and then construct new nodes L_j with label t'_j and add directed, unlabeled edges from L to L_j for all j.

subsume *If there is a leaf L with term label $t_1 \neq \bot$, and a node $N \neq L$ with term label t_2, and $t_1 \subseteq_\gamma t_2$, then add a directed unlabeled edge from L to N under the following condition: After completion of this operation, the graph does not contain a cycle of unlabeled directed edges.*

subsume2 *Let L be a leaf with term label t_1 where $\bot \neq t_1 = (\texttt{letrec}\ Env\ \texttt{in}\ t_{11})$, let N be a node with $N \neq L$ with term label t_2, and let one of the following two conditions hold:*

1. *For $C = (\texttt{letrec}\ Env\ \texttt{in}\ C') \in \mathcal{PSFS}(t_1)$ with $t_1 = C[t_{12}]$ and $t'_1 := \mathrm{simp}(\texttt{letrec}\ Env\ \texttt{in}\ t_{12})$, we have $t'_1 \subseteq_\gamma t_2$, or*
2. *For $C = (\texttt{letrec}\ y = C', Env\ \texttt{in}\ t_{11}) \in \mathcal{PSFS}(t_1)$ with $t_1 = C[t_{12}]$ and $t'_1 := \mathrm{simp}(\texttt{letrec}\ y = C'[x], Env, x = t_{12}\ \texttt{in}\ x)$ we have $t'_1 \subseteq_\gamma t_2$.*

Then add a directed edge from L to N labeled with (subsume2).

generalizeFun *Let a leaf L be labeled with t, and let $t \equiv S[(x\ s)]$, where S is a surface context, and x is bound to Fun. Then apply the rule* generalize *such that $t' = S[\mathrm{top}]$, where* top *is an ac-variable bound to $\top$.*

generalize *Given a leaf L with label t, construct a new term t' as follows: add a binding* top $= \top$ *to the top* `letrec`*-environment, where* top *is a new ac-variable. Select a subterm of t on a surface position and in the upper part of t and replace this subterm by the variable* top*. Add an unlabeled directed edge from L to the node L' labelled with t'.*

Note that a subsume-edge may end in any node. It is not necessary that it is a predecessor of the leaf. Note also that in order to make $\subseteq_\gamma$ effective in the rule subsume, the criterion in Lemma 12.

The usual strategy for the construction of the directed graph is to apply the following rules ordered by their priority:

- subsume-rules.
- nred, i.e. sp-reduction.
- scsplit only if the splitted set constant is in an SF-context.
- generalizeFun if the generalized application is in an SF-context.
- The other rules.

SAL has two sources of non-determinism: The different possibilities for SF-contexts and the rule applications not in SF-contexts.

14.1 Correspondence between Concrete and Abstract Terms

Theorem 7. *Let t be a closed ac-labeled abstract term, such that $s \in \gamma(t)$ and $s{\Downarrow}$.*

1. *(*nred*) Let $t \xrightarrow{a} t'$ be an sp-reduction with $a \in \{$(case), (seq), (lbeta), (cp),(seq-Fun)$\}$.*
   Then there is a term $s' \in \gamma(t')$, such that $s \xrightarrow{a} s'$. If $a \in \{$(case), (seq), (lbeta),(seq-Fun)$\}$ then $\mathrm{rl}\sharp\sharp(s) > \mathrm{rl}\sharp\sharp(s')$ and if $a =$ (cp) then $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s')$.
2. *(*ired*) Let $t \xrightarrow{a} t'$, where $\xrightarrow{a}$ is a base-reduction, an extra reduction, or a simplification. Then there is some $s' \in \gamma(t')$ with $s \sim_c s'$ and $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s')$.*
3. *Let the rule (*scsplit*) be applied to the term t resulting in the sons $t_0, t_1, \ldots, t_k$. Then there exists a $j \in \{0, 1, \ldots, k\}$, and an $s' \in \gamma(t_j)$ such that $s \sim_c s'$, and $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s')$.*
4. *(*generalize*) If t' is the result of a generalization (*generalize *or* generalizeFun*) applied to t, then there is a concretization $s' \in \gamma(t')$ with $s' \sim_c s$ and $\mathrm{rl}\sharp\sharp(s) = \mathrm{rl}\sharp\sharp(s')$.*

Proof. 1. This follows from Proposition 11 and Lemma 10.
2. This follows from Lemma 10 and Proposition 10.
3. In the case of an scsplit, this follows from Lemma 11.
4. Let t' be the result of a generalization applied to t. Then a reverse (ucp) in an application surface is possible in s at the same position in the upper part, where the binding is placed in the ac-environment. Note that the replaced term has only free variables that are bound in the top `letrec`. Since $\top$ is maximal by Lemma 6 we obtain $s' \in \gamma(t')$. From Theorem 5 we derive $\mathrm{rl}\sharp\sharp(s) = \mathrm{rl}\sharp\sharp(s')$. □

Proposition 12. *Let (N_1, N_2) be an edge introduced by one of the subsume-rules. Let t_1 be the term at N_1 and t_2 be the term at N_2. Let $s_1 \in \gamma(t_1)$ with $s_1{\Downarrow}$. Then the following holds:*

1. *If the node is generated by the rule (*subsume*), then $s_1 \in \gamma(t_2)$.*

2. *If the node is generated by the rule (*subsume2*), then there is a concretization* $s_2 \in \gamma(t_2)$ *with* $s_2\Downarrow$ *and* $\mathrm{rl}\sharp\sharp(s_2) < \mathrm{rl}\sharp\sharp(s_1)$.

Proof. For the rule (subsume), this holds, since $t_1 \subseteq_\gamma t_2$.
For the rule (subsume2), we apply Proposition 8: The conditions in (subsume2), the definition of $\mathcal{PSFS}(\cdot)$ and the Definition 21 of strict and deep subterms imply that it is sufficient to consider the expression $t_1'' := (\texttt{letrec}\ y = C'[x], Env, x = t_{12}\ \texttt{in}\ x)$.
Using an appropriate renaming, from $s_1 = (\texttt{letrec}\ \ldots, C'[t_{12}]\ \texttt{in}\ t_{11})$ we construct the expression $s_1' = (\texttt{letrec}\ \ldots, C'[x], x = t_{12}\ \texttt{in}\ t_{11})$ with $\mathrm{rl}\sharp\sharp(s_1) = \mathrm{rl}\sharp\sharp(s_1')$. Proposition 8 implies that for $s_1'' = (\texttt{letrec}\ \ldots, C'[x], x = t_{12}\ \texttt{in}\ x)$, we have $\mathrm{rl}\sharp\sharp(s_1') > \mathrm{rl}\sharp\sharp(s_1'')$. Simplification does not change the measure $\mathrm{rl}\sharp\sharp(\cdot)$ by Theorem 7, hence with $s_2 = \mathrm{simp}(s_1'')$, we obtain $\mathrm{rl}\sharp\sharp(s_1) > \mathrm{rl}\sharp\sharp(s_2)$.

Corollary 2. *Let* N, N' *be two nodes in a graph generated by SAL, such that* (N, N') *is an edge. Let* t, t' *be the corresponding abstract terms. Let* $s \in \gamma(t)$ *be a terminating concretization.*

1. *If the edge is labeled, then there exists* $s' \in \gamma(t')$ *with* $\mathrm{rl}\sharp\sharp(s) > \mathrm{rl}\sharp\sharp(s')$.
2. *If the edge is not labeled and was generated using* nred, ired, generalizeFun *or* generalizeFun, *then there is a concretization* $s' \in \gamma(t')$ *with* $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s')$.
3. *Let the sons* $N_0, N_1 \ldots, N_k$ *be generated using* scsplit. *Then there is a* N_j *with term label* t_j, *and a concretization* $s' \in \gamma(t_j)$ *with* $s \sim_c s'$ *and* $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s')$.

15 Correctness of Strictness Detection

15.1 Main Theorems

Theorem 8. *Let* t *be a closed abstract term. If* t *leads to successful termination using SAL, then* $s \in \gamma(t) \Rightarrow s\Uparrow$, *i.e.* $s \sim_c \Omega$.

Proof. Assume that there is a closed concrete term $s \in \gamma(t)$ that has a terminating normal order reduction. Theorem 7 and Corollary 2 show that for every node N: if t_N at N has a concretization s_N with WHNF, then there is a direct successor node N' with term $t_{N'}$, $s_{N'} \in \gamma(t_{N'})$ and $\mathrm{rl}\sharp\sharp(s_N) \geq \mathrm{rl}\sharp\sharp(s_{N'})$. If the edge is labeled, then we have $\mathrm{rl}\sharp\sharp(s_N) > \mathrm{rl}\sharp\sharp(s_{N'})$ by Corollary 2 and Proposition 12. It is not possible that s has a successor in a leaf labeled $\bot$. Among the nodes that have a terminating concretization we select a node N_{min} with term label $t_{N,min}$ that has the minimal length $\mathrm{rl}\sharp\sharp(s_{N,min})$ of all terminating concretization $s_{N,min} \in \gamma(t_{N,min})$. Since $s_{N,min}\Downarrow$, there is an outgoing edge of N_{min} to a node $N_{min,2}$. Minimality shows that the corresponding edge cannot be labeled. The same holds for $N_{min,2}$, such that we find a path $N_{min}, N_{min,2}, N_{min,3}, \ldots$ of nodes connected with unlabeled edges. However, since the graph is finite, this enforces a cycle with unlabeled edges, which does not exist due to the construction. This is a contradiction, and we have thus shown that for all $s \in \gamma(t) : s\Uparrow$.

Corollary 3. *If* (`letrec` $bot = \bot$ `in` $f\ bot$) *leads to successful termination using SAL, then* f *is strict in its argument.*

Proof. The term $s := (\texttt{letrec}\ bot = \Omega\ \texttt{in}\ f\ bot)$ is a concretization of $t := (\texttt{letrec}\ bot = \bot\ \texttt{in}\ f\ bot)$. Theorem 8 implies that $(\texttt{letrec}\ bot = \Omega\ \texttt{in}\ f\ bot) \sim_c \Omega$. Proposition 10 and correctness of (ucp) and (gc) show that $\Omega \sim_c (\texttt{letrec}\ bot = \Omega\ \texttt{in}\ f\ bot) \sim_c f\ \Omega$, hence by the definition of strictness, f is strict in its argument.

Corollary 4. *If the term*

$$\begin{aligned} t := \texttt{letrec}\ & bot = \bot, top_1 = \top, \ldots, top_n = \top \\ & \texttt{in}\ (f\ top_1\ \ldots\ top_{i-1}\ bot\ top_{i+1}\ \ldots\ top_n) \end{aligned}$$

leads to successful termination using SAL, then f *is strict in its* i^{th} *argument.*

Proof. The definition of strictness requires that for every closed expression $t_j, j = 1, \ldots, n$: $f\ t_1\ \ldots t_{i-1}\ \Omega\ t_{i+1} \ldots t_n \sim_c \Omega$. We have:

$$\begin{aligned} & f\ t_1\ \ldots t_{i-1}\ \Omega\ t_{i+1} \ldots t_n \\ \sim_c\ & \texttt{letrec}\ x_1 = t_1, \ldots, x_{i-1} = t_{i-1}, x_i = \Omega, x_{i+1} = t_{i+1}, \ldots, x_n = t_n \\ & \quad \texttt{in}\ f\ x_1\ \ldots x_{i-1}\ x_i\ x_{i+1} \ldots x_n\ , \\ =:\ & s \end{aligned}$$

This follows using correctness of (lwas), (ucp), (gc) (see Theorem 2). We also have $s \in \gamma(t)$ by Lemma 6. Theorem 8 implies that $s \sim_c \Omega$. By the definition of strictness, we obtain that f is strict in its i^{th} argument.

Remark 4. A difference between Nöcker's subsumption rule and SAL is that in Nöcker's algorithm, it is always assumed that set constants are different, i.e. ac-variables are different. This in turn means that his subsumption rule appears to be stronger, since set constants in expressions can be subsumed, regardless of sharing. However, this can be easily simulated by SAL by applying the rule generalize for equal $\top$'s, and by adding a rule similar to generalize that makes all the occurrences of ac-variables distinct.
Interestingly, adding a linearized subsumption rule to SAL would make SAL incorrect, an example can easily be constructed on the basis of Example 5. Presumably, a linearized subsumption rule could also be used in SAL if the subsumption is only permitted for ancestors w.r.t. reduction. The argument for justifying this rule is that the graph can be further expanded such that the linearization will occur in a deeper part of the graph.

16 Examples

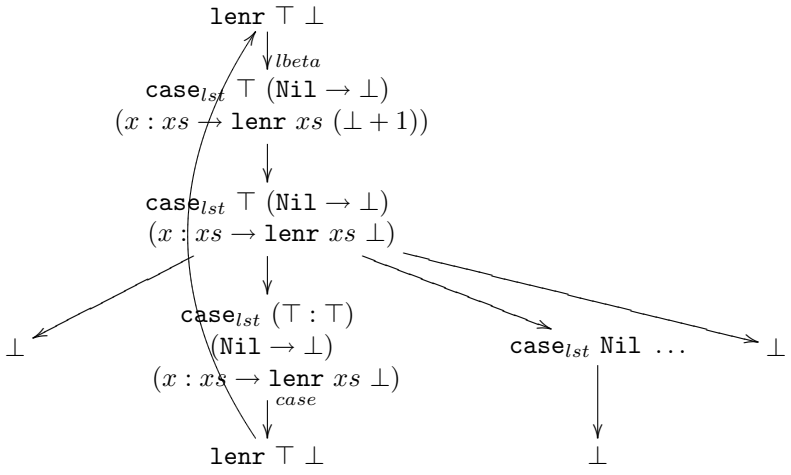
We demonstrate the algorithm SAL for some nontrivial functions and show the generated directed graphs. However, for readability, the graphs differ from the SAL-graphs in the following respects: The top letrec environment is not shown, instead the ac-variables are written using set-variables directly. Not all reductions are shown, e.g. sometimes (cp)-reductions are not shown. Case distinctions for $\top$ in the rule scsplit are only depicted for the list-constructors.

Example 2. We show that the tail-recursive length function (lenr) is strict in its second argument:

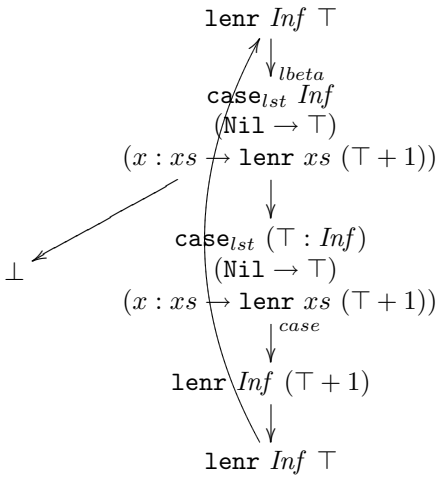
```
letrec len  = \lst   -> lenr lst 0,
       lenr = \lst s -> case lst of
              (Nil -> s)
              (x:xs ->  letrec z = 1+s in lenr xs z in  ... )
```

This can be shown by running SAL on the expression ($\texttt{letrec}\ top = \top, bot = \bot\ \texttt{in}\ (\texttt{lenr}\ top\ bot)$), including the definition of the functions in the environment.

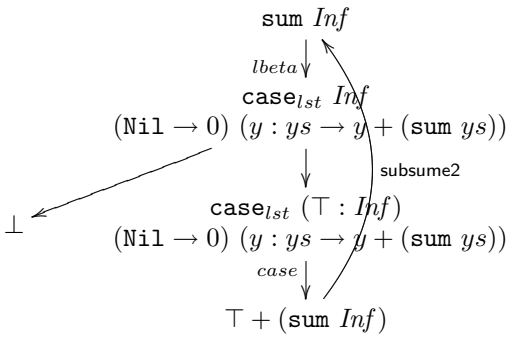
In the following diagram, we write $\bot$ for a variable that is bound to $\bot$, and *Inf* for a variable that is bound to *Inf*.



Example 3. We show that the function `lenr` is tail-strict in the first argument: For definition of *Inf* and the properties see the definition in subsection 12.1 and Lemma 6. Here we use the fact – which is, however, not proved in this paper – that a 2-ary function f is tail-strict in the first argument, if (f *Inf* $\top$) has no terminating concretization.



Example 4. We show that the function `sum` is tail-strict in its argument.
Let `sum` be defined in the environment by
$\texttt{sum} = \texttt{case}_{lst}\ xs\ (\texttt{Nil} \to 0)\ (y : ys \to y + (\texttt{sum}\ ys))$.
The resulting graph is:



Here the subterm criterion (subsume2) was used, and strictness of + in both arguments. Note that the simplification step in the subsume-rule is used, and that the two *Inf*-variables are different in SAL.

Example 5. We want to show that sharing of abstract constants is sometimes a (slight) improvement of SAL over Nöcker's method as described in [Nöc93].

```
f x z    =  g x x z
g x y z  = if x then if y then z
                         else False
               else if y then False
                         else z
```

Checking whether f is strict in its second argument means to check ($\texttt{letrec}\ top = \top, bot = \bot, \ldots\ \texttt{in}\ f\ top\ bot$)

Reducing this by (cp), (lbeta) yields (`letrec` $top = \top, bot = \bot, \ldots$ `in` $g\ top\ top\ bot$)
Now the effect is that the variable is the same, and $\top$ is not copied. The expression

```
if  top then if top then bot
             else False
        else if top then False
             else bot
```

yields for the `True` case:

```
if  True then if True then bot
              else False
         else if True then False
              else bot
```

which evaluates to $\bot$. The case `False` yields also $\bot$.
As published, Nöcker's method copies the $\top$ and the information that it is the same variable is lost. Perhaps the implementation is able to copy the top.

Example 6. Further set constants from Nöcker [N$\ddot{9}$0] can also be used in the analysis, slightly adapted:
`topmem` $= \{\bot\} \cup$ `Nil` $\cup (\top :$ `topmem`$)$
`botelem` $= \{\bot\} \cup (\top :$ `botelem`$) \cup (\bot :$ `topmem`$)$
As an example we show how SAL shows that `(reverse (concat botelem))` is non-terminating, which indicates that in the expression `(reverse (concat xs))`, the elements of the list `xs` can be evaluated first. The definitions are in a Haskell-like notation:

```
reverse xs   = rev  xs []
rev xs st    = case_list xs ([] -> st) (y:ys -> rev  ys (y:st))
concat xs    = foldr (++) [] xs
foldr f e xs = case_lst ([] -> e) (y:ys-> f y (foldr f e ys))
```

We assume that we already know that `++` is strict in its first argument. The presentation below will not show the `letrec`-structure, and the `++`-reductions will be done in one step. In step 2. there is a generalization, which has to be guessed. The standard strategy will not find a subsumption possibility, since at the stack position, the successive expressions are: $[], \top : [], \top : \top : []$. One possibility of a directed graph successfully generated by SAL in linear notation is as follows:

1. `reverse` (`concat botelem`)
2. `rev` (`concat botelem`) []
   Here we add a generalization for the stack
3. `rev` (`concat botelem`) $\top$
4. (`case`$_{lst}$ (`concat botelem`)([] $\rightarrow$ []) ($y : ys \rightarrow$ (`rev` ys ($y : \top$))))
5. (`case`$_{lst}$ (`foldr (++) [] botelem`) ([] $\rightarrow$ []) ($y : ys \rightarrow$ (`rev` ys ($y : \top$))))
6. (`case`$_{lst}$ (`case`$_{lst}$ `botelem`
   ([] $\rightarrow$ []) ($y : ys \rightarrow y$ `++` (`foldr (++) [] ys`)))
   ([] $\rightarrow$ []) ($y : ys \rightarrow$ (`rev` ys ($y : \top$))))

Now `botelem` will be splitted:

7.A.1 $(\texttt{case}_{lst}\ (\texttt{case}_{lst}\ \bot\ \ldots)\ \ldots)$

7.A.2 $\bot$

7.B.1 $(\texttt{case}_{lst}\ (\texttt{case}_{lst}\ (\top : \texttt{botelem})\ ([] \to [])\ \ldots)$

7.B.2 $(\texttt{case}_{lst}\ (\top\ \texttt{++}\ (\texttt{foldr}\ (\texttt{++})\ []\ \texttt{botelem}))$
$([] \to [])\ (y : ys \to (\texttt{rev}\ ys\ (y : \top))))$

$\top$ will be splitted into $\bot, [], \top : \top$ omitting untyped cases

7.B.2.A $(\texttt{case}_{lst}\ \bot\ \ldots) \to \bot$

7.B.2.B.1 $(\texttt{case}_{lst}\ (\texttt{foldr}\ (\texttt{++})\ []\ \texttt{botelem})$
$([] \to [])\ (y : ys \to (\texttt{rev}\ ys\ (y : \top))))$

7.B.2.B.2 Subsume-Link back to 4.

7.B.2.C.1 $(\texttt{case}_{lst}\ (\top : (\top\ \texttt{++}\ (\texttt{foldr}\ (\texttt{++})\ []\ \texttt{botelem})))$
$([] \to [])\ (y : ys \to (\texttt{rev}\ ys\ (y : \top))))$

7.B.2.C.2 $(\texttt{rev}\ (\top\ \texttt{++}\ (\texttt{foldr}\ (\texttt{++})\ []\ \texttt{botelem}))\ (\top : []))$

7.B.2.C.3 $(\texttt{case}_{lst}\ (\top\ \texttt{++}\ (\texttt{foldr}\ (\texttt{++})\ []\ \texttt{botelem}))$
$([] \to [])\ (y : ys \to (\texttt{rev}\ ys\ (y : \top : []))))$

7.B.2.C.4 $(\texttt{case}_{lst}\ (\top\ \texttt{++}\ (\texttt{foldr}\ (\texttt{++})\ []\ \texttt{botelem}))$
$([] \to [])\ (y : ys \to (\texttt{rev}\ ys\ (y : \top))))$

Subsume-Link to 7.B.2

7.C.1 $(\texttt{case}_{lst}\ (\texttt{case}_{lst}\ (\bot : \texttt{topmem})$
$([] \to [])\ (y : ys \to y\ \texttt{++}\ (\texttt{foldr}\ (\texttt{++})\ []\ ys)))$
$([] \to [])\ (y : ys \to (\texttt{rev}\ ys\ (y : \top))))$

7.C.2 $(\texttt{case}_{lst}\ (\bot\ \texttt{++}\ (\texttt{foldr}\ (\texttt{++})\ []\ \texttt{topmem}))$
$([] \to [])\ (y : ys \to (\texttt{rev}\ ys\ (y : \top))))$

7.C.3 $(\texttt{case}_{lst}\ \bot$ (since `++` is strict in its first argument)
$([] \to [])\ (y : ys \to (\texttt{rev}\ ys\ (y : \top))))$

7.C.4 $\bot$

17 Conclusion and Future Research

The paper gives a reconstruction of Nöcker's strictness analysis in a call-by-need lambda calculus using `letrec` and set constants. It provides a correctness proof for all the essential steps of the algorithm. The strictness analyzer based on abstract reduction was first described and implemented by Eric Nöcker in C and later by Marko Schütz in Haskell [Sch94]. It is based on a call-by-need calculus with sharing using `letrec`, set constants to represent sets and unions, and a contextual preorder.

Our proof is a modified version of the proof in [SSSS04], which used a non-deterministic lambda-calculus, where non-determinism was exploited for representing sets of expressions. However, the correctness proof in [SSSS04] requires the correctness of a conjecture on the relation between simulation and contextual preorder in this non-deterministic calculus. We believe that this conjecture holds, a justification for this belief is a proof in [Man04] that bisimulation implies

contextual preorder for a non-deterministic lambda-calculus, which however uses a nonrecursive `let` and is constructor-free. We are working on a proof of this conjecture.
A challenge for future research is to extend the results of the strictness analysis to FUNDIO [SS03], a call-by-need functional programming language with direct-call IO having an operational semantics is a combination of a trace- and a contextual semantics.

References

AH87. S. Abramsky and C. Hankin. *Abstract interpretation of declarative languages*. Ellis Horwood, 1987.

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

BHA85. G. L. Burn, C. L. Hankin, and S. Abramsky. The theory for strictness analysis for higher order functions. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Structures*, number 217 in Lecture Notes in Computer Science, pages 42–62. Springer, 1985.

BN98. Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

Bur91. Geoffrey Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Pitman, London, 1991.

CC77. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 252–252. ACM Press, 1977.

CDG02. M. Coppo, F. Damiani, and P. Giannini. Strictness, totality, and non-standard type inference. *Theoretical Computer Science*, 272(1-2):69–112, 2002.

CHH00. David Clark, Chris Hankin, and Sebastian Hunt. Safety of strictness analysis via term graph rewriting. In *SAS 2000*, pages 95–114, 2000.

GNN98. Kirsten Lackner Solberg Gasser, Hanne Riis Nielson, and Flemming Nielson. Strictness and totality analysis. *Sci. Comput. Program.*, 31(1):113–145, 1998.

Jen98. Thomas P. Jensen. Inference of polymorphic and conditional strictness properties. In *Symposium on Principles of Programming Languages*, pages 209–221, San Diego, January 1998. ACM Press.

Jon03. Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003. `www.haskell.org`.

KM89. Tsun-Ming Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. In *Functional Programming Languages and Computer Architecture*, pages 260–272. ACM Press, 1989.

LPJ95. John Launchbury and Simon Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(2):293–341, 1995.

Man04. Matthias Mann. Towards sharing in lazy computation systems. Frank report 18, Institut für Informatik, J.W.Goethe-Universität Frankfurt, Germany, 2004.

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

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 *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

MST96. Ian Mason, Scott F. Smith, and Carolyn L. Talcott. From operational semantics to domain theory. *Information and Computation*, 128:26–47, 1996.

Myc81. Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, University of Edinburgh, 1981.

N$\ddot{9}$0. Eric Nöcker. Strictness analysis using abstract reduction. Technical Report 90-14, Department of Computer Science, University of Nijmegen, 1990.

Nöc92. Eric Nöcker. Strictness analysis by abstract reduction in orthogonal term rewriting systems. Technical Report 92-31, University of Nijmegen, Department of Computer Science, 1992.

Nöc93. Eric Nöcker. Strictness analysis using abstract reduction. In *Functional Programming Languages and Computer Architecture*, pages 255–265. ACM Press, 1993.

NSvP91. E. Nöcker, J. E. Smetsers, M. van Eekelen, and M. J. Plasmeijer. Concurrent Clean. In *Proc of Parallel Architecture and Languages Europe (PARLE'91)*, number 505 in LNCS, pages 202–219. Springer Verlag, 1991.

Pap98. Dirk Pape. Higher order demand propagation. In K. Hammond, A.J.T. Davie, and C. Clack, editors, *Implementation of Functional Languages (IFL '98) London*, volume 1595 of *Lecture Notes in Computer Science*, pages 155–170. Springer-Verlag, 1998.

Pap00. Dirk Pape. *Striktheitsanalysen funktionaler Sprachen*. PhD thesis, Fachbereich Mathematik und Informatik, Freie Universität Berlin, 2000. in German.

Pat96. Ross Paterson. Compiling laziness using projections. In *Static Analysis Symposium*, volume 1145 of *LNCS*, pages 255–269, Aachen, Germany, September 1996. Springer.

PJS94. Simon L. Peyton Jones and André Santos. Compilation by transformation in the Glasgow Haskell Compiler. In *Functional Programming, Glasgow 1994*, Workshops in Computing, pages 184–204. Springer, 1994.

PvE03. R. Plasmeijer and M. van Eekelen. The concurrent Clean language report: Version 1.3 and 2.0. Technical report, Dept. of Computer Science, University of Nijmegen, 2003. `http://www.cs.kun.nl/~clean/`.

San95. Andre Santos. *Compilation by Transformation in non-strict functional Languages*. PhD thesis, University of Glasgow, 7 1995.

Sch94. Marko Schütz. Striktheits-Analyse mittels abstrakter Reduktion für den Sprachkern einer nicht-strikten funktionalen Programmiersprache. Diploma thesis, Johann Wolfgang Goethe-Universität, Frankfurt, 1994.

Sch00. Marko Schütz. *Analysing demand in nonstrict functional programming languages*. Dissertation, J.W.Goethe-Universität Frankfurt, 2000. available at http://www.ki.informatik.uni-frankfurt.de/papers/marko.

SS03. Manfred Schmidt-Schauß. FUNDIO: A lambda-calculus with a `letrec`, `case`, constructors, and an IO-interface: Approaching a theory of `unsafePerformIO`. Frank report 16, Institut für Informatik, J.W. Goethe-Universität Frankfurt am Main, 2003.

SSPS95. Manfred Schmidt-Schauß, Sven Eric Panitz, and Marko Schütz. Strictness analysis by abstract reduction using a tableau calculus. In *Proc. of the Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science, pages 348–365. Springer-Verlag, 1995.

SSSS04. Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. On the safety of Nöcker's strictness analysis. Technical Report Frank-19, Institut für Informatik. J.W.Goethe-University, 2004.

vEGHN93. M. van Eekelen, E. Goubault, C.L. Hankin, and E. Nöcker. Abstract reduction: Towards a theory via abstract interpretation. In M.R. Sleep, M.J. Plasmeijer, and M.C.J.D. van Eekelen, editors, *Term Graph Rewriting - Theory and Practice*, chapter 9. Wiley, Chichester, 1993.

Wad87. Phil Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12. Ellis Horwood Limited, Chichester, 1987.

WH87. Philip Wadler and John Hughes. Projections for strictness analysis. In *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 385–407. Springer, 1987.

A A Difference Between LR and an Untyped Core Language

Example 7. This example shows that the language LR is closer to a polymorphically typed language like Haskell and Clean than a functional core language without types. So assume for this example, that there is a core language LR_{ut} without types, but with `letrec`s, constructors, and abstractions. The language LR_{ut} has only one `case`-construct: there are alternatives for every constructor, and also either a default alternative, or an alternative for abstractions. Note that `seq` can be defined in LR_{ut}.
Now define the following functions:

$$s_0 = \lambda f.\texttt{if } (f\ \texttt{True})\ \texttt{then } (\texttt{if } (f\ \texttt{Nil})\ \texttt{then Bot else True})\ \texttt{else Bot}$$
$$t_0 = \lambda f.\texttt{if } (f\ \texttt{Nil})\ \texttt{then } (\texttt{if } (f\ \texttt{True})\ \texttt{then Bot else False})\ \texttt{else Bot}$$

We claim that s_0, t_0 cannot be distinguished in LR, since $(f\ \texttt{True})$ and $(f\ \texttt{Nil})$ can not result in different (terminating) Boolean values. If a function f outputs different values for the inputs `True` and `Nil`, then there must be an evaluation of a `case`-expression scrutinizing the inputs. But then one of the results must be $\bot$, since $\texttt{case}_T$ is typed. This reasoning can be extended to show that $s_0 \sim_c t_0$.
However, the expressions s_0, t_0 can be distinguished in LR_{ut}, since it is easy to define a function f as follows: The top level is a case having alternatives for all constructors, for `True` it yields `True`, and for `Nil` it yields `False`. Applying s_0, t_0 to the function f gives different results.
This means that the reason for the difference between the languages LR and LR_{ut} is only the restricted typing. The reason is that typing restricts the number of contexts in LR in contrast to LR_{ut}.

B Proof of the Context Lemma

In this section we prove the context lemma (Lemma 2). We will show the claim:

> Let s, t be terms. If for all reduction contexts R: $(R[s]{\Downarrow} \Rightarrow R[t]{\Downarrow})$, then $\forall C : (C[s]{\Downarrow} \Rightarrow C[t]{\Downarrow})$; i.e. $s \leq_c t$.

Proof. In this proof we will use multicontexts, which are generalizations of contexts having several holes, mentioning every occurrence of a hole in the argument list of the multicontext.
We prove the more general claim:
For $i = 1, \ldots, n$, let s_i, t_i be expressions. Let the following hold:
$\forall i : \forall$ reduction contexts R: $(R[s_i]{\Downarrow} \Rightarrow R[t_i]{\Downarrow})$.
Then $\forall C : C[s_1, \ldots, s_n]{\Downarrow} \Rightarrow C[t_1, \ldots, t_n]{\Downarrow}$.
Assume the claim is false. Then there is a counterexample. I.e., there is a multicontext C, a number $n \geq 1$ and terms s_i, t_i for $i = 1, \ldots, n$, such that $\forall i : \forall$ reduction contexts R: $(R[s_i]{\Downarrow} \Rightarrow R[t_i]{\Downarrow})$, and $C[s_1, \ldots, s_n]{\Downarrow}$, but $C[t_1, \ldots, t_n]{\Uparrow}$.
We select the counterexample minimal w.r.t. the following lexicographic ordering:

1. the number of normal order reduction steps of a shortest terminating normal order reduction of $C[s_1, \ldots, s_n]$.
2. the number of holes of C.

Either some hole of $C[\cdot_1, \ldots, \cdot_n]$ is in a reduction context or no hole is in a reduction context. The definition of reduction contexts and some easy reasoning shows that the unwind applied to $C[\cdot_1, \ldots, \cdot_n]$ either arrives at some hole, or does not arrive at a hole, and moreover, that this is not affected by filling the holes.

If one hole of $C[\cdot_1, \ldots, \cdot_n]$ is in a reduction context, then we assume wlog that it is the first one.
Then $C[\cdot, t_2 \ldots, t_n]$ is a reduction context. Let $C' := C[s_1, \cdot_2, \ldots, \cdot_n]$. Since $C'[s_2, \ldots, s_n] \equiv C[s_1, \ldots, s_n]$, these expressions have the same normal order reduction. Since the number of holes is smaller, we obtain $C'[t_2, \ldots, t_n]{\Downarrow}$, which means $C[s_1, t_2, \ldots, t_n]{\Downarrow}$. Since $C[\cdot, t_2, \ldots, t_n]$ is a reduction context, the preconditions of the lemma applied to s_1, t_1 imply $C[t_1, t_2, \ldots, t_n]{\Downarrow}$, a contradiction.

If no hole of $C[\cdot_1, \ldots, \cdot_n]$ is in a reduction context, then the first normal order reduction step $C[s_1, \ldots, s_n] \xrightarrow{n} C'[s'_1, \ldots, s'_m]$ can also be used for $C[t_1, \ldots, t_n]$ giving $C'[t'_1, \ldots, t'_m]$, where for every $i : \rho_{i,j}(s_j, t_j) = (s'_i, t'_i)$ for some variable renaming $\rho_{i,j}$ and some j. To verify this, we have to check that for a normal order redex, the parts that are modified are also in a reduction context.

- in a (cp) normal order reduction, every superterm of the to-variable position is in a reduction context.
- For normal order reductions (llet), (lapp), (lcase), (lseq), the inner `letrec` is in a reduction context.
- The constructor application used in a (case) is in a reduction context.

The following may happen to the terms s_i, t_i in the holes:

- If the hole is in an alternative of a (case)-expression that is discarded by the reduction, then the hole, and hence s_i as well as t_i, is eliminated after reduction.
- If the hole is not eliminated, and if the reduction is not a (cp), then the terms s_i, t_i in the holes are unchanged and also not copied, but both may appear at a different position in the resulting expression.
- If the reduction is a (cp), and the hole is not in the copied expression, then again the terms s_i, t_i in the holes are unchanged and also not copied.
- If the reduction is a (cp), and the hole is within the copied expression, then the terms s_i, t_i in the holes may be duplicated giving s'_i, t'_i. Since the reduction is a normal order reduction, and since we have assumed the "distinct bound variable convention", the renaming concerns the free variables in s_i, t_i which are bound in C. For a fixed i, we can use the same renaming ρ_i for the variables in s_i and t_i, so we have $\rho_i(s_i) = s'_i, \rho_i(t_i) = t'_i$. This means that the assumption holds also for the new pair of terms:

$$\forall i : \forall \text{ reduction contexts } R : (R[s'_i]{\Downarrow} \Rightarrow R[t'_i]{\Downarrow}).$$

Now we can use induction on the number of $\xrightarrow{n}$-reductions.
Since the number of steps in a shortest normal order reduction of $C[s'_1, \ldots, s'_m]$ is strictly smaller, we also have $C'[t'_1, \ldots, t'_m]{\Downarrow}$. But then we have also $C[t_1, \ldots, t_n]{\Downarrow}$, which contradicts the assumption that we have chosen a counterexample.
Now we look at the base case. If C has no holes, then a counterexample is impossible.
If the number of normal order reduction steps is 0, then $C[s_1, \ldots, s_n]$ is already a WHNF. Since we can assume that no hole is in a reduction context, the context is a WHNF, and thus this holds for $C[t_1, \ldots, t_n]$ as well, which is impossible.
Concluding, we have proved that there is no counterexample to the general claim, hence the lemma holds, since it is a specialization of this claim.

C Correctness of Reductions

In this section we prove that the reduction rules of the calculus (see Definition 8) and the extra reduction rules defined in Definition 15 are correct program transformations, i.e. maintain contextual equivalence.
Non-normal order reductions for the language LR are called *internal* and denoted by a label i. An internal reduction in a reduction context is marked by $i\mathcal{R}$, and an internal reduction in a surface context by $i\mathcal{S}$.

C.1 The reductions (case-c), (seq-c), (lbeta), (lapp), (lcase), (lseq)

Lemma 13. *Every a-reduction in a reduction context where $a \in \{$(case-c), (seq-c), (lbeta), (lapp), (lcase), (lseq)$\}$ is a normal order reduction.*

Proof. This follows by checking the possible term structures in a reduction context.

Proposition 13. *Contextual equivalence remains unchanged under the reductions (case-c), (seq-c), (lbeta), (lapp), (lcase), (lseq). I.e. $s \xrightarrow{a} t$ with $a \in \{$(case-c), (seq-c), (lbeta), (lapp), (lcase), (lseq)$\}$ implies $s \sim_c t$.*

Proof. This follows from the context lemma 2. It is sufficient to consider $R[s]$ and $R[t]$. From $s \xrightarrow{a} t$ and Lemma 13 it follows that $R[s] \xrightarrow{n} R[t]$.
Since normal order reduction is unique, it follows $R[s]{\Downarrow}$ iff $R[t]{\Downarrow}$. Now we apply the context lemma.

The reductions (lll), (cp), (case-e), (case-in), (seq-e), (seq-in) may be non-normal order in a reduction context, so other arguments are required.

C.2 Complete Sets of Commuting and Forking Diagrams

For proving correctness of further program transformations, we require the notions of complete sets of commuting diagrams and of complete sets of forking diagrams.

A *reduction sequence* is of the form $t_1 \to \ldots \to t_n$, where t_i are terms and $t_i \to t_{i+1}$ is a reduction as defined in definition 8. In the following definition we describe transformations on reduction sequences. Therefore we use the notation

$$\xrightarrow{iX,red} . \xrightarrow{n,a_1} \ldots \xrightarrow{n,a_k} \quad \rightsquigarrow \quad \xrightarrow{n,b_1} \ldots \xrightarrow{n,b_m} . \xrightarrow{iX,red_1} \ldots \xrightarrow{iX,red_h}$$

for transformations on reduction sequences. Here the notation $\xrightarrow{iX,red}$ means a reduction with $iX \in \{iC, i\mathcal{R}, i\mathcal{S}\}$, and *red* is a reduction from LR.
In order for the above transformation rule to be applied to the prefix of the reduction sequence RED, the prefix has to be $s \xrightarrow{iX,red} t_1 \xrightarrow{n,a_1} \ldots t_k \xrightarrow{n,a_k} t$. Since we will use sets of transformation rules, it may be the case that there is a transformation rule in the set, where the pattern matches a prefix, but it is not applicable, since the right hand side cannot be constructed.
We will say the transformation rule

$$\xrightarrow{iX,red} . \xrightarrow{n,a_1} \ldots \xrightarrow{n,a_k} \quad \rightsquigarrow \quad \xrightarrow{n,b_1} \ldots \xrightarrow{n,b_m} . \xrightarrow{iX,red_1} \ldots \xrightarrow{iX,red_h}$$

is *applicable* to the prefix $s \xrightarrow{iX,red} x_1 \xrightarrow{n,a_1} \ldots x_k \xrightarrow{n,a_k,} t$ of the reduction sequence RED iff the following holds:

$$\exists y_1, \ldots, y_m, z_1, \ldots, z_{h-1} :$$
$$s \xrightarrow{n,b_1} y_1 \ldots \xrightarrow{n,b_m} y_m \xrightarrow{iX,red_1} z_1 \ldots z_{h-1} \xrightarrow{iX,red_h} t$$

The transformation consists in replacing this prefix with the result:

$$s \xrightarrow{n,b_1} t'_1 \ldots t'_{m-1} \xrightarrow{n,b_m} t'_m \xrightarrow{iX,red_1} t''_1 \ldots t''_{h-1} \xrightarrow{iX,red_h} t$$

where the terms in between are appropriately constructed.

Definition 34.
• *A* complete set of commuting diagrams for the reduction $\xrightarrow{iX,red}$ *is a set of transformation rules on reduction sequences of the form*

$$\xrightarrow{iX,red} . \xrightarrow{n,a_1} \ldots \xrightarrow{n,a_k} \quad \rightsquigarrow \quad \xrightarrow{n,b_1} \ldots \xrightarrow{n,b_m} . \xrightarrow{iX,red_1} \ldots \xrightarrow{iX,red_{k'}},$$

where $k, k' \geq 0, m \geq 1$*, such that in every reduction sequence* $t_0 \xrightarrow{iX,red} t_1 \xrightarrow{n} \ldots \xrightarrow{n} t_h$*, where* t_h *is a WHNF, at least one of the transformation rules is applicable to a prefix of the sequence.*
In the special case $h = 1$*, we require that in* $t_0 \xrightarrow{iX,red} t_1$*, the term* t_1 *is a WHNF, and the term* t_0 *is not a WHNF.*

• *A* complete set of forking diagrams for the reduction $\xrightarrow{iX,red}$ *is a set of transformation rules on reduction sequences of the form*

$$\xleftarrow{n,a_1} \ldots \xleftarrow{n,a_k} . \xrightarrow{iX,red} \quad \rightsquigarrow \quad \xrightarrow{iX,red_1} \ldots \xrightarrow{iX,red_{k'}} . \xleftarrow{n,b_1} \ldots \xleftarrow{n,b_m},$$

where $k, k' \geq 0, m \geq 1$*, such that for every reduction sequence* $t_h \xleftarrow{n} \ldots t_2 \xleftarrow{n} t_1 \xrightarrow{iX,red} t_0$*, where* t_h *is a WHNF, at least one of the transformation rules from*

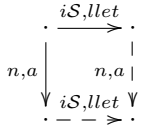
the set is applicable to a suffix of the sequence. In the special case that $h = 1$*, we require that in* $t_1 \xrightarrow{iX,red} t_0$*, the term* t_1 *is a WHNF, and that* t_0 *is not a WHNF.*

The two different kinds of diagrams are required for two different parts of the proof for the contextual equivalence of two terms.

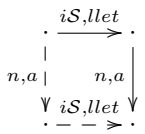
As a notation, we also use the * and +-notation of regular expressions for the diagrams. The interpretation is obvious and is intended to stand for an infinite set accordingly constructed.

In most of the cases, the same diagrams can be drawn for a complete set of commuting and forking diagrams, though the interpretation is different for commuting and forking diagrams. We will follow this in this paper and give in general only the drawing in the form of diagrams. The starting term is in the northwestern corner, and the normal order reduction sequences are always downwards. where the deviating reduction is pointing to the east. There are rare exceptions for degenerate diagrams, which are self explaining.

For example, the forking diagram $\xleftarrow{n,a} \cdot \xrightarrow{i\mathcal{S},llet} \rightsquigarrow \xrightarrow{i\mathcal{S},llet} \cdot \xleftarrow{n,a}$ is represented as

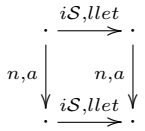


The commuting diagram $\xrightarrow{i\mathcal{S},llet} \cdot \xrightarrow{n,a} \rightsquigarrow \xrightarrow{n,a} \cdot \xrightarrow{i\mathcal{S},llet}$ is represented as



The straight arrows mean given reductions and dashed arrows mean existential arrows. A common representation is without the dashed arrows, where the interpretation depends on whether the diagrams is interpreted as forking or commuting diagram.

Note that the selection of the reduction label is considered to occur outside the transformation rule, i.e. if $\xrightarrow{n,a}$ occurs on both sides of the transformation rule the label a is considered to be the same on both sides.

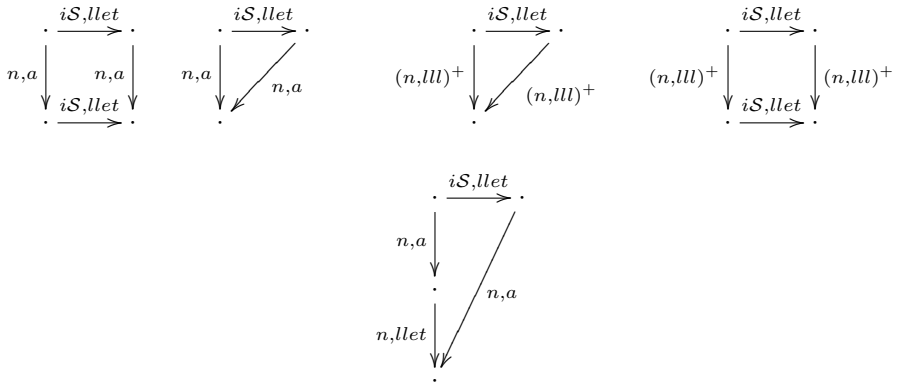


C.3 Diagrams for (llet), (seq) and (cp)

We prove equivalence of the reductions (llet), (seq) and (cp) by computing the required forking and commuting diagrams, and then give hints on an inductive proof for constructing normal order reductions.

C.3.1 Equivalence of (llet) For the reduction (llet), we use the reductions in $\mathcal{S}$-contexts instead of reduction contexts, since they are more general and cover all reduction contexts.

Lemma 14. *A complete set of forking diagrams and commuting diagrams for* $(i\mathcal{S}, llet)$ *can be read off the following graphical diagrams:*



The corresponding complete set of commuting diagrams is:

$$
\begin{aligned}
&\xrightarrow{i\mathcal{S},llet}\cdot\xrightarrow{n,a} \quad\rightsquigarrow\quad \xrightarrow{n,a}\cdot\xrightarrow{i\mathcal{S},llet}\\
&\xrightarrow{i\mathcal{S},llet}\cdot\xrightarrow{n,a} \quad\rightsquigarrow\quad \xrightarrow{n,a}\\
&\xrightarrow{i\mathcal{S},llet}\cdot\xrightarrow{(n,lll)^+} \quad\rightsquigarrow\quad \xrightarrow{(n,lll)^+}\\
&\xrightarrow{i\mathcal{S},llet}\cdot\xrightarrow{(n,lll)^+} \quad\rightsquigarrow\quad \xrightarrow{(n,lll)^+}\cdot\xrightarrow{i\mathcal{S},llet}\\
&\xrightarrow{i\mathcal{S},llet}\cdot\xrightarrow{n,a} \quad\rightsquigarrow\quad \xrightarrow{n,a}\cdot\xrightarrow{n,llet}
\end{aligned}
$$

The corresponding complete set of forking diagrams is:

$$
\begin{aligned}
&\xleftarrow{n,a}\cdot\xrightarrow{i\mathcal{S},llet} \quad\rightsquigarrow\quad \xrightarrow{i\mathcal{S},llet}\cdot\xleftarrow{n,a}\\
&\xleftarrow{n,a}\cdot\xrightarrow{i\mathcal{S},llet} \quad\rightsquigarrow\quad \xleftarrow{n,a}\\
&\xleftarrow{(n,lll)^+}\cdot\xrightarrow{i\mathcal{S},llet} \quad\rightsquigarrow\quad \xleftarrow{(n,lll)^+}\\
&\xleftarrow{(n,lll)^+}\cdot\xrightarrow{i\mathcal{S},llet} \quad\rightsquigarrow\quad \xrightarrow{i\mathcal{S},llet}\cdot\xleftarrow{(n,lll)^+}\\
&\xleftarrow{n,llet}\cdot\xleftarrow{n,a}\cdot\xrightarrow{i\mathcal{S},llet}\rightsquigarrow\ \xleftarrow{n,a}
\end{aligned}
$$

Proof. Diagram 1 covers the cases where the (iS, llet) and (n,a)-reductions commute. Diagram 2 covers the case of removed expressions in a (case)-reduction or a (seq)-reduction. Lemma 3 describes the same cases as diagrams 3 and 4.
Diagram 5 is the case where in diagram 1 the closing (llet) is turned into a normal order reduction. The typical case is (`letrec` x = (`letrec` *Env* `in` s) `in seq True` x).

Lemma 15. *If* $s \xrightarrow{i,lll} t$*, then* s *is a WHNF iff* t *is a WHNF.*

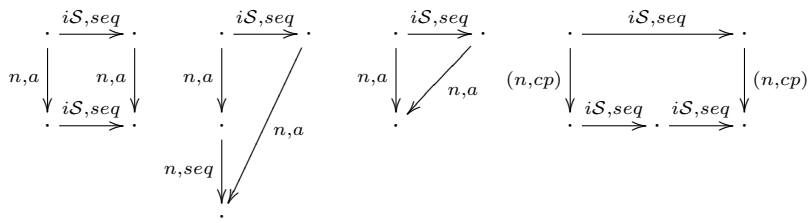
Proposition 14. *If* $s \xrightarrow{llet} t$*, then* $s \sim_c t$*.*

Proof. By the context lemma 2, it is sufficient to prove $R[s]{\Downarrow} \Leftrightarrow R[t]{\Downarrow}$ for all reduction contexts R. If $R[s] \xrightarrow{n,llet} R[t]$, then this is trivial. In the case $R[s] \xrightarrow{i\mathcal{S},llet} R[t]$, we use the complete sets of diagrams to show that from a terminating normal order reduction sequence of $R[s]$, we can construct a terminating normal order reduction of $R[t]$, and vice versa.

1. If $R[s]{\Downarrow}$, then by induction on the length of the normal order reduction sequence *Red* of $R[s]$, there is also a normal order reduction sequence for $R[t]$. We use the fact that of $s \xrightarrow{i\mathcal{S},llet} t$, then also $R[s] \xrightarrow{i\mathcal{S},llet} R[t]$, since reduction contexts are also surface contexts and the combination of surface contexts again gives a surface context.
   In the base case we use Lemma 15. If *Red* is not trivial, then the complete set of forking diagrams in lemma 14 provides all cases. Let $Red = R[s] \xrightarrow{n} s' \cdot Red'$. Diagrams 2,3,5 directly construct a terminating normal order reduction for $R[t]$. For diagrams 1 and 4, the induction hypothesis can be applied to $s' \xrightarrow{i\mathcal{S},llet} t'$ with $R[t] \xrightarrow{n,+} t'$, and we obtain a terminating normal order reduction for $R[t]$.
2. If $R[t]{\Downarrow}$, then we use similar methods. We apply induction on the number of normal order reduction steps of $R[t]$ to a WHNF using the complete set of commuting diagrams in lemma 14. In the base case we use Lemma 15. □

C.3.2 Equivalence of (seq) For the reduction (seq), we treat reductions in $\mathcal{S}$-contexts.

Lemma 16. *A complete set of forking diagrams and commuting diagrams for* $(i\mathcal{S}, seq)$ *can be read off the following graphical diagrams:*



Proof. Diagram 1 covers the cases where the $(i\mathcal{S}, seq)$ and (n,a)-reductions commute. Diagram 2 is the case where the closing $(i\mathcal{S}, seq)$ is turned into a normal order reduction. Diagram 3 covers the case where the redex is removed in a (case)-reduction or a (seq)-reduction. Diagram 4 covers the case where a $(i\mathcal{S}, seq)$ is applied within an abstraction that is copied by a (n, cp), e.g. in $(\texttt{letrec}\ y = \lambda z.z, y_1 = \lambda x.(\texttt{seq}\ y\ b)\ \texttt{in}\ y_1)$ there is a `seq`-reduction in a surface context, but the modified subexpression is within the body of an abstraction that will be copied in a normal order reduction.

Lemma 17. *If $s \xrightarrow{i\mathcal{S},seq} t$, then s is a WHNF iff t is a WHNF.*

Proposition 15. *If $s \xrightarrow{seq} t$, then $s \sim_c t$.*

Proof. It is sufficient to prove $R[s]{\Downarrow} \Leftrightarrow R[t]{\Downarrow}$ for all reduction contexts by the context lemma.
If $R[s] \xrightarrow{n,seq} R[t]$, then the claim is trivial. In the case $R[s] \xrightarrow{i\mathcal{S},seq} R[t]$, we use the diagrams.

1. If $R[s]{\Downarrow}$, then we have to use the forking diagrams in Lemma 16. We use the following measure for a normal order reduction *Red*: the pair $(\mu_1(Red), \mu_2(Red))$, where μ_1 is the number of (n, cp)-reductions, and μ_2 is the number of normal order reductions. The pairs are ordered lexicographically. We show by induction on $(\mu_1(Red), \mu_2(Red))$, that if $R[s]$ has a reduction *Red* to WHNF, then $R[t]$ has a reduction Red' with $\mu(Red) \geq \mu(Red')$. For diagram 1 we can apply the induction hypothesis using the number of normal order reductions. For the diagrams 2 and 3 this is obvious, and for diagram 4, we can apply the induction hypothesis twice. In the base case we use Lemma 17.
2. If $R[t]{\Downarrow}$, then the induction argument is slightly different. We consider the reduction *Red* which consists of $R[s] \xrightarrow{i\mathcal{S}} R[t] \xrightarrow{n,*} t_0$, where t_0 is a WHNF. The complete set of commuting diagrams in Lemma 16 is used as a transformation system on reductions, which consists of n- and $\xrightarrow{i\mathcal{S}}$ -reductions. The ordering on these mixed reductions is a multiset consisting of the following pairs of numbers: for every $\xrightarrow{i\mathcal{S}}$-reduction in the sequence, a pair (n_1, n_2), where n_1 is the number of (n,cp)-reductions to the right of it, and n_2 is the number of reductions to the right of it before the next (n,cp)-reduction. The pairs are ordered lexicographically, and the multiset is ordered by the induced multiset-ordering. By well-known arguments, this ordering is well-founded.
   Now we go through the diagrams:
   - Diagram 1 strictly decreases one pair: either the number of (n,cp)-reductions to the right is strictly decreased, or the second component of the pair is strictly decreased.
   - Diagram 2 removes one pair and does not change the other pairs.
   - Diagram 3 removes one pair, and does not change the other pairs.

– Diagram 4 replaces one pair by two pairs, which are strictly smaller, since the number of (n,cp)-reductions to the right is strictly smaller. Other pairs are either equal or strictly decreased.

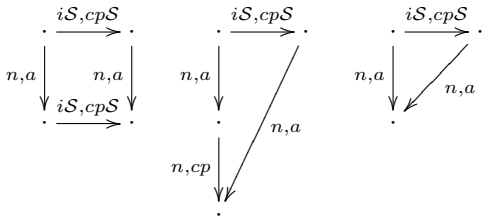
The base case is that the reduction $\xrightarrow{i\mathcal{S}}$ is the final one and results in a WHNF. In this case we use Lemma 17. and remove this reduction. Finally, we obtain a normal order reduction for $R[s]$.

C.3.3 Correctness of (cp) To show that the (cp)-reduction is correct as a program transformation, we have to split the reduction into two different reductions, depending on the position of the target variable.

(cpS) = (cp) where the position of the replaced variable is in a surface context.

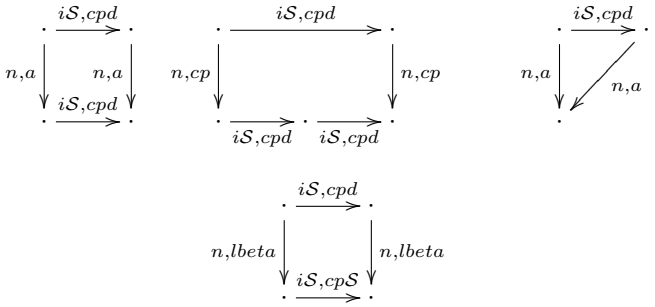
(cpd) = (cp) where the position of the replaced variable is not in a surface context.

Lemma 18. *A complete set of forking diagrams and commuting diagrams for* $\xrightarrow{i\mathcal{S},cp\mathcal{S}}$ *can be read off the following graphical diagrams:*



Proof. By case analysis. For a more detailed version see [SS03]

Lemma 19. *A complete set of forking diagrams and commuting diagrams for* $\xrightarrow{i\mathcal{S},cpd}$ *can be read off the following graphical diagrams:*



Proof. By case analysis. For a more detailed version see [SS03]

Lemma 20. *If* $s \xrightarrow{i\mathcal{S},cp} t$*, then* s *is a WHNF iff* t *is a WHNF.*

Proposition 16. *If* $s \xrightarrow{cp} t$*, then* $s \sim_c t$*.*

Proof. It is sufficient to prove $R[s]\Downarrow \Leftrightarrow R[t]\Downarrow$ for all reduction contexts. If $R[s] \xrightarrow{n,cp} R[t]$, then this is trivial. In the case $R[s] \xrightarrow{i\mathcal{S},cp} R[t]$, we use the diagrams for (cp), i.e., for (cpd) and (cpS).

1. Assume $R[t]\Downarrow$. The method is to transform the reduction $R[s] \xrightarrow{i\mathcal{S},cp} R[t] \cdot RED$, where RED is a normal order reduction to WHNF into a normal order reduction for $R[s]$ to a WHNF, where the transformations are used that correspond to the complete set of commuting diagrams in Lemmas 19 and 18.
   We have to show that the transformation terminates with a normal order reduction, where the local effect of the transformation is to shift $(i\mathcal{S}, cpd)$ and $(i\mathcal{S}, cp\mathcal{S})$ to the right. We define a well-founded measure for reduction sequences RED where $\xrightarrow{(i\mathcal{S},cpd)}$, $\xrightarrow{(i\mathcal{S},cp\mathcal{S})}$ and normal order reductions are mixed.
   A single $(i\mathcal{S}, cpd)$ or $(i\mathcal{S}, cp\mathcal{S})$ in RED has as measure the triple consisting of
   (a) the number of (n,lbeta)-reductions to the right of it;
   (b) the number of all (n, cp)- and $(i\mathcal{S}, cp\mathcal{S})$-reductions right of it before the next (n,lbeta)-reduction;
   (c) the number of reductions to the right.
   The triples are ordered lexicographically. The measure μ of the whole reduction sequence is the multiset of the triples for all $i\mathcal{S}$-reductions, ordered by the multiset-ordering. Every transformation rule of the commuting diagrams for $(i\mathcal{S}, cpd)$ and $(i\mathcal{S}, cp\mathcal{S})$ strictly decreases the measure μ. That the measure is decreased must also be checked for $(i\mathcal{S}, cpd)$ and $(i\mathcal{S}, cp\mathcal{S})$-reductions that are not directly involved in the transformation.

   (a) Diagram cpS-1: if $a =$ (lbeta), then it strictly decreases μ_1 for the involved reduction, and it does not increase the measure for other $(i\mathcal{S}, cpd)$ or $(i\mathcal{S}, cp\mathcal{S})$-reductions. If $a \neq$ (lbeta), then the μ_3-component of the involved reductions is strictly decreased.
   (b) Diagram cpS-2: One triple is removed, and the other triples are not increased.
   (c) Diagram cpS-3: One triple is removed, and the other triples are not increased.
   (d) Diagram cpd-1: Similar to diagram cpS-1.
   (e) Diagram cpd-2: one triple is replaced by two triples that have a strictly smaller μ_2-component, hence the multiset is strictly decreased.
   (f) Diagram cpd-3: see cpS-3.
   (g) Diagram cpd-4: a triple is replaced by a triple with strictly smaller μ_1-component.
   Since a diagram is applicable whenever there is a (cpd) or (cpS) reduction for a non-WHNF term, the transformation terminates with a normal order reduction.
   For the base case use Lemma 20.

2. If $R[s]\Downarrow$, then the transformation has to treat reductions sequences RED' that are mixtures of $\xrightarrow{(i\mathcal{S},cpd)}$, $\xrightarrow{(i\mathcal{S},cp\mathcal{S})}$ and normal order reductions $\xleftarrow{n,a}$, where the transformation has the local effect of shifting $\xrightarrow{(i\mathcal{S},cpd)}$ and $\xrightarrow{(i\mathcal{S},cp\mathcal{S})}$ to the left. We apply induction using the following measure:
   The well-founded measure for the mixed reduction sequences RED' is as follows:
   An $\xrightarrow{(i\mathcal{S},cpd)}$ or $\xrightarrow{(i\mathcal{S},cp\mathcal{S})}$ in RED' has as measure the triple consisting of
   (a) the number of $\xleftarrow{(n,lbeta)}$-reductions left of it;
   (b) the number of $\xleftarrow{(n,cp)}$- and $\xrightarrow{(i\mathcal{S},cp\mathcal{S})}$-reductions left of it before the next $\xleftarrow{(n,lbeta)}$-reduction.
   (c) the number of reductions to the left.
   The triples are ordered lexicographically. The measure μ of the whole reduction sequence is the multiset of the triples for all $i\mathcal{S}$-reductions, ordered by the multiset-ordering. A similar case analysis as above show that a normal order reduction for $R[t]$ can be constructed.

Now we can conclude by applying the context lemma for the two directions, that $s \sim_c t$.

C.3.4 Summary of the Properties The following proposition summarizes the results for the reduction (llet), (seq) and (cp).

Proposition 17. *The reductions (llet), (seq) and (cp) maintain contextual equivalence, i.e. if* $s \xrightarrow{a} t$, *with* $a \in \{(llet), (seq), (cp)\}$ *then* $s \sim_c t$.

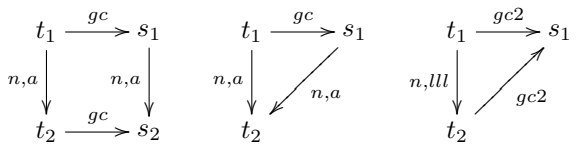
Proof. Follows from Propositions 14, 15 and 16.

C.4 Equivalence of (gc), (cpx), (xch), (abs) and (cpcx)

We show in this section that the extra reductions (gc), (cpx), (cpax), (cpcx), (xch), (ucp) and (lwas) are correct program transformations in the calculus LR. This finally will lead to a proof that (case) is a correct program transformation. In this section we extend the notion of complete sets of commuting and forking diagrams slightly by allowing the extra reductions as defined below in the place of the internal reductions.

C.4.1 Correctness of (gc)

Lemma 21. *A complete set of commuting and forking diagrams for* $(\mathcal{S}, gc)$ *can be read off the following set of graphical diagrams:*



Proof. This follows by a case analysis. Diagram 2 occurs if the (gc)-redex is in a removed alternative of a case or in the removed term in a (seq). Diagram 3 occurs, e.g. in the case (`seq` (`letrec` Env `in` t_1) t_2) if (gc) removes the environment Env, and in similar cases. A further example for the third case is

$$\begin{array}{ll} & R[((\texttt{letrec}\ Env_1\ \texttt{in}\ t_1)\ x)] \\ \xrightarrow{n,lapp} & R[(\texttt{letrec}\ Env_1\ \texttt{in}\ (t_1\ x))] \\ \hline \xrightarrow{gc} & R[(t_1\ x)] \end{array} \qquad \begin{array}{ll} & R[((\texttt{letrec}\ Env_1\ \texttt{in}\ t_1)\ x)] \\ \xrightarrow{gc} & R[(t_1\ x)] \end{array}$$

The following nontrivial overlapping results in a diagram of type 1.

$$\begin{array}{ll} & (\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ s)\ t) \\ \xrightarrow{gc} & ((\texttt{letrec}\ Env_2\ \texttt{in}\ s)\ t) \\ \xrightarrow{n,lapp} & (\texttt{letrec}\ Env_2\ \texttt{in}\ (s\ t)) \\ \hline \xrightarrow{n,lapp} & (\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ (s\ t))) \\ \xrightarrow{gc} & (\texttt{letrec}\ Env_2\ \texttt{in}\ s\ t) \end{array}$$

Lemma 22. *Let t, t' be expressions and $t \xrightarrow{gc} t'$. Then*

- *If t is a WHNF, then t' is a WHNF.*
- *If t' is a WHNF and t is not a WHNF, then $t \xrightarrow{n,llet} t'$; or $t \xrightarrow{[],n,llet} t'' \xrightarrow{gc2} t'$, and t'' is a WHNF.*

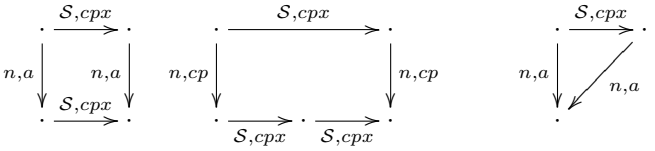
Proposition 18. *Let t be an expression. If $t \xrightarrow{gc} t'$, then $t \sim_c t'$.*

Proof. Using the context lemma and the same technique as in the proof of Proposition 16, we have only to ensure that transforming a terminating normal order reduction of $R[t]$ using the diagrams in Lemma 21 to a (terminating) normal order reduction of $R[t']$, and vice versa, always successfully terminates.
The measure for both directions is the number of normal order reductions, where the base case requires Lemma 22. In constructing from a reduction $R[t] \xrightarrow{gc} R[t'] \xrightarrow{n,*} t_0$ a terminating normal order reduction for $R[t']$, Proposition 2 shows that there there are only finitely many repeated applications of diagram 3.

C.4.2 Equivalence of (cpx) Note that the reduction $\xrightarrow{\mathcal{R},cpx}$ may not terminate:
`letrec` $x = y, y = x$ `in` $C[x] \xrightarrow{\mathcal{R},cpx}$ `letrec` $x = y, y = x$ `in` $C[y] \xrightarrow{\mathcal{R},cpx}$ `letrec` $x = y, y = x$ `in` $C[x]$.

A further example for non-termination is: `letrec` $x = y, y = x, z = x$ `in` $t \xrightarrow{\mathcal{R},cpx}$ `letrec` $x = y, y = x, z = y$ `in` $t \xrightarrow{\mathcal{R},cpx}$ `letrec` $x = y, y = x, z = x$ `in` t

Lemma 23. *A complete set of forking and commuting diagrams for* $\xrightarrow{\mathcal{S},cpx}$ *can be read off the following graphical diagrams:*



Proof. The second case happens if the target of the (cpx)-reduction is in the copied abstraction of the (cp). The third case may happen if the reduction is a (case), (cp) or (seq). An example for the last case is

$$\begin{array}{r} \texttt{letrec}\ x = s, y = x\ \texttt{in}\ C[y] \\ \xrightarrow{\mathcal{S},cpx} \texttt{letrec}\ x = s, y = x\ \texttt{in}\ C[x] \\ \xrightarrow{n,cp} \texttt{letrec}\ x = s, y = x\ \texttt{in}\ C[s] \\ \hline \xrightarrow{n,cp} \texttt{letrec}\ x = s, y = x\ \texttt{in}\ C[s] \end{array}$$

Lemma 24. *If* $s \xrightarrow{\mathcal{S},cpx} t$*, then* s *is a WHNF iff* t *is a WHNF.*

Proposition 19. *The reduction (cpx) is a correct program transformation.*

Proof. We only show the non-standard parts of the proof, which is termination of the transformation process. We use Lemmas 23 and 24.
There are two cases for the transformation. First consider the transformation of $s \xrightarrow{\mathcal{S},cpx} t \cdot RED$ into a normal order reduction sequence from s to WHNF, where RED is a normal order reduction to a WHNF. Intermediate steps have a sequence of normal order reductions mixed with (S,cpx)-reductions. We measure the sequences by the multiset consisting of the following numbers: for every (S,cpx)-reduction, the number of normal order reductions to the right of it. This is a well-founded order, and it is easy to see that the transformations strictly reduce this measure in every step using the commuting diagrams.
The other case is the transformation of $\overline{RED} \cdot s \xrightarrow{\mathcal{S},cpx} t$ to a normal order reduction of t, where RED is a normal order reduction sequence of s to WHNF, and $\overline{RED}$ the inverted sequence. Now the measure is the multiset consisting of the following numbers: for every (S,cpx)-reduction, the number of normal order reductions to the left of it. This is a well-founded order, and it is easy to see that the transformations strictly reduce this measure in every step using the forking diagrams.

C.4.3 Equivalence of the reduction rule (xch)

Lemma 25. *The reduction* $\xrightarrow{\mathcal{S},xch}$ *commutes with normal order reductions. I.e.*

$$\xrightarrow{\mathcal{S},xch} \cdot \xrightarrow{n,a} \quad \rightsquigarrow \quad \xrightarrow{n,a} \cdot \xrightarrow{\mathcal{S},xch}$$

Proof. It is easy to verify that this holds for the different kinds of reductions. Only for (case) and a specific type of interference we show the concrete transformation:

$$\begin{array}{ll} & (\texttt{letrec}\ x = c\ t, y = x\ \texttt{in}\ \texttt{case}_T\ x\ ((c\ u) \to r)) \\ \xrightarrow{xch} & (\texttt{letrec}\ y = c\ t, x = y\ \texttt{in}\ \texttt{case}_T\ x\ ((c\ u) \to r)) \\ \xrightarrow{n,case} & (\texttt{letrec}\ y = c\ z, z = t, x = y\ \texttt{in}\ (\texttt{letrec}\ u = z\ \texttt{in}\ r)) \\ \hline \xrightarrow{n,case} & (\texttt{letrec}\ x = c\ z, z = t, y = x\ \texttt{in}\ (\texttt{letrec}\ u = z\ \texttt{in}\ r)) \\ \xrightarrow{xch} & (\texttt{letrec}\ y = c\ z, z = t, x = y\ \texttt{in}\ (\texttt{letrec}\ u = z\ \texttt{in}\ r)) \end{array}$$

Lemma 26. *The $\xrightarrow{\mathcal{S},xch}$-reduction has trivial forking diagrams with normal order reductions. I.e.*

$$\xleftarrow{n,a} \cdot \xrightarrow{\mathcal{S},xch} \rightsquigarrow \xrightarrow{\mathcal{S},xch} \cdot \xleftarrow{n,a}$$

Lemma 27. *If $t \xrightarrow{xch} t'$, then t is a WHNF iff t' is a WHNF.*

Proposition 20. *The reduction (xch) is a correct program transformation.*

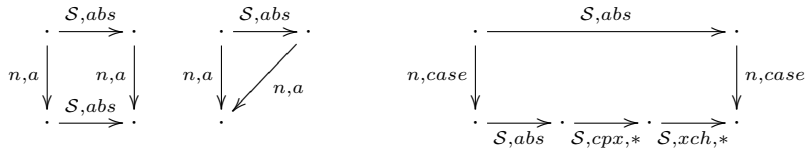
Proof. This follows using standard methods, since there are only the trivial diagrams.
It also follows from the reductions

$$\begin{array}{ll} & (\texttt{letrec}\ x = t, y = x, Env\ \texttt{in}\ r) \\ \xrightarrow{cpax} & (\texttt{letrec}\ x = t[x/y], y = x, Env[x/y]\ \texttt{in}\ r[x/y]) \\ \xrightarrow{gc} & (\texttt{letrec}\ x = t[x/y], Env[x/y]\ \texttt{in}\ r[x/y]) \\ =_\alpha & (\texttt{letrec}\ y = t[y/x], Env[y/x]\ \texttt{in}\ r[y/x]) \\ \xleftarrow{gc} & (\texttt{letrec}\ y = t[y/x], x = y, Env\ \texttt{in}\ r[y/x]) \\ \xleftarrow{cpax} & (\texttt{letrec}\ y = t, x = y, Env\ \texttt{in}\ r) \end{array}$$

and from the correctness of (gc) and (cpx); see Proposition 18, 19.

C.4.4 Equivalence of (abs)

Lemma 28. *A complete set of commuting and forking diagrams for $\xrightarrow{\mathcal{S},abs}$ can be read off the following diagrams:*



Proof. Instead of a complete proof, we only show the typical hard case:

$$
\begin{array}{ll}
 & (\texttt{letrec}\ x = c\ t\ \texttt{in}\ \texttt{case}_T\ x\ (c\ y \to s)) \\
\xrightarrow{abs} & (\texttt{letrec}\ x = c\ z, z = t\ \texttt{in}\ \texttt{case}_T\ x\ ((c\ y) \to s)) \\
\xrightarrow{n,case} & (\texttt{letrec}\ x = c\ u, u = z, z = t\ \texttt{in}\ (\texttt{letrec}\ y = u\ \texttt{in}\ s)) \\
\hline
\xrightarrow{n,case} & (\texttt{letrec}\ x = c\ u, u = t\ \texttt{in}\ (\texttt{letrec}\ y = u\ \texttt{in}\ s)) \\
\xrightarrow{abs} & (\texttt{letrec}\ x = c\ z, z = u, u = t\ \texttt{in}\ (\texttt{letrec}\ y = u\ \texttt{in}\ s)) \\
\xrightarrow{cpx,*} & (\texttt{letrec}\ x = c\ u, z = u, u = t\ \texttt{in}\ (\texttt{letrec}\ y = u\ \texttt{in}\ s)) \\
\xrightarrow{xch,*} & (\texttt{letrec}\ x = c\ u, u = z, z = t\ \texttt{in}\ (\texttt{letrec}\ y = u\ \texttt{in}\ s))
\end{array}
$$

The second diagram covers the case where the (abs)-redex is removed by a (case), or (seq).

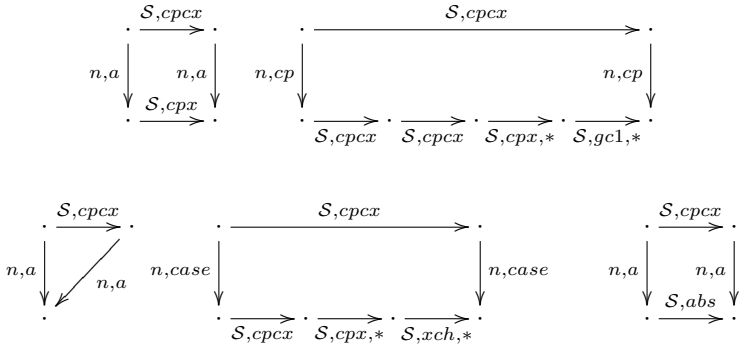
Lemma 29. *If $s \xrightarrow{\mathcal{S},abs} t$, then s is a WHNF iff t is a WHNF.*

Proposition 21. *The reduction (abs) is a correct program transformation.*

Proof. We only show the non-standard parts of the proof, which is termination of the transformation process.
There are two cases for the transformation.
First consider the transformation of $s \xrightarrow{\mathcal{S},abs} t \cdot RED$ into a normal order reduction sequence from s to WHNF, where RED is a normal order reduction to a WHNF. Intermediate steps have a sequence of normal order reductions mixed with (S,abs), (S,cpx), and (S,xch)-reductions. We measure the sequences by the multiset consisting of the following numbers: for every reduction (S,abs), (S,cpx), and (S,xch), the number of normal order reductions to the right of it. This is a well-founded order, and it is easy to see that the transformations strictly reduces this measure in every step using the commuting diagrams in Lemma 28, 23, and 25.
The other case is the transformation of $\overline{RED} \cdot s \xrightarrow{\mathcal{S},abs} t$ to a normal order reduction of t, where RED is a normal order reduction sequence of s to WHNF, and $\overline{RED}$ the inverted sequence. Now the measure of the mixed reduction sequence is the multiset consisting of the following numbers: for every reduction (S,abs), (S,cpx), and (S,xch), the number of normal order reductions to the left of it. This is a well-founded order, and it is easy to see that the transformations strictly reduce this measure in every step using the forking diagrams in Lemmas 28, 23, and 26.

C.4.5 Properties of (cpcx) Note that there are infinite reduction sequences using only (cpcx):
$(\texttt{letrec}\ x = c\ x\ \texttt{in}\ x) \xrightarrow{cpcx} (\texttt{letrec}\ x = c\ x_1, x_1 = x\ \texttt{in}\ c\ x_1) \xrightarrow{cpcx} (\texttt{letrec}\ x = c\ x_2, x_2 = x_1, x_1 = x\ \texttt{in}\ c\ (c\ x_2)) \ldots$

Lemma 30. *A complete set of commuting and forking diagrams for $\xrightarrow{\mathcal{S},cpcx}$ can be read off the following diagrams:*



where a in the last diagram may be (case) or (seq).

Proof. Instead of a complete proof, we only show the typical cases:

$$
\begin{array}{ll}
 & (\texttt{letrec}\ x = c\ t, y = \lambda u.C[x]\ \texttt{in}\ y) \\
\xrightarrow{\mathcal{S},cpcx} & (\texttt{letrec}\ x = c\ z, z = t, y = \lambda u.C[c\ z]\ \texttt{in}\ y) \\
\xrightarrow{n,cp} & (\texttt{letrec}\ x = c\ z, z = t, y = \lambda u.C[c\ z]\ \texttt{in}\ \lambda u'.C'[c\ z]) \\
\hline
\xrightarrow{n,cp} & (\texttt{letrec}\ x = c\ t, y = \lambda u.C[x]\ \texttt{in}\ \lambda u'.C'[x]) \\
\xrightarrow{cpcx} & (\texttt{letrec}\ x = c\ z, z = t, y = \lambda u.C[c\ z]\ \texttt{in}\ \lambda u'.C'[x]) \\
\xrightarrow{cpcx} & (\texttt{letrec}\ x = c\ z', z' = z, z = t, y = \lambda u.C[c\ z]\ \texttt{in}\ \lambda u'.C'[c\ z']) \\
\xrightarrow{cpx} & (\texttt{letrec}\ x = c\ z', z' = z, z = t, y = \lambda u.C[c\ z]\ \texttt{in}\ \lambda u.C[c\ z]) \\
\xrightarrow{cpx} & (\texttt{letrec}\ x = c\ z, z' = z, z = t, y = \lambda u.C[c\ z]\ \texttt{in}\ \lambda u.C[c\ z]) \\
\xrightarrow{gc} & (\texttt{letrec}\ x = c\ z, z = t, y = \lambda u.C[c\ z]\ \texttt{in}\ \lambda u.C[c\ z])
\end{array}
$$

$$
\begin{array}{ll}
 & (\texttt{letrec}\ x = c\ t\ \texttt{in}\ \texttt{case}_T\ x\ (c\ y \to s)) \\
\xrightarrow{cpcx} & (\texttt{letrec}\ x = c\ z, z = t\ \texttt{in}\ \texttt{case}_T\ (c\ z)\ ((c\ y) \to s)) \\
\xrightarrow{n,case} & (\texttt{letrec}\ x = c\ z, z = t\ \texttt{in}\ (\texttt{letrec}\ y = z\ \texttt{in}\ s)) \\
\hline
\xrightarrow{n,case} & (\texttt{letrec}\ x = c\ z, z = t\ \texttt{in}\ (\texttt{letrec}\ y = z\ \texttt{in}\ s))
\end{array}
$$

In the following example we use a multicontext $C[.,.]$ that may have different holes, every hole is mentioned as an argument.

$$
\begin{array}{ll}
 & (\texttt{letrec}\ x = c\ t\ \texttt{in}\ C[\texttt{case}_T\ x\ (c\ y \to s), x]) \\
\xrightarrow{cpcx} & (\texttt{letrec}\ x = c\ z, z = t\ \texttt{in}\ C[\texttt{case}_T\ x\ (c\ y \to s), c\ z]) \\
\xrightarrow{n,case} & (\texttt{letrec}\ x = c\ z', z' = z, z = t\ \texttt{in}\ C[(\texttt{letrec}\ y = z'\ \texttt{in}\ s), c\ z]) \\
\hline
\xrightarrow{n,case} & (\texttt{letrec}\ x = c\ z', z' = t\ \texttt{in}\ C[(\texttt{letrec}\ y = z'\ \texttt{in}\ s), x]) \\
\xrightarrow{cpcx} & (\texttt{letrec}\ x = c\ z, z = z', z' = t\ \texttt{in}\ C[(\texttt{letrec}\ y = z'\ \texttt{in}\ s), c\ z]) \\
\xrightarrow{cpx} & (\texttt{letrec}\ x = c\ z', z = z', z' = t\ \texttt{in}\ C[(\texttt{letrec}\ y = z'\ \texttt{in}\ s), c\ z]) \\
\xrightarrow{xch} & (\texttt{letrec}\ x = c\ z', z' = z, z = t\ \texttt{in}\ C[(\texttt{letrec}\ y = z'\ \texttt{in}\ s), c\ z])
\end{array}
$$

$$\begin{array}{ll} & (\texttt{letrec}\ x = c\ t\ \texttt{in}\ \texttt{seq}\ x\ r) \\ \xrightarrow{cpcx} & (\texttt{letrec}\ x = c\ z, z = t\ \texttt{in}\ \texttt{seq}\ (c\ z)\ r) \\ \xrightarrow{n,seq} & (\texttt{letrec}\ x = c\ z, z = t\ \texttt{in}\ r) \\ \hline \xrightarrow{n,seq} & (\texttt{letrec}\ x = c\ t\ \texttt{in}\ r) \\ \xrightarrow{abs} & (\texttt{letrec}\ x = c\ z, z = t\ \texttt{in}\ r) \end{array}$$

Lemma 31. *If $s \xrightarrow{\mathcal{S},cpcx} t$, then s is a WHNF iff t is a WHNF.*

Proposition 22. *The reduction (cpcx) is a correct program transformation.*

Proof. The non-standard part of the proof is the termination part.
First consider the transformation of $s \xrightarrow{\mathcal{S},cpcx} t \cdot RED$ to a normal order reduction of s to WHNF. We assume that always the rightmost S-reduction before a WHNF is transformed according to a diagram in the corresponding complete set. Intermediate reduction sequences consist of normal order reductions mixed with (S,cpcx)-, (S,cpx), (S,xch)
, (S,abs) and (S,gc)-reductions. The measure cannot be the number of normal order reductions, since (gc) is involved and the third diagram increases this number. We measure the reduction sequences by the multiset consisting of the following triples of numbers:
for every S-reduction the triple (n_1, n_2, n_3), where

1. n_1 is the number of normal order (case)- or (cp)-reductions to the right of it,
2. n_2 is the number of normal order reduction steps after the rightmost non-normal order reduction in the sequence.
3. n_3 is $\mu_{lll}(t')$, where $t' \xrightarrow{a} t''$ is the rightmost non-normal order reduction in the sequence.

The triples are compared lexicographically. This is a well-founded order on multisets. The commuting diagrams in Lemmas 30, 25, 23, 21, and 28 show that the transformations corresponding to (S,cpcx), (S,cpx), (S,abs), and (S,xch) strictly reduce this multiset-measure in every transformation step, if always the rightmost S-reduction before a WHNF is transformed. The third diagram in Lemma 21 leads to an increase of (lll)-reductions to the left of the (gc)-reduction, however, the component n_3 is strictly reduced, if this diagram is applied. If a WHNF is reached by the non-normal order reduction, then Lemmas 24, 22, 29, and 31 show that the non-normal order reduction can be shifted after a WHNF, hence it is removed.
The other case is the transformation of $\overline{RED} \cdot s \xrightarrow{\mathcal{S},cpcx} t$ to a normal order reduction of t. We measure the sequences by the multiset consisting of the number of normal order reductions to the left for every $\mathcal{S}, a$-reduction.
It is easy to see that the transformations strictly reduce this measure in every step using the forking diagrams for the reductions (S,cpcx), (S,cpx), (S,abs), and (S,xch).

C.4.6 Summary of the Properties

Theorem 9. *The reductions (cpcx), (cpx), (abs), (xch) and (gc) maintain contextual equivalence. I.e. whenever $t \xrightarrow{a} t'$, with $a \in \{cpcx, cpx, abs, gc\}$, then $t \sim_c t'$.*

Proof. Follows from Propositions 22, 19, 20, 21 and 18.

C.5 Correctness of (case)

Proposition 23. *The reductions (case-in) and (case-e) are correct program transformations.*

Proof. Proposition 13 shows that (case-c) is a correct program transformation. From Theorem 9 we obtain that (cpcx) and (cpx) are correct program transformations. We show by induction that a (case-e) and (case-in)-reduction is correct by using the correctness of the reductions (cpcx), (case-c) and (cpx) The induction is on the length of the variable chain in the (case-in) (or (case-e), respectively). We give the proof only for (case-in), the other is a copy of this proof. For the base case the (case-in) reduction can also be performed by the sequence of reductions: $\xrightarrow{cpcx}\xrightarrow{case-c}$

$$
\begin{array}{ll}
 & (\texttt{letrec}\ x = c\ t, Env\ \texttt{in}\ C[\texttt{case}_t\ x\ (c\ z \to s)\ alts]) \\
\xrightarrow{cpcx} & (\texttt{letrec}\ x = c\ y, y = t, Env\ \texttt{in}\ C[\texttt{case}_T\ (c\ y)\ (c\ z \to s)\ alts]) \\
\xrightarrow{case-c} & (\texttt{letrec}\ x = c\ y, y = t, Env\ \texttt{in}\ C[(\texttt{letrec}\ z = y\ \texttt{in}\ s)])
\end{array}
$$

For the induction we replace a (case-in) reduction operating on a chain $\{x_i = x_{i-1}\}_{i=2}^m$ with the sequence $\xrightarrow{cpcx}\xrightarrow{case-in}\xrightarrow{cpx^n}\xleftarrow{cpx^n}\xleftarrow{cpcx}$, where n is the arity of the constructor and the (case-in) reduction operates on the chain $\{x_i = x_{i-1}\}_{i=3}^m$:

$$
\begin{array}{ll}
 & (\texttt{letrec}\ x_1 = c\ \overrightarrow{t}, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ C[\texttt{case}_T\ x_m\ (c\ \overrightarrow{z} \to s)\ alts]) \\
\xrightarrow{cpcx} & \texttt{letrec}\ x_1 = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^n, x_2 = c\ \overrightarrow{y}, \{x_i = x_{i-1}\}_{i=3}^m, Env \\
 & \texttt{in}\ C[\_T\ x_1\ (c\ \overrightarrow{z} \to s)\ alts] \\
\xrightarrow{case-in} & \texttt{letrec}\ x_1 = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^n, x_2 = c\ \overrightarrow{y'}, \{y'_i = y_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=3}^m, Env \\
 & \texttt{in}\ C[(\texttt{letrec}\ \{z_i = y'_i\}_{i=1}^n\ \texttt{in}\ s)] \\
\xrightarrow{cpx^n} & \texttt{letrec}\ x_1 = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^n, x_2 = c\ \overrightarrow{y'}, \{y'_i = y_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=3}^m, Env \\
 & \texttt{in}\ C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^n\ \texttt{in}\ s)] \\
\xleftarrow{cpx^n} & \texttt{letrec}\ x_1 = c\ \overrightarrow{y'}, \{y_i = t_i\}_{i=1}^n, x_2 = c\ \overrightarrow{y'}, \{y'_i = y_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=3}^m, Env \\
 & \texttt{in}\ C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^n\ \texttt{in}\ s)] \\
\xleftarrow{cpcx} & \texttt{letrec}\ x_1 = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=2}^m, Env \\
 & \texttt{in}\ C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^n\ \texttt{in}\ s)]
\end{array}
$$

□

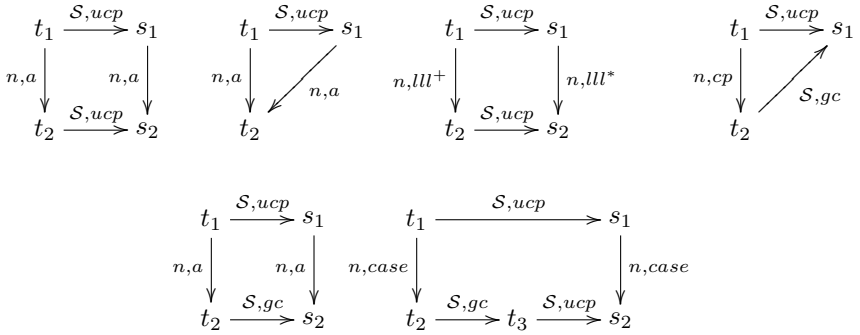
Proposition 24. *The reduction (case) is a correct program transformation.*

Proof. Follows from Proposition 23 and 13.

C.6 Correctness of (ucp), (abse), (cpax) and (lwas)

C.6.1 Correctness of (ucp) A difference between (ucp) and (cp) is that (ucp) can be applied even if the expression bound to a variable is not an abstraction.

Lemma 32. *The complete sets of forking and commuting diagrams for* $\xrightarrow{\mathcal{S},ucp}$ *can be read off the following graphical diagrams:*



Where $a \in \{(\text{seq})\}$ in the fifth diagram.

Proof. We show the typical overlappings.

$$\begin{array}{ll} & (\texttt{letrec}\ x = (\texttt{letrec}\ y = t\ \texttt{in}\ s), \mathit{Env}\ \texttt{in}\ (x\ z)) \\ \xrightarrow{ucp} & (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ ((\texttt{letrec}\ y = t\ \texttt{in}\ s)\ z)) \\ \xrightarrow{n,lapp} & (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (\texttt{letrec}\ y = t\ \texttt{in}\ (s\ z))) \\ \xrightarrow{n,llet} & (\texttt{letrec}\ \mathit{Env}, y = t\ \texttt{in}\ (s\ z)) \\ \hline \xrightarrow{n,llet} & (\texttt{letrec}\ x = s, y = t, \mathit{Env}\ \texttt{in}\ (x\ z)) \\ \xrightarrow{ucp} & (\texttt{letrec}\ y = t, \mathit{Env}\ \texttt{in}\ (s\ z)) \end{array}$$

$$\begin{array}{ll} & (\texttt{letrec}\ x = (\texttt{letrec}\ y = t_y\ \texttt{in}\ t_x), z = R'[x], \mathit{Env}\ \texttt{in}\ R[z]) \\ \xrightarrow{ucp} & (\texttt{letrec}\ z = R'[(\texttt{letrec}\ y = t_y\ \texttt{in}\ t_x)], \mathit{Env}\ \texttt{in}\ R[z]) \\ \xrightarrow{n,llet,+} & (\texttt{letrec}\ z = R'[t_x], y = t_y, \mathit{Env}\ \texttt{in}\ R[z]) \\ \hline \xrightarrow{n,llet,+} & (\texttt{letrec}\ x = t_x, y = t_y, z = R'[x], \mathit{Env}\ \texttt{in}\ R[z]) \\ \xrightarrow{ucp} & (\texttt{letrec}\ y = t_y, z = R'[t_x], \mathit{Env}\ \texttt{in}\ R[z]) \end{array}$$

$$\begin{array}{ll} & (\texttt{letrec}\ x = (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ v)\ \texttt{in}\ x) \\ \xrightarrow{ucp} & (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ v) \\ \hline \xrightarrow{n,llet} & (\texttt{letrec}\ x = v, \mathit{Env}\ \texttt{in}\ x) \\ \xrightarrow{ucp} & (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ v) \end{array}$$

$$\begin{array}{ll} & (\texttt{letrec}\ x = s, Env\ \texttt{in}\ (x\ y)) \\ \xrightarrow{ucp} & (\texttt{letrec}\ Env\ \texttt{in}\ (s\ y)) \\ \hline \xrightarrow{n,cp} & (\texttt{letrec}\ x = s, Env\ \texttt{in}\ (s\ y)) \\ \xrightarrow{gc} & (\texttt{letrec}\ Env\ \texttt{in}\ (s\ y)) \end{array}$$

$$\begin{array}{ll} & (\texttt{letrec}\ x = cs\ \texttt{in}\ (\texttt{seq}\ x\ r)) \\ \xrightarrow{ucp} & (\texttt{seq}\ (c\ s)\ r) \\ \xrightarrow{n,seq} & r \\ \hline \xrightarrow{n,seq} & (\texttt{letrec}\ x = cs\ \texttt{in}\ r) \\ \xrightarrow{gc} & r \end{array}$$

$$\begin{array}{ll} & (\texttt{letrec}\ x = c\ s\ \texttt{in}\ (\texttt{case}_T\ x\ (c\ z \to r))) \\ \xrightarrow{ucp} & (\texttt{case}_T\ (c\ s)\ (c\ z \to r)) \\ \xrightarrow{n,case} & (\texttt{letrec}\ z = s\ \texttt{in}\ r) \\ \hline \xrightarrow{n,case} & (\texttt{letrec}\ x = c\ y, y = s\ \texttt{in}\ (\texttt{letrec}\ z = y\ \texttt{in}\ r)) \\ \xrightarrow{gc} & (\texttt{letrec}\ y = s\ \texttt{in}\ (\texttt{letrec}\ z = y\ \texttt{in}\ r)) \\ \xrightarrow{ucp} & (\texttt{letrec}\ z = s\ \texttt{in}\ r) \end{array}$$

Lemma 33. *Let t, t' be expressions and $t \xrightarrow{ucp} t'$. Then*

- *If t is a WHNF, then t' is a WHNF.*
- *If t' is a WHNF, then there are the following cases:*
  - *t is a WHNF*
  - *$t \xrightarrow{n,(lll \cup\ cp),*} t''$, where t'' is a WHNF*

Proof. The first case is obvious.
In the second case there are the following possibilities:

- $t = (\texttt{letrec}\ x = \lambda v.r, Env\ \texttt{in}\ x) \xrightarrow{ucp} (\texttt{letrec}\ Env\ \texttt{in}\ \lambda v.r)$. In this case a (n,cp)-reduction is sufficient to transform t into WHNF.
- $t = (\texttt{letrec}\ x = t_0\ \texttt{in}\ x) \xrightarrow{ucp} t_0$, where t_0 is a WHNF. Then either an (n,cp)-reduction, or a (n,llet)-reduction, or a (n, llet) followed by an (n,cp)-reduction transform t into WHNF.
- $t = (\texttt{letrec}\ x = v\ \texttt{in}\ (\texttt{letrec}\ Env\ \texttt{in}\ x)) \xrightarrow{ucp} (\texttt{letrec}\ Env\ \texttt{in}\ v)$, where v is a value. Then an (n,cp)-reduction or an (n,llet)-reduction or an (n,llet)-reduction followed by a (n,cp)-reduction transform t into WHNF.
- $t = (\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{letrec}\ x = v\ \texttt{in}\ x)) \xrightarrow{ucp} (\texttt{letrec}\ Env\ \texttt{in}\ v)$. Again an (n, llet)-reduction or an (n, llet)-reduction followed by an (n,cp)-reduction transforms t into WHNF.

Proposition 25. *Let t be a expression. If $t \xrightarrow{ucp} t'$, then $t \sim_c t'$*

Proof. Using the context lemma and the same technique as in the proofs of Proposition 16, we have only to ensure that transferring a terminating normal order reduction of $R[t]$ to a normal order reduction of $R[t']$, and vice versa, really terminates.

- Let $R[t]\Downarrow$. We show that $R[t']\Downarrow$ by using the forking diagrams in Lemma 32 and 21 to transform $\overline{Red}\cdot \xrightarrow{,\mathcal{S},ucp}$ into a terminating normal order reduction of $R[t']$, where *Red* is a terminating normal order reduction of $R[t]$. We use as measure on the intermediate reductions a multiset of the following numbers: for every $\mathcal{S}$-reduction, the number of normal order reductions to the left of it.
  The diagrams 1 – 5 of ucp obviously strictly decrease this measure. Diagram 6 replaces a number in the multiset by two smaller ones, hence the multiset is also strictly decreased.
  For the base case, we apply Lemma 33.
- Let $R[t']\Downarrow$. We show that $R[t]\Downarrow$ by transforming a terminating reduction $\xrightarrow{\mathcal{S},ucp} \cdot Red$ into a normal order reduction of $R[t]$, where *Red* is a normal order reduction of $R[t']$. We use the commuting diagrams in Lemma 32 and 21 for the transformation. First we shift the single $\xrightarrow{\mathcal{S},ucp}$-reduction to the right using the diagrams in Lemma 32. This terminates, since the number of normal order reductions to the right either strictly decreases or in the case of diagram 3, some (lll)-reductions are added to the left, which also terminates. In the final step, we apply Lemma 33 and replace the final reduction by $\xrightarrow{n,(lll\cup\ cp),*}$. This leaves a reduction sequence which is a mixture of normal order reductions and $\xrightarrow{\mathcal{S},gc}$-reductions. Now we can apply Lemma 21 and can use the same arguments as in Proposition 18 to transform this mixed sequence into a normal order reduction to a WHNF.

C.6.2 Correctness of (abse)

Proposition 26. *The reduction (abse) is a correct program transformation.*

Proof. This follows from Proposition 25 and Theorem 9, since (abse) can be undone by several (ucp)-and (gc)-reductions.

C.6.3 Correctness of (cpax)

Proposition 27. *The reduction (cpax) is a correct program transformation.*

Proof. This follows from proposition 19, since (cpax) can also be performed by several (cpx) reductions.

Proposition 28. *The reduction (cpax) is terminating.*

Proof. Every (cpax) reduction strictly decreases the number of let-bound variables that have occurrences in the expression.

C.6.4 Correctness of (lwas) In this subsection we show the correctness of the reduction *(lwas)*, which lifts letrec bindings over an $\mathcal{AS}^{-}_{(1)}$ context.

Proposition 29. *The (lwas)-reduction is correct. I.e., if* $s \xrightarrow{lwas} t$, *then* $s \sim_c t$.

Proof. The reduction sequence

$$\begin{array}{ll} & AS^{-}_{(1)}[(\texttt{letrec}\ Env\ \texttt{in}\ t)] \\ \xleftarrow{ucp} & (\texttt{letrec}\ x = (\texttt{letrec}\ Env\ \texttt{in}\ t)\ \texttt{in}\ AS^{-}_{(1)}[x]) \\ \xrightarrow{llet} & (\texttt{letrec}\ x = t, Env\ \texttt{in}\ AS^{-}_{(1)}[x]) \\ \xrightarrow{ucp} & (\texttt{letrec}\ Env\ \texttt{in}\ AS^{-}_{(1)}[t]) \end{array}$$

and the correctness of (ucp) and (lll), which are proved in Proposition 17, and Proposition 25 show that the proposition holds.

C.7 Equivalence of the variants of (case)-reductions

Lemma 34. *The following holds:*

1. *The reduction rule (cpcxnoa) is a correct program transformation. It can be simulated by (cpcx) with a subsequent* $\xrightarrow{cpx,*} \cdot \xrightarrow{gc,*}$ *reduction.*
2. *A (case-cx) reduction can be simulated by (case) and subsequent* $\xrightarrow{cpx,*} \cdot \xrightarrow{gc,*}$ *reduction.*
3. *Every (case-e) and (case-in)-reduction can be simulated by an (abs)-reduction followed by a (case-cx)-reduction.*
4. *The reduction rule (case-cx) is a correct program transformation.*

Proof. Easy.

Lemma 35. *Let* $t \xrightarrow{abs} t'$, $t{\Downarrow}$, *and Red be the normal order reduction of* t *to a WHNF. Then* $t'{\Downarrow}$, *and* t' *has a normal order reduction* Red', *where the type of the base reductions in Red and* Red' *are in the same sequence.*

Proof. This follows from the diagrams in Lemmas 43 and 41 by induction, shifting (abs), (xch) and (cpx) to the WHNF.

C.8 Proofs of Theorem 1 and 2

First we prove Theorem 1, where the claim is:

> All the reductions in the base calculus maintain contextual equivalence.
> I.e. whenever $t \xrightarrow{a} t'$, with $a \in \{\text{cp, lll, case, seq, lbeta}\}$, then $t \sim_c t'$.

Proof. Follows from Propositions 17, 24 and 13.

Finally we prove the claim of Theorem 2:

The reductions (ucp), (cpx), (cpax), (gc), (lwas), (cpcx), (abs), (abse) (xch), (cpcxnoa) and (case-cx) maintain contextual equivalence. I.e. whenever $t \xrightarrow{a} t'$, with $a \in \{\text{ucp}, \text{cpx}, \text{cpax}, \text{gc}, \text{lwas}, \text{cpcx}, \text{abs}, \text{abse}, \text{xch}, \text{cpcxnoa}, \text{case-cx}\}$, then $t \sim_c t'$.

Proof. Follows from Theorem 9, Propositions 25, 27, 29, 26 and 20 and Lemma 34.

D Properties of Bot

In this section we show that all bot-terms, i.e., all terms t with $t{\Uparrow}{\Uparrow}$ are equivalent and that Ω is the least element w.r.t. $\leq_c$.
The following proposition shows that Ω is the least element w.r.t. $\leq_c$.

Proposition 30. *Let t be an expression such that $t{\Uparrow}{\Uparrow}$ and let s be an arbitrary expression.*
Then $t \leq_c s$.

Proof. The context lemma shows that it is sufficient to prove for all reduction contexts R: $R[t]{\Downarrow} \Rightarrow R[s]{\Downarrow}$. We simply prove that $R[t]{\Downarrow}$ does not hold. Assume that there is a terminating normal order reduction of $R[t]$ to WHNF. We prove by induction that this implies that t has a terminating normal order reduction to a WHNF.
Let $t \xrightarrow{n,*} t_1$, such that t_1 is the first `letrec`-expression in the sequence. If $t \neq t_1$, we can use induction, since the normal order reductions of $R[t] \xrightarrow{n,*} R[t_1]$ are precisely the same reductions. This holds, since inserting the maximal weak reduction context of t into a reduction context R yields a maximal reduction context.
In the rest of the proof we assume that t is a `letrec`-expression.
By Lemma 3, if $t = (\texttt{letrec}\ E_t\ \texttt{in}\ t')$, the normal order reduction reduces $R[t] = R[(\texttt{letrec}\ E_t\ \texttt{in}\ t')]$ to $(\texttt{letrec}\ E_t, E_R\ \texttt{in}\ R'[t'])$ in several steps, where R' is a weak reduction context, E_t the environment that belongs to t, and E_R the environment part that is at the top level of R. If R is not a `letrec`-expression, then E_R is empty.
The correspondence between normal order reductions of t and of $(\texttt{letrec}\ E_t, E_R\ \texttt{in}\ R'[t'])$ is as follows:

- If there is a (n,llet-in)-reduction $t = (\texttt{letrec}\ E_t\ \texttt{in}\ (\texttt{letrec}\ E_1\ \texttt{in}\ t'')) \xrightarrow{n} (\texttt{letrec}\ E_t, E_1\ \texttt{in}\ t'')$, then the corresponding normal order reduction of $(\texttt{letrec}\ E_t, E_R\ \texttt{in}\ R'[(\texttt{letrec}\ E_1\ \texttt{in}\ t'')])$ is a normal order (mll)-reduction resulting in $(\texttt{letrec}\ E_t, E_1, E_R\ \texttt{in}\ R'[t''])$. With $E'_t = E_t \cup E_1$, the correspondence holds.
- If there is another reduction of t, then this is of the form $(\texttt{letrec}\ E_t\ \texttt{in}\ t') \xrightarrow{n} (\texttt{letrec}\ E'_t\ \texttt{in}\ t'')$. It is easy to see that $(\texttt{letrec}\ E_t, E_R\ \texttt{in}\ R'[t']) \xrightarrow{n} (\texttt{letrec}\ E'_t, E_R\ \texttt{in}\ R'[t''])$. The environment E_R is never involved, since we have assumed that $t{\Uparrow}{\Uparrow}$.

Summarizing, the normal order reductions of $R[t]$ correspond to the number of normal order reductions of t. The number of (lll)-reductions may vary, but the non-(lll)-reductions are the same. Hence, if $R[t]$ terminates with a WHNF, then we also obtain a WHNF of t by the translation above. Hence we get a contradiction.
This finally shows that for a non-terminating t, the term $R[t]$ cannot have a terminating normal order reduction.

Corollary 5. *1. If t_1, t_2 are expressions with $t_1{\Uparrow}{\Uparrow}$ and $t_2{\Uparrow}{\Uparrow}$, then $\Omega \sim_c t_1 \sim_c t_2$.*
2. For all expressions s: $\Omega \leq_c s$.
3. $R[\Omega] \sim_c \Omega$.
4. If $t = R[s]$ is an expression and R a reduction context, then s is a strict subexpression of t.

Proof. The first two claims follow from Proposition 30. Claim 3 and 4 follow using the arguments in the proof of Proposition 30.

D.1 Reduction Rules for Bot-terms

Definition 35. *The reduction rules that treat the bot-term Ω are defined in figure 5. Note that these reductions are permitted in all contexts.*

Proposition 31. *If $t \to t'$ by a bot-reduction as defined in Figure 5, then*

- *$t \sim_c t'$.*
- *If t is a closed concrete term with $t{\Downarrow}$, then $t'{\Downarrow}$ and $\mathrm{rl}\sharp\sharp(t) = \mathrm{rl}\sharp\sharp(t')$.*

Proof. Contextual equivalence follows from Corollary 5 for (beta-bot), (case-bot), (app-bot) (letrec-bot), (case-untyped), (seq) and (strict-bot). For the rules (cp-in-bot), (cp-e-bot), and (hole) other arguments are required. The context lemma shows the claim: If $t \to t'$, and we check the normal order reductions of $R[t]$ and $R[t']$, then they are synchronous, as long as neither Ω nor x is required. The values of x or Ω are required in t iff they are required in t'. In this case this term is in a reduction context. We already know that then the expression is contextually equivalent to Ω. Thus the context lemma 2 shows contextual equivalence. Finally, the correctness of (merge-bot) follows from 2, since (merge-bot) can be simulated by performing some (cpx) and a (cp-e-bot) reduction.
The claim on the lengths of reductions can be proved as follows. In a terminating normal order reduction to WHNF, the subterm Ω, the subterm x, or the untyped expressions cannot be required by evaluation in any reduction, hence the lengths of the normal order reduction sequences are the same.

(beta-bot)	$(\Omega\ y) \to \Omega$
(cp-in-bot)	$(\texttt{letrec}\ x = \Omega, Env\ \ \texttt{in}\ C[x]) \to (\texttt{letrec}\ x = \Omega, Env\ \ \texttt{in}\ C[\Omega])$
(cp-e-bot)	$(\texttt{letrec}\ x = \Omega, y = C[x], Env\ \ \texttt{in}\ r) \to (\texttt{letrec}\ x = \Omega, y = C[\Omega], Env\ \ \texttt{in}\ r)$
(hole)	$(\texttt{letrec}\ x = x, Env\ \ \texttt{in}\ r) \to (\texttt{letrec}\ x = \Omega, Env\ \ \texttt{in}\ r)$
(case-bot)	$(\texttt{case}_T\ \Omega\ \ldots((c_i\ y_1 \ldots y_n)\ \texttt{->}\ t)\ldots) \to \Omega$
(app-bot)	$(v\ t) \to \Omega$ if v is a constructor application
(letrec-bot)	$(\texttt{letrec}\ Env\ \texttt{in}\ \Omega) \to \Omega$
(case-untyped1)	$(\texttt{case}_T\ v\ alts) \to \Omega$ if v is an abstraction or the top-constructor does not belong to the type T
(case-untyped2)	$(\texttt{letrec}\ x_n = v, \ldots, x_1 = x_2, Env\ \texttt{in}\ \texttt{case}_T\ x_1\ alts) \to \Omega$ if v is an abstraction or its top-constructor does not belong to the type T
(seq-bot)	$(\texttt{seq}\ \Omega\ t) \to \Omega$
(strict-bot)	$D[(f\ x_1 \ldots x_n)] \to D[\Omega]$ if f is strict in its i^{th} argument for arity n and x_i is bound to Ω in D
(merge-bot)	$(\texttt{letrec}\ x = \Omega, y = \Omega, Env\ \texttt{in}\ r) \to (\texttt{letrec}\ x = \Omega, Env[x/y]\ \texttt{in}\ r[x/y])$

Fig. 5. Reduction rules for bot-terms

E Strict Subexpressions

We prove that a strict subexpression s of t in a surface context can be reduced eagerly to WHNF.
In the following a strict subterm s of t always includes its position in t. We assume that the subterm s is labeled, that the reduction respects labels and that labels can be identified in the reduct, unless the reduction is an (llet)-reduction that destroys the top level `letrec` of s, or s is eliminated.

Lemma 36. *Let s be a strict subterm of the expression t, where $t = S[s]$ and S is a surface context. Then for every terminating normal order reduction sequence $t \xrightarrow{n,*} t_0$, there is an intermediate term $R[s]$, such that $t \xrightarrow{n,*} R[s] \xrightarrow{n,*} t_0$, R is a reduction context, and $R[s]$ is the first term in this sequence where the subexpression s is in a reduction context.*

Proof. Since S is a surface context, if the reduction is independent of s, then the term s is either removed by a normal order reduction step or it remains in a surface context, and in particular, the successor subterm of s is unique. Suppose there is a terminating normal order reduction $t \xrightarrow{n,*} t_0$, where s is never in a reduction context. Then we can replace s by Ω and get the same terminating

normal order reduction sequence. This contradicts the assumption that s is a strict subterm of t. Hence we will find an intermediate term $R[s]$, as required.

Lemma 37. *Let s be a strict subterm of the expression t, where $t = S[s]$ and S is a surface context. Then the following holds:*

1. *If s is of one of the following forms:*
   $(\texttt{letrec}\ E\ \texttt{in}\ s')$, $(s'\ s'')$, $(\texttt{seq}\ s'\ s'')$, or $(\texttt{case}_T\ s'\ alts)$,
   then s' is a strict subterm of t.
2. *Every superterm of s in t is a strict subterm of t.*
3. *If $t = C[(\texttt{letrec}\ x = s', E\ \texttt{in}\ C'[x])]$, and x is a strict subterm of t in a surface context, then s' is also a strict subterm of t.*

Proof. The first claim follows from Corollary 5, since $(\texttt{letrec}\ E\ \texttt{in}\ \Omega) \sim_c (\Omega\ s'') \sim_c (\texttt{seq}\ \Omega\ s'') \sim_c (\texttt{case}_T\ \Omega\ alts) \sim_c \Omega$, since in each of the expressions Ω is in a reduction context of s.
The second claim follows from the properties of a precongruence: If $t = C[D[s]]$ we have $C[D[\Omega]] \sim_c \Omega$. Since $\Omega \leq_c D[\Omega]$, we obtain $C[\Omega] \leq_c C[D[\Omega]] \sim_c \Omega$, hence $C[\Omega] \sim_c \Omega$.
The third claim can be proved using Lemma 36 which shows that every terminating normal order reduction of t has an intermediate term $R[x]$. Hence s' will occur under a reduction context in every terminating normal order reduction, Thus s' is a strict subterm of t.

Lemma 38. *Let t be a term with $t{\Downarrow}$, let s be a strict subterm of the expression t with $s \neq t$, $t = S[s]$ where S is a surface context, and s is not a value and not a* `letrec`*-expression. If $t \xrightarrow{a} t'$ by a reduction a from the base calculus, then s or its contractum is also a strict subterm of t'.*

Proof. If the reduction is within s, i.e. $s \to s'$ and $t' = t[s'/s]$, then the lemma holds, since $t[\Omega/s] \sim_c \Omega$, and so also $t[\Omega/s'] \sim_c \Omega$. This also holds, if a (cp) or (seq) has its inner redex in s, but the redex is not in s. If a (case)-reduction is such that the `case`-expression is within s, and the constructor application is not in s, then we have $t[\Omega/s] \xrightarrow{abs} t'[\Omega/s']$, hence by Theorem 2, we obtain $t'[\Omega/s'] \sim_c \Omega$,
If the reduction does not change s, then also $t[\Omega/s] \to t'[\Omega/s]$ by a reduction from the base calculus. In this case Theorem 1 shows that $t'[\Omega/s] \sim_c \Omega$.
The other case is that s is not changed, but eliminated by (seq) or (case). In this case we have $t[\Omega/s] \to t'[\Omega/s] = t'$ and we reach the contradiction $t' \sim_c \Omega$ using Theorem 1.
It is not possible by assumption that s is copied by a (cp), or that the top level of s is destroyed by a (lll)-reduction, or that s is the constructor application used in a (case)-reduction, or that s is the head of a (lbeta)-reduction.

F Reduction Lengths for Different Reductions

We prove the properties of the length of normal order reduction sequences.

For claims about lengths of reductions, only complete sets of forking diagrams are required. On the other hand, we cannot use the context lemma, and thus also have to treat overlappings where the reduction is within the body of a lambda abstraction.
In the following section we prove 5. The claim is:

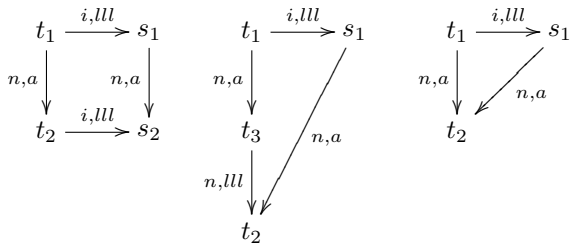
> Let t_1, s_1 be closed and terminating concrete expressions with $t_1{\Downarrow}$ and $t_1 \to s_1$. Then $s_1{\Downarrow}$ and the following holds:
>
> 1. If $t_1 \xrightarrow{a} s_1$ with $a \in \{\text{case}, \text{seq}, \text{lbeta}, \text{cp}\}$, then $\text{rl}(t_1) \geq \text{rl}(s_1)$, $\text{rl}\sharp(t_1) \geq \text{rl}\sharp(s_1)$ and $\text{rl}\sharp\sharp(t_1) \geq \text{rl}\sharp\sharp(s_1)$.
> 2. If $t_1 \xrightarrow{\mathcal{S},a} s_1$ with $a \in \{\text{caseS}, \text{seqS}, \text{lbeta}, \text{cpS}\}$, then $\text{rl}\sharp(t_1) \geq \text{rl}\sharp(s_1) \geq \text{rl}\sharp(t_1) - 1$ and $\text{rl}\sharp\sharp(t_1) \geq \text{rl}\sharp\sharp(s_1) \geq \text{rl}\sharp\sharp(t_1) - 1$. For $a = \text{cpS}$, the equation $\text{rl}\sharp\sharp(t_1) = \text{rl}\sharp\sharp(s_1)$ holds.
> 3. If $t_1 \xrightarrow{a} s_1$ with $a \in \{\text{lll}, \text{gc}\}$, then $\text{rl}(t_1) \geq \text{rl}(s_1)$, $\text{rl}\sharp(t_1) = \text{rl}\sharp(s_1)$ and $\text{rl}\sharp\sharp(t_1) = \text{rl}\sharp\sharp(s_1)$. For $a = gc1$ in addition $\text{rl}(t_1) = \text{rl}(s_1)$ holds.
> 4. If $t_1 \xrightarrow{a} s_1$ with $a \in \{\text{cpx}, \text{cpax}, \text{xch}, \text{cpcx}, \text{abs}\}$, then $\text{rl}(t_1) = \text{rl}(s_1)$, $\text{rl}\sharp(t_1) = \text{rl}\sharp(s_1)$ and $\text{rl}\sharp\sharp(t_1) = \text{rl}\sharp\sharp(s_1)$.
> 5. If $t_1 \xrightarrow{ucp} s_1$, then $\text{rl}(t_1) \geq \text{rl}(s_1)$, $\text{rl}\sharp(t_1) \geq \text{rl}\sharp(s_1)$ and $\text{rl}\sharp\sharp(t_1) = \text{rl}\sharp\sharp(s_1)$.
> 6. If $t_1 \xrightarrow{lwas} s_1$, then $\text{rl}\sharp\sharp(t_1) = \text{rl}\sharp\sharp(s_1)$.

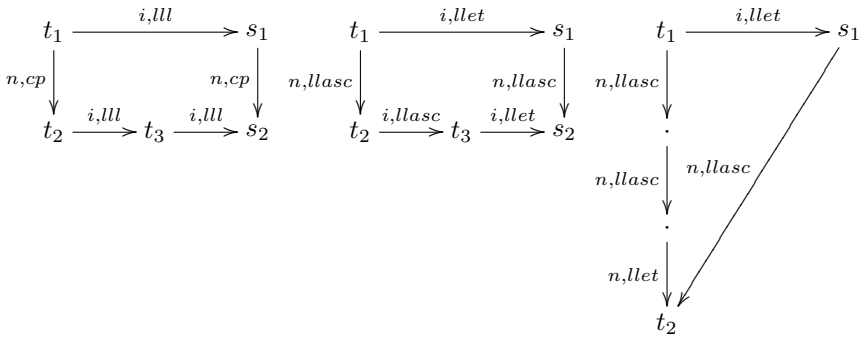
Proof. 1 follows from Proposition 39. 2 follows from Proposition 40. 3 follows from Proposition 32. 4 follows from Propositions 34, 35, 37 and 41. 5 follows from Proposition 36. 6 follows from Proposition 38.

F.1 Reduction Lengths for (lll) and (gc)

For the purposes of this subsection we denote the union of the reductions (lapp), (lseq), (lcase) as (llasc).

Lemma 39. *A complete set of forking and commuting diagrams for (i, lll), where a is an arbitrary reduction type, is as follows:*





Proof. We make the cases analysis for the forking diagrams. There are a number of standard cases:

- the reductions commute, or
- the reductions commute, and the (i,lll)-reduction is turned into a (n,lll)-reduction, or
- the (i,lll)-reduction is in a term that is removed by the reduction, i.e., a lost case-alternative.
- the (i,lll)-reduction is within a copied abstraction.

This leads to cases 1 to 4.
All overlappings of an (i,b)-reduction, where $b \in \{(\text{lseq}), (\text{lcase}), (\text{lapp}), (\text{llet-e})\}$ lead to a commuting diagram. The non-standard cases are overlappings of a reduction $\xrightarrow{i,\text{llet-in}}$ with a normal order redex: we demonstrate the reductions by representative examples.

- $$\begin{array}{ll} & (((\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ t_1))\ t_2)\ t_3) \\ \xrightarrow{n,lapp} & ((\texttt{letrec}\ Env_1\ \texttt{in}\ ((\texttt{letrec}\ Env_2\ \texttt{in}\ t_1)\ t_2))\ t_3) \\ \xrightarrow{i,lapp} & ((\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ (t_1\ t_2)))\ t_3) \\ \xrightarrow{i,llet} & ((\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ (t_1\ t_2))\ t_3) \end{array}$$
- $$\begin{array}{ll} & (((\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ t_1))\ t_2)\ t_3) \\ \xrightarrow{i,llet} & (((\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ t_1)\ t_2)\ t_3) \\ \xrightarrow{n,lapp} & ((\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ (t_1\ t_2))\ t_3) \end{array}$$

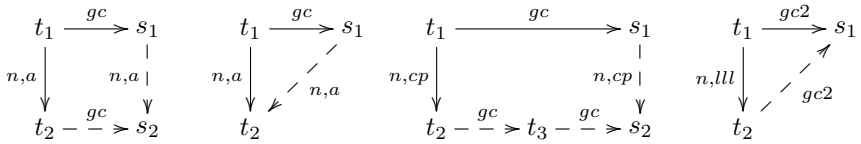
This is covered in the diagram number 5.
A slight variation is the case:

- $((\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ t_1))\ t_2)$
  $\xrightarrow{n,lapp}$ $(\texttt{letrec}\ Env_1\ \texttt{in}\ ((\texttt{letrec}\ Env_2\ \texttt{in}\ t_1)\ t_2))$
  $\xrightarrow{n,lapp}$ $(\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ (t_1\ t_2)))$
  $\xrightarrow{n,llet}$ $(\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ (t_1\ t_2))$

- $((\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ t_1))\ t_2)$
  $\xrightarrow{i,llet}$ $((\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ t_1)\ t_2)$
  $\xrightarrow{n,lapp}$ $(\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ (t_1\ t_2))$

This corresponds to the diagram 6.
The same holds if (lapp) is replaced by (lseq), or (lcase).
Checking all cases shows that no further diagrams are required.

Lemma 40. *A complete set of forking diagrams for (gc), where a is arbitrary, is as follows:*



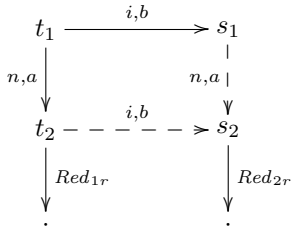
Proof. We omit the arguments for the cases 1,2,3.

Checking all possibilities for an overlap, it is clear that a (gc)-reduction can only overlap with a normal order reduction that requires a `letrec`. A non-trivial overlap is only possible, if (gc) removes the complete environment, i.e. only with (gc2). It is easy to check that all cases are covered by the diagrams (see also Lemma 21).

Now we can prove claim 3 of theorem 5.

Proposition 32. *Let t_1, s_1 be closed expressions with $t_1{\Downarrow}$, $Red_1 := nor(t_1)$ and $t_1 \xrightarrow{a} s_1$ where $a \in \{lll, gc\}$. Then $s_1{\Downarrow}$ and with $Red_2 := nor(s_1)$: $\mathrm{rl}(Red_1) \geq \mathrm{rl}(Red_2)$, $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ and $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$. If $a = (gc1)$, then Red_2 can be selected such that in addition $\mathrm{rl}(Red_1) = \mathrm{rl}(Red_2)$.*

Proof. The proof constructs a reduction Red_2 using induction on $\mathrm{rl}(Red_1)$.
If t_1 is a WHNF, then s_1 is also a WHNF by Lemmas 15 and 22.
First we treat the case that the reduction is an (i,lll):
Let t_1 be the starting term. Let $Red_1 = t_1 \xrightarrow{n,a} t_2 \cdot Red_{1r}$. In the triangular diagrams, it is easy to see that the reduction Red_2 satisfies the length properties.
For diagrams 1,4,5, the induction hypothesis has to be used.
The diagrams in Lemma 39 fix the notation of the terms t_i, s_i. So we associate a reduction Red_{2r} to s_2.



In any case, we have $\mathrm{rl}(Red_1) > \mathrm{rl}(Red_{1r})$, and so we can apply the induction hypothesis to Red_{1r}, perhaps two times, and obtain a reduction Red_{2r} starting from s_2.
It is easy to see inspecting the diagrams, that $\mathrm{rl}\flat(Red_1) \geq \mathrm{rl}\flat(Red_2)$. The additional contribution of the (n,a)-reduction, or the (n,cp)-reduction to $\mathrm{rl}\sharp(Red_1)$ or $\mathrm{rl}\sharp(Red_2)$ is the same in all diagrams, hence $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ holds using induction.
Now we consider the case that the internal reduction is a (gc). The diagrams in Lemma 40 are used.
In any case, we have $\mathrm{rl}(Red_1) > \mathrm{rl}(Red_{1r})$, and so we can apply the induction hypothesis to Red_{1r}.
The equality $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ holds in the diagram cases 1,2 since the (n,a)-reductions contribute the same number of reductions. The same for diagram 3, but we have to apply the induction hypothesis twice. In diagram 4, the (n,a)-reduction is a (n,lll), hence $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_{2r})$, and $\mathrm{rl}\sharp(Red_{2r}) = \mathrm{rl}\sharp(Red_1)$ by induction.
Exactly the same arguments show the claim of the theorem for $\mathrm{rl}\sharp\sharp(\cdot)$ for (gc) and (lll)-reductions.
The easy induction proof for the property of (gc1) is done by the length $\mathrm{rl}(\cdot)$, omitting diagram 4.

Proposition 33. *Let t_1, s_1 be a closed expressions with $t_1\Downarrow$, $Red_2 := nor(s_1)$ and $t_1 \xrightarrow{lll} s_1$. Then $s_1\Downarrow$ and with$Red_1 := nor(t_1)$:* $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ *and* $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$.

Proof. Using Lemma 39 the proof constructs a reduction Red_1 using induction on the following measure of reduction sequences, which are a mix of (i,lll)-reductions and normal order reductions:
It is a multiset with the multiset ordering, where the multiset consists of a triple for every (i,lll)-reduction:

1. The number of (n,cp)-reductions to the right of it.
2. If the reduction is $r_1 \xrightarrow{i,lll} r_2$, then $\mu_{lll}(r_1)$.

1. Diagram 1 strictly reduces in one triple either μ_1, or leaves μ_1 and strictly decreases μ_2.
2. Diagram 2,3,6 remove one triple from the multiset.

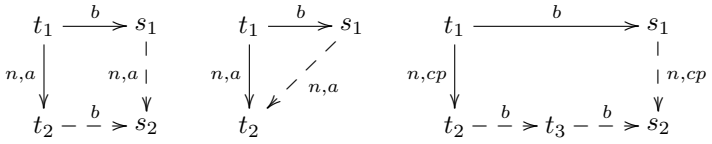
3. Diagram 4 replaces a triple by two strictly smaller triples, where the first component is strictly smaller.
4. Diagram 5 replaces a triple by two strictly smaller triples, where the second component is strictly smaller.

Since the ordering is well-founded, the shifting terminates with a normal order reduction. Furthermore, the number of (seq), (case), (lbeta), (cp)-reductions is not modified. Hence the claim holds.

F.2 Reduction Length for (cpx)-, (cpax)- and (xch)-Reductions

We compute the effect of (cpx)- and (xch)-reductions on the length of normal order reduction sequences. Note that the diagrams from Lemma 23 have to be reconsidered, since now all positions in a term have to be covered.

Lemma 41. *A complete set of forking diagrams for $b \in \{cpx, xch\}$ in all contexts is as follows:*



Proof. There are only the standard overlappings.

Concerning the length of normal order reductions, the following holds:

Proposition 34. *Let t_1 be a closed expression with $t_1{\Downarrow}$, $Red_1 := nor(t_1)$, and $t_1 \xrightarrow{b} s_1$ where $b \in \{cpx, xch\}$. Then $s_1{\Downarrow}$ and with $Red_2 := nor(s_1)$ we have* $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$, $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ *and* $\mathrm{rl}(Red_1) = \mathrm{rl}(Red_2)$.

Proof. This follows by induction on $\mathrm{rl}(Red_1)$ from Lemma 41, Lemma 24 and 27.

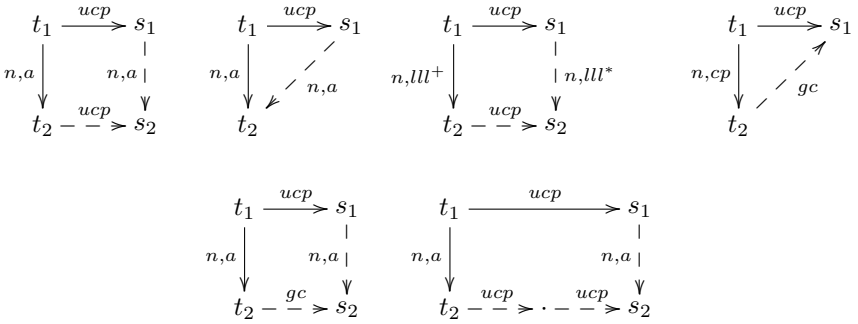
We have to treat the length-modifications by (cpax)-reductions:

Proposition 35. *Let t_1 be a closed expression with $t_1{\Downarrow}$, $Red_1 := nor(t_1)$, and $t_1 \xrightarrow{cpax} s_1$. Then $s_1{\Downarrow}$ and with $Red_2 := nor(s_1)$ we have* $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$, $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ *and* $\mathrm{rl}(Red_1) = \mathrm{rl}(Red_2)$.

Proof. This follows by induction on the number of variables occurrences that are replaced by the (cpax)-reduction, and from Proposition 34, since the (cpax)-reduction can be simulated by several (cpx) reductions.

F.3 Reduction Length for ucp-Reductions

Lemma 42. *A complete sets of forking diagrams for $\xrightarrow{ucp}$ in arbitrary contexts is as follows:*



where $a \in \{(cp), (case)\}$ in the 6^{th} diagram.

Proof. The first five diagrams are as in Lemma 32, the 6^{th} diagram covers the same case as the 6^{th} case in Lemma 32 and in addition the case that the (ucp) takes place in the body of an abstraction.

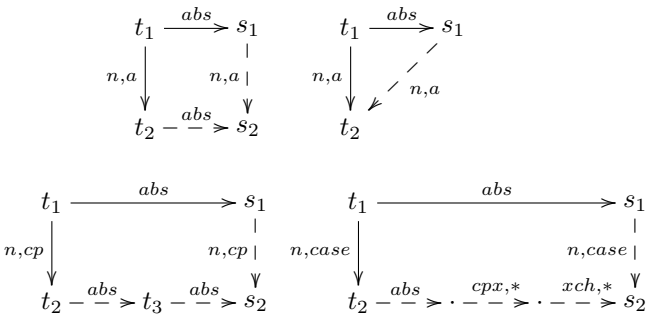
Proposition 36. *Let t_1 be a closed expression with $t_1{\Downarrow}$, $Red_1 := nor(t_1)$ and $t_1 \xrightarrow{ucp} s_1$. Then $s_1{\Downarrow}$ and with $Red_2 := nor(s_1)$ we have $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$ and $\mathrm{rl}\sharp(Red_1) \geq \mathrm{rl}\sharp(Red_2)$.*

Proof. This follows by induction on $\mathrm{rl}\sharp(Red_1)$ and then on $\mathrm{rl}(Red_1)$ from Lemma 33, Lemma 42 and Proposition 32.

F.4 Reduction Length for (abs)

For the definition of the (abs)-reduction see figure 3.

Lemma 43. *The forking diagrams for (abs) in arbitrary contexts are as follows.*



Proof. The cases are standard, only the last diagram requires an explicit justification:

$$
\begin{array}{ll}
 & (\texttt{letrec}\ x = c\ t_1\ t_2\ \texttt{in}\ C[\texttt{case}_T\ x\ (c\ y_1\ y_2) \to s]) \\
\xrightarrow{abs} & (\texttt{letrec}\ x = c\ x_1\ x_2, x_1 = t_1, x_2 = t_2\ \texttt{in}\ C[\texttt{case}_T\ x\ (c\ y_1\ y_2) \to s]) \\
\xrightarrow{n,case} & (\texttt{letrec}\ x = c\ z_1\ z_2, z_1 = x_1, z_2 = x_2, x_1 = t_1, x_2 = t_2\ \texttt{in} \\
 & \quad C[(\texttt{letrec}\ y_1 = z_1, y_2 = z_2\ \texttt{in}\ s)]) \\
\hline
\xrightarrow{n,case} & (\texttt{letrec}\ x = c\ z_1\ z_2, z_1 = t_1, z_2 = t_2\ \texttt{in}\ C[(\texttt{letrec}\ y_1 = z_1, y_2 = z_2\ \texttt{in}\ s)]) \\
\xrightarrow{abs} & (\texttt{letrec}\ x = c\ x_1\ x_2, x_1 = z_1, x_2 = z_2, z_1 = t_1, z_2 = t_2\ \texttt{in} \\
 & \quad C[(\texttt{letrec}\ y_1 = z_1, y_2 = z_2\ \texttt{in}\ s)]) \\
\xrightarrow{cpx,*} & (\texttt{letrec}\ x = c\ z_1\ z_2, x_1 = z_1, x_2 = z_2, z_1 = t_1, z_2 = t_2\ \texttt{in} \\
 & \quad C[(\texttt{letrec}\ y_1 = z_1, y_2 = z_2\ \texttt{in}\ s)]) \\
\xrightarrow{xch,*} & (\texttt{letrec}\ x = c\ z_1\ z_2, z_1 = x_1, z_2 = x_2, x_1 = t_1, x_2 = t_2\ \texttt{in} \\
 & \quad C[(\texttt{letrec}\ y_1 = z_1, y_2 = z_2\ \texttt{in}\ s)])
\end{array}
$$

Proposition 37. *Let t_1, s_1 be closed expressions with $t_1{\Downarrow}$, $Red_1 := nor(t_1)$ and $t_1 \xrightarrow{abs} s_1$. Then $s_1{\Downarrow}$ and with $Red_2 := nor(s_1)$ we have* $\mathrm{rl}(Red_1) = \mathrm{rl}(Red_2)$, $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ *and* $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$.

Proof. The proof is by induction on $\mathrm{rl}(Red_1)$, where the diagrams in Lemma 43 are used, and part 4 in Theorem 5, and since (abs) transforms WHNFs into WHNFs and vice versa.

F.5 Reduction Length for (lwas)-Reductions

Proposition 38. *Let t_1 be a closed expression with $t_1{\Downarrow}$, $Red_1 := nor(t_1)$ and $t_1 \xrightarrow{lwas} s_1$. Then $s_1{\Downarrow}$ and with $Red_2 := nor(s_1)$ we have* $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$.

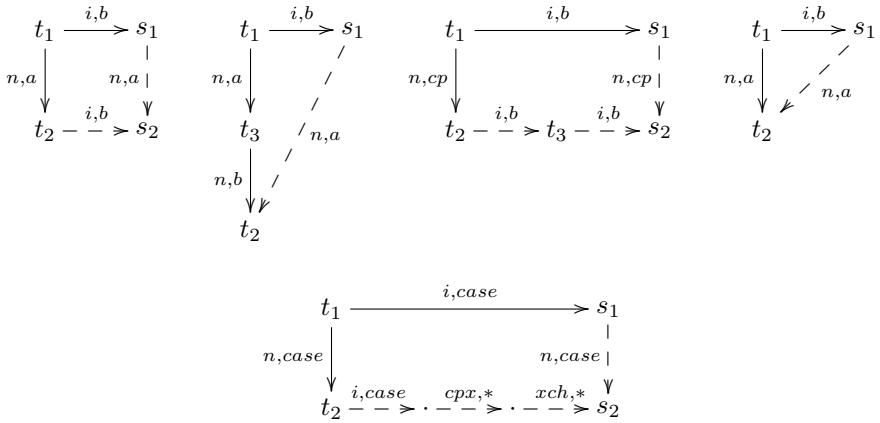
Proof. Since (lwas) can be simulated using (ucp) and (llet)-reductions in both directions (see proof of Lemma 29), Propositions 36, and 32 show the claim.

It would also be possible to sharpen this proposition, however, this is not necessary for the further development.

F.6 Using Diagrams for Internal Base Reductions

Now we analyze the length of normal order reductions for internal base reductions.

Lemma 44. *A complete set of forking diagrams for internal reductions with $b \in \{case, seq, lbeta, cp\}$, where a is the kind of the normal order reduction, and all contexts are permitted, is as follows*



Proof. The conflicts are only between (i,b) and the rule (cp), in which case the b-reduction may be within the copied expression, or in a removed alternative of a case, or in a subterm removed by (seq).
The exceptional diagram is a (case)-(case)-overlapping:

$$
\begin{array}{ll}
 & (\texttt{letrec}\ x = c\ t_1\ t_2\ \texttt{in}\ C[\texttt{case}_T\ x\ (c\ z_{1,1}\ z_{1,2}) \to s_1, \texttt{case}_T\ x\ (c\ z_{2,1}\ z_{2,2}) \to s_2]) \\
\xrightarrow{i,case} & (\texttt{letrec}\ x = c\ y_1\ y_2, y_1 = t_1, y_2 = t_2\ \texttt{in} \\
 & C[\texttt{case}_T\ x\ (c\ z_{1,1}\ z_{1,2}) \to s_1, (\texttt{letrec}\ z_{2,1} = y_1, z_{2,2} = y_2\ \texttt{in}\ s_2)]) \\
\xrightarrow{n,case} & (\texttt{letrec}\ x = c\ y_1'\ y_2', y_1 = t_1, y_2 = t_2, y_1' = y_1, y_2' = y_2\ \texttt{in} \\
 & C[(\texttt{letrec}\ y_1' = z_{1,1}, y_2' = z_{1,2}\ \texttt{in}\ s_1), (\texttt{letrec}\ z_{2,1} = y_1, z_{2,2} = y_2\ \texttt{in}\ s_2)]) \\
\hline
\xrightarrow{n,case} & (\texttt{letrec}\ x = c\ y_1'\ y_2', y_1' = t_1, y_2' = t_2\ \texttt{in} \\
 & C[(\texttt{letrec}\ z_{1,1} = y_1', z_{1,2} = y_2'\ \texttt{in}\ s_1), \texttt{case}_T\ x\ (c\ z_{2,1}\ z_{2,2}) \to s_2]) \\
\xrightarrow{i,case} & (\texttt{letrec}\ x = c\ y_1\ y_2, y_1' = t_1, y_2' = t_2, y_1 = y_1', y_2 = y_2'\ \texttt{in} \\
 & C[(\texttt{letrec}\ z_{1,1} = y_1', z_{1,2} = y_2'\ \texttt{in}\ s_1), (\texttt{letrec}\ z_{2,1} = y_1, z_{2,2} = y_2\ \texttt{in}\ s_2)]) \\
\xrightarrow{i,cpx,*} & (\texttt{letrec}\ x = c\ y_1'\ y_2', y_1' = t_1, y_2' = t_2, y_1 = y_1', y_2 = y_2'\ \texttt{in} \\
 & C[(\texttt{letrec}\ z_{1,1} = y_1', z_{1,2} = y_2'\ \texttt{in}\ s_1), (\texttt{letrec}\ z_{2,1} = y_1, z_{2,2} = y_2\ \texttt{in}\ s_2)]) \\
\xrightarrow{i,xch,*} & (\texttt{letrec}\ x = c\ y_1'\ y_2', y_1 = t_1, y_2 = t_2, y_1' = y_1, y_2' = y_2\ \texttt{in} \\
 & C[(\texttt{letrec}\ z_{1,1} = y_1', z_{1,2} = y_2'\ \texttt{in}\ s_1), (\texttt{letrec}\ z_{2,1} = y_1, z_{2,2} = y_2\ \texttt{in}\ s_2)])
\end{array}
$$

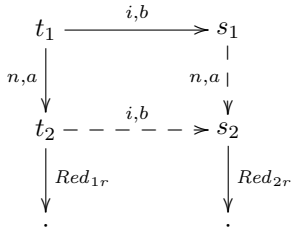
Lemma 45. *If t is a closed WHNF, and $t \xrightarrow{i,b} t'$ for $b \in \{case, seq, cp, lbeta\}$, then t' is a (closed) WHNF.*

Proof. This follows by checking the possible positions of the reduction in a WHNF.

Now we can prove claim 1 of Theorem 5

Proposition 39. *Let t_1, s_1 be closed expressions with $t_1{\Downarrow}$, $Red_1 := nor(t_1)$ and $t_1 \xrightarrow{a} s_1$ where $a \in \{case, seq, lbeta, cp\}$. Then $s_1{\Downarrow}$ and with $Red_2 := nor(s_1)$ we have $\mathrm{rl}(Red_1) \geq \mathrm{rl}(Red_2)$, $\mathrm{rl}\sharp(Red_1) \geq \mathrm{rl}\sharp(Red_2)$ and $\mathrm{rl}\sharp\sharp(Red_1) \geq \mathrm{rl}\sharp\sharp(Red_2)$.*

Proof. The proof will be done by induction on the length $\mathrm{rl}(Red_1)$.
The induction base is that t_1 is in WHNF, in which case we apply Lemma 45 to show that $t_1 \xrightarrow{i,b} s_1$ and $\mathrm{rl}(Red_1) = 0$ imply $\mathrm{rl}(Red_2) = 0$, $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2) = 0$, and $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2) = 0$.
For the induction step assume that $Red_1 = t_1 \xrightarrow{n} t_2 \cdot Red_{1r}$ and $t_1 \xrightarrow{i,b} s_1$. Lemma 44 shows that there are 5 possible cases.
In any case, we have $\mathrm{rl}(Red_1) > \mathrm{rl}(Red_{1r})$, and so we can apply the induction hypothesis to Red_{1r}.
In case 2 the relations $\mathrm{rl}\sharp(Red_1) > \mathrm{rl}\sharp(Red_2)$, and $\mathrm{rl}(Red_1) > \mathrm{rl}(Red_2)$, and $\mathrm{rl}\sharp\sharp(Red_1) \geq \mathrm{rl}\sharp\sharp(Red_2)$ can be directly derived from the diagrams, and in case 4, we obtain $\mathrm{rl}\sharp(Red_1) \geq \mathrm{rl}\sharp(Red_2)$, $\mathrm{rl}(Red_1) \geq \mathrm{rl}(Red_2)$, and $\mathrm{rl}\sharp\sharp(Red_1) \geq \mathrm{rl}\sharp\sharp(Red_2)$.
We use the following notational conventions in this proof for the rectangle-cases 1,3,5:

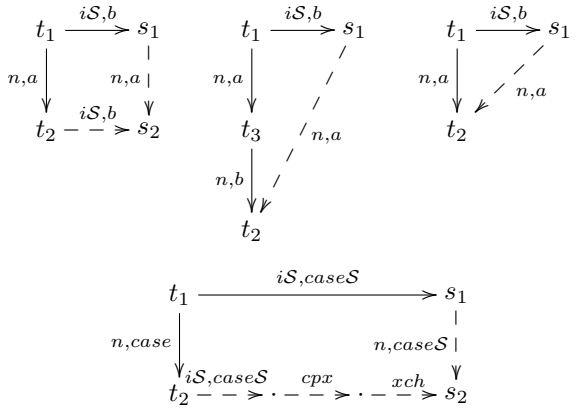


In case 1, we obtain by induction that there exists a reduction Red_{2r} of t_2 with $\mathrm{rl}\sharp(Red_{1r}) \geq \mathrm{rl}\sharp(Red_{2r})$, $\mathrm{rl}(Red_{1r}) \geq \mathrm{rl}(Red_{2r})$, and $\mathrm{rl}\sharp\sharp(Red_{1r}) \geq \mathrm{rl}\sharp\sharp(Red_{2r})$.
In case 3, we have to apply the induction hypothesis twice and obtain that there is a reduction Red_3 with $\mathrm{rl}(Red_{1r}) \geq \mathrm{rl}(Red_3)$, hence also a reduction Red_{2r} of s_2 with $\mathrm{rl}\sharp(Red_{1r}) \geq \mathrm{rl}\sharp(Red_{2r})$, and $\mathrm{rl}(Red_{1r}) \geq \mathrm{rl}(Red_{2r})$, and $\mathrm{rl}\sharp\sharp(Red_{1r}) \geq \mathrm{rl}\sharp\sharp(Red_{2r})$.
In cases 1 and 3, we obtain $\mathrm{rl}(Red_1) \geq \mathrm{rl}(Red_2)$. Since the first normal order reductions starting from t_1 and from s_1 are of the same kind, we obtain also $\mathrm{rl}\sharp(Red_1) \geq \mathrm{rl}\sharp(Red_2)$ and $\mathrm{rl}\sharp\sharp(Red_1) \geq \mathrm{rl}\sharp\sharp(Red_2)$.
In the fifth case, we apply induction using the existence of appropriate normal order reduction sequences and the preservation of the lengths of these sequences by the (xch)- and (cpx)-reductions proved in Proposition 34.

F.7 Base Reductions in Surface Contexts

Now we treat the case of S-restricted internal base reductions in surface contexts, which is necessary to obtain sharper bounds in this case.

Lemma 46. *A complete set of forking diagrams for $b \in \{caseS, seqS, lbeta, cpS\}$, where a is the kind of the normal order reduction, and the b-reduction is in a surface context, is as follows:*

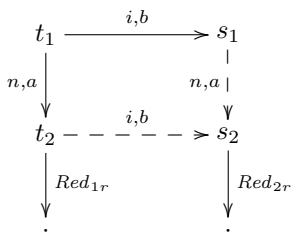


Proof. The same arguments as in the proof of Lemma 44 can be used. See also Lemma 18. Note that the duplicating (n,cp)-diagram does not occur, since the reductions are in surface contexts and the context C in their definition is also restricted to a surface context.

Now we can prove claim 2 of Theorem 5

Proposition 40. *Let t_1, s_1 be a closed expression with $t_1\Downarrow$, $Red_1 := nor(t_1)$ and $t_1 \xrightarrow{\mathcal{S},a} s_1$ where $a \in \{caseS, seqS, lbeta, cpS\}$. Then $s_1\Downarrow$ and with $Red_2 := nor(s_1)$ we have $\mathrm{rl}\sharp(Red_1) \geq \mathrm{rl}\sharp(Red_2) \geq \mathrm{rl}\sharp(Red_1) - 1$ and $\mathrm{rl}\sharp\sharp(Red_1) \geq \mathrm{rl}\sharp\sharp(Red_2) \geq \mathrm{rl}\sharp\sharp(Red_1) - 1$. For $a = cpS$, in addition $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$ holds.*

Proof. Proposition 39 already shows that there exists $Red_2 = nor(s_1)$ with $\mathrm{rl}(Red_1) \geq \mathrm{rl}(Red_2)$, $\mathrm{rl}\sharp(Red_1) \geq \mathrm{rl}\sharp(Red_2)$ and $\mathrm{rl}\sharp\sharp(Red_1) \geq \mathrm{rl}\sharp\sharp(Red_2)$. So it remains to prove that $\mathrm{rl}\sharp(Red_2) \geq \mathrm{rl}\sharp(Red_1) - 1$ and $\mathrm{rl}\sharp\sharp(Red_2) \geq \mathrm{rl}\sharp\sharp(Red_1) - 1$ for the same constructed reduction Red_2.
The proof will be done by induction on the length $\mathrm{rl}(Red_1)$. The induction base is that t_1 is in WHNF, in which case we apply Lemma 45 to show that $t_1 \xrightarrow{i,a} s_1$ and $\mathrm{rl}(Red_1) = 0$ imply $\mathrm{rl}(Red_2) = 0$, $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2) = 0$, and $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2) = 0$.
For the induction step assume that $t_1 \xrightarrow{n} t_2$ and $t_1 \xrightarrow{i\mathcal{S},b} s_1$. Lemma 46 shows that there are four possible cases.
We use the following notational conventions in this proof for the rectangle-case 1 :

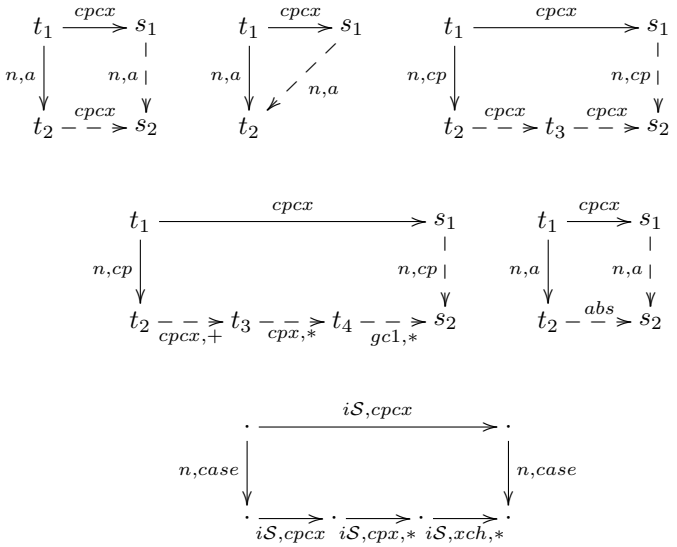


In case 1 we have $\mathrm{rl}(Red_1) > \mathrm{rl}(Red_{2r})$, and so we can apply the induction hypothesis to Red_{1r}.
Furthermore, there is a reduction Red_{2r} of s_2 with $\mathrm{rl}\sharp(Red_{2r}) \geq \mathrm{rl}\sharp(Red_{1r}) - 1$ and $\mathrm{rl}\sharp\sharp(Red_{2r}) \geq \mathrm{rl}\sharp\sharp(Red_{1r}) - 1$ by induction hypothesis. This implies the claim, by adding a δ to either side of the two inequations, where δ may be 0 or 1 depending on the kind of reduction a.
In case 2, the measures depend on the kind of reductions a, b: The equation $\mathrm{rl}\sharp(Red_1) - 1 = \mathrm{rl}\sharp(Red_2)$ holds, and either the equation $\mathrm{rl}\sharp\sharp(Red_1) - 1 = \mathrm{rl}\sharp\sharp(Red_2)$ or $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$ holds.
In case 3, the equations $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ and $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$ hold.
In case 4, the equations $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ and $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$ hold by induction similar to diagram 1 using Proposition 34.
In the case that $a = \mathrm{cpS}$, the equation $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$ follows by induction using the diagrams 1,2,3.

F.8 Reduction Length for (cpcx)

The reduction (cpcx) is defined as follows (see also Definition 15).

Lemma 47. *A complete set of forking diagrams for (cpcx) in arbitrary contexts is as follows.*



Proof. The first three cases cover the standard cases, prototypical examples for the other diagrams are already in the proof of Lemma 30. We give a further prototypical example for diagram 6:

$$
\begin{array}{ll}
 & (\texttt{letrec}\ x = c\ t_1\ t_2, y = x\ \texttt{in}\ \texttt{case}_T\ y\ (c\ y_1\ y_2) \to s) \\
\xrightarrow{cpcx} & (\texttt{letrec}\ x = c\ x_1\ x_2, x_1 = t_1, x_2 = t_2, y = c\ x_1\ x_2\ \texttt{in}\ \texttt{case}_T\ y\ (c\ y_1\ y_2) \to s) \\
\xrightarrow{n,case} & \texttt{letrec}\ x = c\ x_1\ x_2, x_1 = t_1, x_2 = t_2, y = c\ z_1\ z_2, z_1 = x_1, z_2 = x_2 \\
 & \qquad \texttt{in}\ (\texttt{letrec}\ y_1 = z_1, y_2 = z_2\ \texttt{in}\ s) \\
\hline
\xrightarrow{n,case} & (\texttt{letrec}\ x = c\ x_1\ x_2, x_1 = t_1, x_2 = t_2, y = x\ \texttt{in}\ (\texttt{letrec}\ y_1 = x_1, y_2 = x_2\ \texttt{in}\ s)) \\
\xrightarrow{cpcx} & \texttt{letrec}\ x = c\ z_1\ z_2, z_1 = x_1, z_2 = x_2, x_1 = t_1, x_2 = t_2, y = c\ z_1\ z_2 \\
 & \qquad \texttt{in}\ (\texttt{letrec}\ y_1 = x_1, y_2 = x_2\ \texttt{in}\ s) \\
\xrightarrow{cpx,*} & \texttt{letrec}\ x = c\ x_1\ x_2, z_1 = x_1, z_2 = x_2, x_1 = t_1, x_2 = t_2, y = c\ z_1\ z_2 \\
 & \qquad \texttt{in}\ (\texttt{letrec}\ y_1 = z_1, y_2 = z_2\ \texttt{in}\ s)
\end{array}
$$

The following case is covered by diagram 5:

$$
\begin{array}{ll}
 & (\texttt{letrec}\ x = c\ t_1\ t_2\ \texttt{in}\ \texttt{case}_T\ x\ (c\ y_1\ y_2) \to s) \\
\xrightarrow{cpcx} & (\texttt{letrec}\ x = c\ x_1\ x_2, x_1 = t_1, x_2 = t_2\ \texttt{in}\ \texttt{case}_T\ (c\ x_1\ x_2)\ (c\ y_1\ y_2) \to s) \\
\xrightarrow{n,case} & \texttt{letrec}\ x = c\ z_1\ z_2, z_1 = x_1, z_2 = x_2, x_1 = t_1, x_2 = t_2 \\
 & \qquad \texttt{in}\ (\texttt{letrec}\ y_1 = z_1, y_2 = z_2\ \texttt{in}\ s) \\
\hline
\xrightarrow{n,case} & \texttt{letrec}\ x = c\ x_1\ x_2, x_1 = t_1, x_2 = t_2 \\
 & \qquad \texttt{in}\ (\texttt{letrec}\ y_1 = x_1, y_2 = x_2\ \texttt{in}\ s) \\
\xrightarrow{n,abs} & \texttt{letrec}\ x = c\ z_1\ z_2, z_1 = x_1, z_2 = x_2, x_1 = t_1, x_2 = t_2 \\
 & \qquad \texttt{in}\ (\texttt{letrec}\ y_1 = x_1, y_2 = x_2\ \texttt{in}\ s)
\end{array}
$$

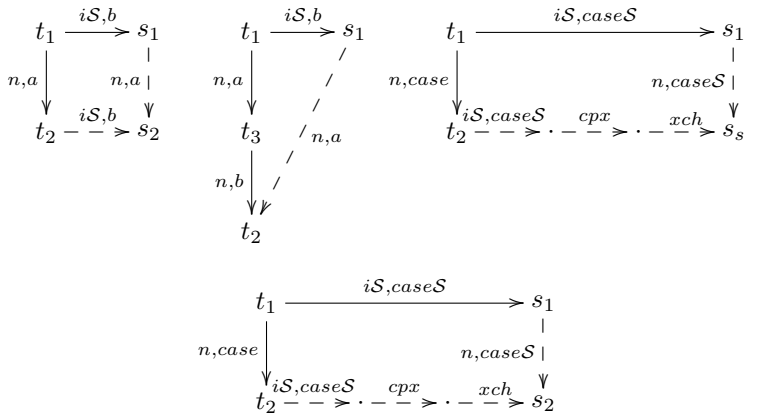
Proposition 41. *Let s_1, t_1 be closed expressions with $t_1{\Downarrow}$, $Red_1 := nor(t_1)$ and $t_1 \xrightarrow{cpcx} s_1$. Then $s_1{\Downarrow}$ and with $Red_2 := nor(s_1)$ we have* $\mathrm{rl}(Red_1) = \mathrm{rl}(Red_2)$, $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ *and* $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$.

Proof. The proof is by induction on $\mathrm{rl}(Red_1)$, where Lemmas 31 is used for the base case, and the diagrams in the following Lemmas are used: 47, 40, 43, and 41.

F.9 Length of Normal Order Reduction Using Strictness Optimization

In this subsection we give a proof of Proposition 6.

Proof (Proof of Proposition 6). We apply induction on $\mathrm{rl}(t_1)$.
It is not possible that t_1 is a WHNF, since then the condition that there is a b-redex on a surface position for $b \in \{(case\mathcal{S}), (seq\mathcal{S}), (lbeta), (cpt)\}$ and that $t_1[\texttt{Bot}/t_0] \sim_c \texttt{Bot}$ cannot hold simultaneously.
Let $t_1 \xrightarrow{n} t_2$.
Lemma 38 shows that the descendent of t_0 is also a strict subterm of t_2. The diagrams are as follows, where the $i\mathcal{S}$-reduction reduces the redex t_0 or its descendent (see also Lemma 46).



The short triangle-diagram from Lemma 46 does not occur, since t_0 remains a strict subterm.
We use induction on $\mathrm{rl}(t_1)$, where the diagrams above are the cases that have to be considered in the induction step, and use the already known results on the lengths of normal order reductions (see Theorem 5) for (xch) and (cpx)-reductions. We obtain that the claim of the proposition holds.

F.10 Local Evaluation and Deep Subterms

In this section we will prove the Propositions 7 and 8

Definition 36. *In the closed concrete term* (`letrec` $x = t, y = s, Env$ `in` r)*, we say* x requires y*, iff the local evaluation of* x *in* (`letrec` $x = t, y = \Omega, Env$ `in` r) *does not produce a WHNF for* x*, i.e., results in* Ω *for* x*.*

Lemma 48. *Let* $t = (\texttt{letrec}\ x = s_x, y = s_y, Env\ \texttt{in}\ r)$ *be a closed term, where* x *requires* y*, and let* $t \to t'$ *by a base- or an extra reduction. Then in* t' *the variable* x *also requires* y*.*

Proof. First assume that the reduction is not an (llet)-reduction.
If $t \to t'$ modifies only r, then the Lemma holds, since there is no difference in the local evaluations of x w.r.t. t and t'. If the reduction modifies a `case`-expression in r, where the constructor application is in the top environment, then $t' = (\texttt{letrec}\ x = s'_x, y = s'_y, Env'\ \texttt{in}\ r')$ and one of the two relations $(\texttt{letrec}\ x = s_x, y = \Omega, Env\ \texttt{in}\ x) = (\texttt{letrec}\ x = s'_x, y = \Omega, Env'\ \texttt{in}\ x)$ or $(\texttt{letrec}\ x = s_x, y = \Omega, Env\ \texttt{in}\ x) \xrightarrow{abs} (\texttt{letrec}\ x = s'_x, y = \Omega, Env'\ \texttt{in}\ x)$ holds. Theorem 2 implies that the Lemma holds.
If $t \to t'$ modifies the top environment, then Theorems 1 and 2 show that the Lemma holds.
Now assume that the reduction is an (llet)-reduction. If the (llet)-reduction does not change the top level structure of t, then again Theorem 2 shows that the Lemma holds.

The only non-standard case is that $s_y = (\texttt{letrec}\ Env_y\ \texttt{in}\ s'_y)$ and that it is modified by a (n,llet-in)-reduction: $t' = (\texttt{letrec}\ x = s_x, Env_y, y = s'_y, Env\ \texttt{in}\ r)$. Now the Lemma follows from Lemma 37.

Lemma 49. *Let the closed concrete term* $t = (\texttt{letrec}\ x_1 = t_1, \ldots, x_n = t_n, Env\ \texttt{in}\ r)$ *have a cyclic dependency, i.e.,* x_i *requires* x_{i+1} *for* $i = 1, \ldots, n-1$ *and* x_n *requires* x_1.
Then for all i*, the local evaluation of* x_i *does not produce a WHNF for* x_i.

Proof. W.l.o.g. we can assume the first reduction step of the local evaluation of x_1 to be a non-(lll) reduction step. Moreover assume, that this is the $\mathrm{rl}\sharp(\cdot)$-shortest such evaluation for all x_i.
If some x_i is bound to a term that is a value, then this contradicts the assumption that there is a cyclic dependency. Hence every x_i is bound to a term that is not a value. W.l.o.g. we can ignore the (lll)-reductions. Let the first reduction step of the local evaluation of x_1 be a non-(lll) reduction step. The cyclic dependency remains as before the reduction (see Lemma 48). The term t' is a counterexample with a shorter $\mathrm{rl}\sharp(\cdot)$-number of a successful local evaluation of an x_i, hence we have a contradiction. This means there is no finite successful local evaluation for x_i for any $i = 1, \ldots, n$.

We prove Proposition 7. The claim is:

> Let $t_1 = (\texttt{letrec}\ Env\ \texttt{in}\ t'_1)$ be a closed concrete LR-expression with $t_1{\Downarrow}$. Let $x \in LV(Env)$ where the binding is $x = t_x$, and t_x is a strict subexpression in t_1.
> Then $\mathrm{rl}\sharp(t_1) \geq \mathrm{rl}\sharp_{loc}(\texttt{letrec}\ Env\ \texttt{in}\ x)$ and $\mathrm{rl}\sharp\sharp(t_1) \geq \mathrm{rl}\sharp\sharp(\texttt{letrec}\ Env\ \texttt{in}\ x)$.

Proof. If $x = t'_1$ there is nothing to show. Hence in the following we assume $x \neq t'_1$.
The proof is by induction on $\mathrm{rl}\sharp_{loc}(\texttt{letrec}\ Env\ \texttt{in}\ x)$. If t_x is in WHNF, then $\mathrm{rl}\sharp_{loc}(\texttt{letrec}\ Env\ \texttt{in}\ x) = 0$, and the claim holds. Now let t_x be a non-WHNF. Let $t_1 \to t_2$ be the reduction corresponding to the first local evaluation step of x. If the reduction is an (lll)-reduction, then we can use induction and Theorem 5. It is easy to see that the inner redex of the reduction is a strict subterm of t_1. The other local reduction types are (cpS), (lbeta), (caseS), (seqS), hence Proposition 6 and induction on the number of local evaluations shows the claim.

Finally, we prove Proposition 8:

> Let $t_1 = (\texttt{letrec}\ Env\ \texttt{in}\ t'_1)$ be a closed concrete LR-expression with $t_1{\Downarrow}$. Let $x \in LV(Env)$ be a variable with binding $x = t_x$, such that t_x is a strict and deep subterm in t_1, and t_x is not a `letrec`-expression.
> Then $\mathrm{rl}\sharp\sharp(t_1) > \mathrm{rl}\sharp\sharp(\texttt{letrec}\ Env\ \texttt{in}\ x)$.

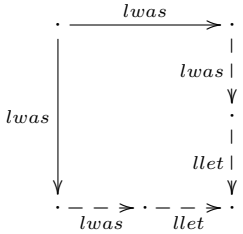
Proof. We show by induction on the number of local evaluation steps of x, that after a local evaluation of x, t_x remains a strict and deep subterm in t_1.

If t_x is already a WHNF, then it is a value. Due to the syntactic form of t_1, the normal order reduction of t_1 must include at least one (case), (seq), or (lbeta)-reduction to reach a normal form, hence the proposition holds.
If t_x is not a value, then consider a single local evaluation step of x in t_1, i.e. $t_1 \to t_2$. Then t_x remains a strict subterm in t_2 by Lemma 38.
We have to show that t_x is also a deep subterm in t_2. The subterm t_1' is not modified. If t_1' is not a variable, then t_x is already a deep subterm. In the case that t_1' is a variable, $t_1 = (\texttt{letrec}\ x = t_x, Env, x_n = r, x_{n-1} = x_n, \ldots, x_1 = x_2\ \texttt{in}\ x_1)$. Since t_x is a strict subterm, all variables $x_i, i = 1, \ldots, n$ require x in t_1. Since $t_1{\Downarrow}$, the variable x does not require $x_i, i = 1, \ldots, n$ by Lemma 49. Hence the local evaluation of x makes modifications only in t_x and Env. Hence t_x', the successor of t_x, is also a deep subterm in t_2.
If the next reduction in the local evaluation is a (lll)-reduction, then the measure $\mathrm{rl}\sharp\sharp$ does not change. If the next reduction in the local evaluation is a (cpS)-reduction, then apply Proposition 6. We obtain that $\mathrm{rl}\sharp\sharp(t_1) = \mathrm{rl}\sharp\sharp(t_2)$. If the next reduction in the local evaluation is a (caseS), (seqS), or (lbeta)-reduction, then apply Proposition 6. We obtain that $\mathrm{rl}\sharp\sharp(t_1) = 1 + \mathrm{rl}\sharp\sharp(t_2)$.
Hence the induction shows that $\mathrm{rl}\sharp\sharp(t_1) > \mathrm{rl}\sharp\sharp(\texttt{letrec}\ Env\ \texttt{in}\ x)$.

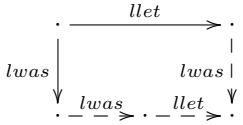
G Proof of Theorem 4

Proof. We have to compute the forking diagrams (critical pairs) between (lwas)-, (llet)-, (cpax)-, and (gc)-reductions in order to show local confluence of the reductions. We omit the cases of commutation of the reductions.
The forking diagrams are:

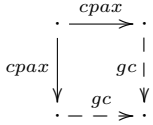
For (lwas) with (lwas):



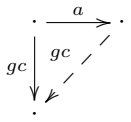
For (lwas) with (llet):



For (cpax) with (cpax):



For (gc) with other reductions:



For termination we only need a well-founded measure of terms that is strictly decreased in every reduction step. This measure μ is a tuple $(\mu_1(t), \mu_2(t), \mu_3(t))$, ordered lexicographically. The measure $\mu_1(t)$ is $\mu_{lll}(t)$ as defined in Definition 16 and used in the proof of Proposition 2, $\mu_2(t)$ is the number of all bindings in `letrec`-subexpressions in t, and $\mu_3(t)$ is the number of let-bound variables that have occurrences in the expression.
This measure of t is strictly decreased by every reduction (lwas), (cpax), (gc), (llet): The reductions (llet) and (lwas) strictly decrease μ_1, the reduction (gc) strictly decreases μ_1 or leaves μ_1 unchanged and strictly decreases μ_2, and the reduction (cpax) leaves μ_1, μ_2 unchanged, and strictly decreases μ_3.
Finally, we apply the well-known Newman's Lemma which states confluence for terminating and locally confluent reduction systems (see e.g. [BN98]).

H Another Definition of Contextual Equivalence

Proposition 42. *$s \leq_c t$ is equivalent to:*

$$\forall C[\cdot]: \ C[s], C[t] \text{ are closed} \Rightarrow \Big(C[s]{\Downarrow} \Rightarrow C[t]{\Downarrow}\Big)$$

Proof. One direction is trivial.
Assume that the following holds

$$\forall C[\cdot]: \ C[s], C[t] \text{ are closed} \Rightarrow \Big(C[s]{\Downarrow} \Rightarrow C[t]{\Downarrow}\Big)$$

Let D be an arbitrary context such that $D[s]{\Downarrow}$. Let $\{x_1, \ldots, x_n\}$ be the variables in $FV(D[s], D[t])$. Let $D' := (\texttt{letrec}\ \{x_i = \texttt{Bot}\}_{i=1}^n\ \texttt{in}\ D)$. Then $D'[s]{\Downarrow}$ follows from $D[s]{\Downarrow}$, and $D'[t]{\Downarrow}$ follows from the assumption. The terminating normal order reduction of $D'[t]{\Downarrow}$ never puts any x_i in a reduction context, since this contradicts Corollary 5. Hence the same method as in the proof of Proposition 30 shows that we can use the same normal order reduction to show that $D[t]{\Downarrow}$.

I Correctness of copying closed subterms

We treat the following situation: Let

$$t = (\texttt{letrec}\ \{x_{1,i} = y_{1,i}\}_{i=1}^n, \{x_{2,i} = y_{1,i}\}_{i=1}^n, \ldots \{x_{m,i} = y_{1,i}\}_{i=1}^n, Env_1, Env_{\text{rest}}\ \texttt{in}\ s)$$

be an expression, where $Env_1 = \{y_{1,i} = s_i\}_{i=1}^n$, $FV(Env_1) \subseteq LV(Env_1)$. We want to show that $t \sim_c t'$, where

$$t' = \texttt{letrec}\ \{x_{1,i} = y_{1,i}\}_{i=1}^n, \{x_{2,i} = y_{2,i}\}_{i=1}^n, \ldots, \{x_{m,i} = y_{m,i}\}_{i=1}^n, \\ Env_1, \ldots, Env_m, Env_{\text{rest}} \\ \texttt{in}\ s$$

where $Env_j, j = 1, \ldots, m$ are renamed copies of Env_1, where the renaming is $\rho_j := \{y_{1,i} \to y_{j,i}\}$, including the let-bound variables. More precisely, $Env_j = \{y_{j,i} = \rho_j(s_i)\}_{i=1}^n$
The variables $x_{j,i}$ are used for a different representation of a renaming of the variables in Env_{rest}, s, only affecting t'.

Proposition 43. *Under the above conditions, we have $t \sim_c t'$.*

Proof. We use the context lemma 2 to show contextual equivalence. For every reduction context R we have to show that $R[t]{\Downarrow} \Leftrightarrow R[t']{\Downarrow}$. It is obvious from the definition of normal order reduction that the first normal order reduction steps of $R[t]$ as well as of $R[t']$ are to shift the top environment of t (or of t') to the top environment of $R[t]$, or of $R[t']$, respectively.
Then the part

$$\{x_{1,i} = y_{1,i}\}_{i=1}^n, \ldots, \{x_{m,i} = y_{1,i}\}_{i=1}^n, Env_1$$

in the reduct of $R[t]$ is a part of the top environment. The rest of the top environment is denoted in the following as Env_{rest}. Note that Env_{rest} may contain a binding of the form $z = R'[s]$. We can write the intermediate term for $R[t]$ as

$$(\texttt{letrec}\ \{x_{1,i} = y_{1,i}\}_{i=1}^n, \ldots, \{x_{m,i} = y_{m,i}\}_{i=1}^n, Env_1, Env_{\text{rest}}\ \texttt{in}\ t_0)$$

and the intermediate term for $R[t']$ as

$$(\texttt{letrec}\ \{x_{1,i} = y_{1,i}\}_{i=1}^n, \ldots, \{x_{m,i} = y_{m,i}\}_{i=1}^n, Env_1, \ldots, Env_m, Env_{\text{rest}}\ \texttt{in}\ t_0).$$

Let R be a reduction context with $R[t]{\Downarrow}$. Let *Red* be the normal order reduction of $R[t]$ to WHNF. This reduction sequence *Red* is modified by replacing every (case)-reduction by $\xrightarrow{abs} \cdot \xrightarrow{case-cx}$, or by $\xrightarrow{case-cx}$, such that the latter replacements have the same effect as the original (case). The constructed reduction sequence is denoted as Red_1. We distinguish the reduction steps in Red_1 as follows:

- Env_1-related reductions: reductions that make changes in Env_1.
- Env_1-independent reductions: reductions that do not make changes in Env_1.

We construct a reduction sequence Red' of $R[t']$ from Red_1 as follows. An Env_1-independent reduction is simply copied to Red'. An Env_1-related reduction in Red_1 result in m reductions in Red': The reduction is copied to Red' and is followed by corresponding reductions w.r.t $Env_2, \ldots, Env_m$. (cp)-reductions in Red_1 may copy abstraction from Env_1 to positions in Env_{rest} or r. These have to

be modified as copying from Env_j if the variable chain contains a variable $x_{j,i}$. At most one $x_{j,i}$ may appear in a variable chain of such a (cp)-reduction. Other reductions remain the same up to potential renaming of the variables $y_{j,i}$. The invariant is that after merging $y_{1,i}, \ldots, y_{m,i}$ for every i and merging successive (corresponding) reductions on $Env_1, \ldots, Env_m$, we obtain the reduction for $R[t]$.

In summary, we have constructed a reduction sequence of $R[t']$ to a WHNF using reductions of the calculus and external reductions. From Lemma 34 and Theorem 3 we obtain $R[t']{\Downarrow}$.

In order to prove the other direction, let R be a reduction context with $R[t']{\Downarrow}$.

We fix a normal order reduction *Red* of $R[t']$ to a WHNF. As mentioned above, the first normal order reduction steps shift the top environment of t' to the top environment. Thus we may start reasoning with the term $(\texttt{letrec}\ \{x_{1,i} = y_{1,i}\}_{i=1}^{n}, \ldots, \{x_{m,i} = y_{1,i}\}_{i=1}^{n}, Env_1, Env_{\text{rest}}\ \texttt{in}\ t_0)$ as a reduct of $R[t]$, and $r_0 := (\texttt{letrec}\ \{x_{1,i} = y_{1,i}\}_{i=1}^{n}, \ldots, \{x_{m,i} = y_{m,i}\}_{i=1}^{n}, Env_1, \ldots, Env_m, Env_{\text{rest}}\ \texttt{in}\ t_0)$ as a reduct of $R[t']$.

Now we show that there is a reduction of the latter term satisfying synchronization conditions. The idea of the construction is to synchronize the reduction steps that occur in the environments $Env_1, \ldots, Env_m$, perhaps adding reductions if necessary. Eventually, this reduction sequence allows to construct a terminating reduction for $R[t]$

Since the environments Env_1 and Env_j are equal up to variable renaming, we speak of *corresponding* terms, positions and reductions. But note that during the course of reduction and construction, the renamed variables will also occur outside of $Env_1, \ldots, Env_m$.

Our first observation is that we can recognize the successor environments of the environments $Env_1, \ldots, Env_m$ in the expressions in the reduction sequence starting from $R[t']$. Our construction will maintain this correspondence property. Note that these environments may have more bindings than the original environments due to (lll)-reductions within the environments. In the following we argue that we can construct a reduction sequence to a WHNF with the following property:

Every reduction step making a modification in Env_j is immediately followed by a reduction that makes the corresponding modification in all environments Env_k for $k = 1, \ldots, m$ and $k \neq j$. Moreover all reductions are in surface contexts, and are base-reductions, (case-cx)-reductions or (abs)-reductions.

We start reduction with r_0, and let r be the current term with $r_0 \xrightarrow{*} r$. We show by induction on the pair $(\mathrm{rl}_\sharp^\sharp(r), \mu_{lll}(r))$, ordered lexicographically, that a normal order reduction starting with r, where r is of the form above, can be transformed into a reduction to WHNF satisfying the correspondence property. We go through the different possibilities of the first reduction step $r \xrightarrow{n} r'$:

If the reduction step $r \xrightarrow{n} r'$ does not modify the environments $Env_1, \ldots, Env_m$, then we use this reduction step and apply induction on r'.

Consider the case that the reduction step $r \xrightarrow{n} r'$ modifies a part of Env_j for

some j (or their successor-environments).

1. If the reduction is an (lll)-reduction, then apply the corresponding reductions in the other environments with index $k \neq j$ giving a term r''. Note that bindings may be added to Env_k, however, this is only possible by an (lll)-reduction within Env_k. Proposition 2 and Theorem 5 show that we can use the induction hypothesis for r''.
2. If the reduction is completely within Env_j, make the same reduction for all $Env_k, k \neq j$ giving r''. Theorem 5 shows that we can use the induction hypothesis for r''.
3. A (cp) into Env_j is not possible due to the conditions on variable occurrences.
4. If the reduction is a (case) where the inner redex is in Env_j, but the redex is external, then split the reduction into an (external) (abs) followed by a (case-cx), and make the corresponding (case-cx)-reductions also for $Env_k, k \neq j$ giving the expression r''. Since a (case-cx) can be simulated by (case) with following (cpx) and (gc) by Lemma 34. Proposition 5 and Theorem 5 show that we can use the induction hypothesis for r''.
5. If the effect in Env_j is an (abs), which comes from an external (case), then perform the corresponding (abs)-reductions also in $Env_k, k \neq j$ giving r''. Theorem 5 shows that the first component of the measure for r'' is not increased. Lemma 35 assures that the next reduction step to consider in the induction is a (case)-reduction, and hence after the next modification, the first component of the measure for r'' is decreased, and we can use the induction hypothesis.

Finally, we obtain a reduction sequence from r_0 to a WHNF, using only surface reduction from the base calculus, some extra reductions, and (case-cx), where the reductions in $Env_1, \ldots, Env_m$ are always corresponding ones and immediately follow each other.
Now it is easy to construct a terminating reduction sequence of $R[t]$: We only use the reductions for Env_1, but with the correct renaming, and also select one from every block of m subsequent corresponding reduction steps.
We finally have a reduction sequence of $R[t]$ ending in a WHNF, where the steps may be from the base calculus, (abs)- reductions and (case-cx). Now Lemma 34 and Theorem 3 show that $R[t]{\Downarrow}$.

Corollary 6. *Let S be a surface context in the expression* $(\texttt{letrec}\ x = t, Env\ \texttt{in}\ S[x])$ *with* $FV(t) = \emptyset$. *Then* $(\texttt{letrec}\ x = t, Env\ \texttt{in}\ S[x]) \sim_c (\texttt{letrec}\ x = t, Env\ \texttt{in}\ S[t])$.

Proof. We apply Proposition 43 with $(\texttt{letrec}\ x_1 = y_1, x_2 = y_1, y_1 = t, Env\ \texttt{in}\ S[x_2])$, where $Env_1 = \{y_1 = t\}$, and x_2 does not occur in $S[\cdot], t, Env$.
It is obvious that $(\texttt{letrec}\ x = t, Env\ \texttt{in}\ S[x]) \sim_c (\texttt{letrec}\ x_1 = y_1, x_2 = y_1, y_1 = t, Env\ \texttt{in}\ S[x_2])$ using (cpx) and (gc) and Theorem 9.
From Proposition 43 we obtain $(\texttt{letrec}\ x_1 = y_1, x_2 = y_1, y_1 = t, Env\ \texttt{in}\ S[x_2]) \sim_c (\texttt{letrec}\ x_1 = y_1, y_1 = t, x_2 = y_2, y_2 = t, Env\ \texttt{in}\ S[x_2]) \xrightarrow{cpx}$

$(\texttt{letrec}\ x_1 = y_1, y_1 = t, x_2 = y_2, y_2 = t, Env\ \texttt{in}\ S[y_2]) \xrightarrow{gc} (\texttt{letrec}\ x_1 = y_1, y_1 = t, y_2 = t, Env\ \texttt{in}\ S[y_2]) \xrightarrow{ucp} (\texttt{letrec}\ x_1 = y_1, y_1 = t, Env\ \texttt{in}\ S[t])$. Now the proof is complete using the correctness of (gc), (cpx) and (ucp) shown in Theorem 9.

Corollary 7. *Let* $t = (\texttt{letrec}\ Env_1, Env_{\text{rest}}\ \texttt{in}\ t_0)$*, such that* $FV(Env_1) \subseteq LV(Env_1)$*, where* $Env_1 = \{y_i = s_i\}_{i=1}^n$*, and let* $t' = (\texttt{letrec}\ y_1 = (\texttt{letrec}\ Env'_1\ \texttt{in}\ y_{1,1}), \ldots, y_n = (\texttt{letrec}\ Env'_n\ \texttt{in}\ y_{n,n}), Env_{\text{rest}}\ \texttt{in}\ t_0)$*, where* Env'_j *is* Env_1 *where* y_i *are renamed into* $y_{i,j}$ *for* $i, j = 1, \ldots, n$*.*
Then $t \sim_c t'$*.*

Proof. Proposition 43 is used as follows: Let Env'_{rest} and t'_0 be Env_{rest} and t_0, respectively, renamed by $\rho := \{y_i \to x_{i,i} \mid i = 1, \ldots, n\}$. Let $t_1 := (\texttt{letrec}\ \{x_{1,i} = y_i\}_{i=1}^n, \ldots, \{x_{n,i} = y_i\}_{i=1}^n, Env_1, Env'_{\text{rest}}\ \texttt{in}\ t'_0)$, such that $x_{i,j}$ for $i, j = 1, \ldots, n$ do not occur in any term $s_i, i = 1, \ldots, n$, y_i does not occur in Env'_{rest} nor in t'_0. Using (cpx) and (gc) and Theorem 9, it is easy to see that $t_1 \sim_c t$.
Let $(\texttt{letrec}\ \{x_{1,i} = y_{1,i}\}_{i=1}^n, \ldots, \{x_{n,i} = y_{n,i}\}_{i=1}^n Env'_1, \ldots, Env'_n, Env'_{\text{rest}}\ \texttt{in}\ t'_0)$, where Env'_j is Env_1 renamed by $\{y_i \to y_{j,i} \mid i = 1, \ldots, n\}$ for $j = 1, \ldots, n$. Then we obtain $t_1 \sim_c t_2$ by Proposition 43. The following equivalences hold:

$$\begin{aligned} t_2 &\sim_c \texttt{letrec}\ x_{1,1} = y_{1,1}, \ldots, x_{n,n} = y_{n,n}, Env'_1, \ldots, Env'_n, Env'_{\text{rest}}\ \texttt{in}\ t'_0 \\ &\sim_c \texttt{letrec}\ \{x_{i,i} = (\texttt{letrec}\ Env'_i\ \texttt{in}\ y_{i,i})\}_{i=1}^n, Env'_{\text{rest}}\ \texttt{in}\ t'_0 \end{aligned}$$

The first equivalence follows from correctness of (gc), and the second equivalence from correctness of (llet-e). The final step is to rename $x_{i,i}$ into y_i, and we obtain that the last term is $\sim_c t'$.

We will now prove Theorem 6. The claim is:

> Let $t = (\texttt{letrec}\ Env\ \texttt{in}\ (c\ t_1 \ldots t_m))$ be a closed expression, where $Env = \{y_i = s_i\}_{i=1}^n$, and let $t'_j := (\texttt{letrec}\ Env_j\ \texttt{in}\ t'_j)$ for $j = 1, \ldots, m$, where Env_j and t'_j is Env and t_j renamed by $\rho_j := \{y_i \to y_{j,i} \mid i = 1, \ldots, n\}$.
> Then for all j: the expressions t'_j are closed and $t \sim_c (c\ t'_1\ \ldots\ t'_m)$

Proof (Proof of Theorem 6). This follows from Proposition 43 as follows. We start with $t' = (c\ (\texttt{letrec}\ Env_1\ \texttt{in}\ t'_1)\ \ldots\ (\texttt{letrec}\ Env_m\ \texttt{in}\ t'_m))$. Using correctness of (lwas) (see Theorem 2) and (llet), we obtain that $t' \sim_c (\texttt{letrec}\ Env_1, \ldots, Env_m\ \texttt{in}\ (c\ t'_1\ \ldots\ t'_m))$. Using correctness of (cpx) and (gc), we obtain that this term is equivalent w.r.t. $\sim_c$ to the term $(\texttt{letrec}\ \{x_{1,i} = y_{1,i}\}_{i=1}^n, \ldots, \{x_{m,i} = y_{m,i}\}_{i=1}^n, Env_1, \ldots, Env_m\ \texttt{in}\ c\ t''_1\ \ldots\ t''_m)$, where t''_j is derived from t'_j by renaming using $\{y_{j,i} \to x_{j,i} \mid i = 1, \ldots, n \wedge j = 1, \ldots, m\}$. Proposition 43 shows that this term is equivalent to $(\texttt{letrec}\ \{x_{1,i} = y_{1,i}\}_{i=1}^n, \ldots, \{x_{m,i} = y_{1,i}\}_{i=1}^n, Env_1\ \texttt{in}\ c\ t''_1\ \ldots\ t''_m)$. Applying (cpx) and (gc) to the last terms and using correctness of (cpx) and (gc), we obtain that this term is equivalent to t, and hence the claim of the corollary.

J Contextual Least Upper Bounds

We represent sequences $s_1, s_2, \ldots$ as $(s_i)_i$.

Definition 37. *Let $s_1 \leq_c s_2 \leq_c \ldots$ be an ascending chain. The expression s is a* least upper bound (*lub*) *of this chain, denoted $s = lub((s_i)_i)$, iff*
$\forall i : s_i \leq_c s$ *and for all r:* $(\forall i : s_i \leq_c r) \Rightarrow s \leq_c r$.

The expression s is a contextual least upper bound (*club*) *of this chain, denoted as $s = club((s_i)_i)$, iff*
$\forall C : C[s] = lub((C[s_i])_i)$.
This is denoted as $s = club((s_i)_i)$.

Note that *club* is unique up to $\sim_c$.
It would be more suggestive to write $C[(club(s_i))_i] = club((C[s_i])_i)$, which means continuity of contexts in analogy to the corresponding notion for complete partial orders.
In a paper by Mason, Smith, Talcott [MST96] there is an example which shows (for a different lambda-calculus) that not every *lub* is also a *club*. This can be reformulated as: "application is not continuous w.r.t. *lub*". Presumably, this example can be translated into our calculus. The definition of *club* enforces that all contexts (in particular applications) are continuous w.r.t. *club*.
The definition of *lub* and *club* is also required for sets of expressions:

Definition 38. *Let A be a set of expressions, and let t be an expression. Then*
$t = lub(A)$, *iff* $\forall a \in A : a \leq_c t$ *and* $\forall s : (\forall a \in A : a \leq_c s) \Rightarrow t \leq_c s$.
$t = club(A)$, *iff for all C:* $C[t] = lub(\{C[a] \mid a \in A\})$.

The following criterion and its improvement for reduction contexts is essential for using *club* as a tool.

Lemma 50. *Let $s_1 \leq_c s_2 \leq_c \ldots$ be an ascending chain and let s be an expression. Assume that the following holds:*

1. *For all $i : s_i \leq_c s$*
2. *For all contexts C: $C[s]\Downarrow \Rightarrow \exists i : C[s_i]\Downarrow$.*

Then $s = club((s_i)_i)$.

Proof. Let C, D be contexts. We will show that $D[s] = lub((D[s_i])_i)$. Let r be an expression with $\forall i : D[s_i] \leq_c r$. The assumption implies that if $CD[s]\Downarrow$, then there exists a j with $CD[s_j]\Downarrow$. Since $D[s_j] \leq_c r$ we have also $C[r]\Downarrow$. Hence $CD[s]\Downarrow \Rightarrow C[r]\Downarrow$. Since this holds for all contexts C, we have proved $D[s] \leq_c r$. This implies for all contexts $D : D[s] = lub((D[s_i])_i)$.

Lemma 51. *Let $s_1 \leq_c s_2 \leq_c \ldots$ be an ascending chain. Let s be an expression. Assume that the following holds:*

1. *For all* $i : s_i \leq_c s$
2. *For all reduction contexts* R: $R[s]\Downarrow \Rightarrow \exists i : R[s_i]\Downarrow$.

Then $s = \mathit{club}((s_i)_i)$.

Proof. We prove that the conditions of Lemma 50 hold. The technique is the same as in the proof of the context lemma. We prove that for a multicontext $C[\cdot,\ldots,\cdot]$, and for ascending chains $(s_{i,j})_j$, $i = 1,\ldots,n$, and for expressions s_i the following holds:
If $\forall i,j : s_{i,j} \leq_c s_i$, and for all reduction contexts R and all $i : R[s_i]\Downarrow \Rightarrow \exists j : R[s_{i,j}]\Downarrow$, then

$$C[s_1,\ldots,s_n]\Downarrow \Rightarrow \exists j : C[s_{1,j},\ldots,s_{n,j}]\Downarrow$$

We assume that there is a counterexample with a minimal number of normal order reductions of $C[s_1,\ldots,s_n]$ to WHNF, and among the minimal counterexamples, we select the minimal number of holes of C. Since it is a counterexample, the number of holes in C is > 0, and C is not a WHNF. There are two cases:

- The normal order redex is within C, i.e. no hole is in a reduction context. Then the first step of the normal order reduction of $C[s_1,\ldots,s_n] \xrightarrow{n} t_1$ may leave the holes or drop or copy some holes. For further arguments on the scopes of variables and the applications of renamings see the proof of the context lemma 2. We obtain a corresponding reduction for all i by reducing the same redex: $C[s_{1,i},\ldots,s_{n,i}] \xrightarrow{n} C'[s'_{1,i},\ldots,s'_{n',i}]$ for all i. Every pair $(s'_i,(s'_{i,j})_j)$ is the same as some pair $(s_i,(s_{i,j})_j)$ after an appropriate renaming of variables of the pairs (see the proof of Lemma 2). Since after the reduction, we do no longer have a counterexample, there is some i_0, such that $C'[s'_{1,i_0},\ldots,s'_{n',i_0}]\Downarrow$, hence $C[s_{1,i_0},\ldots,s_{n,i_0}]\Downarrow$, which is a contradiction.
- The second case is that the normal order reduction requires a part of some s_i. Then there is a hole of C that is in a reduction context in C. We assume it is the first one. Then let $C'[\ldots] := C[s_1,\cdot,\ldots]$. Since the number of holes is smaller, and since $C'[s_2,\ldots,s_n]\Downarrow$, we obtain that there is an i such that $C'[s_{2,i},\ldots,s_{n,i}]\Downarrow$, which means $C[s_1,s_{2,i},\ldots,s_{n,i}]\Downarrow$. The context $C[\cdot,s_{2,i},\ldots,s_{n,i}]$ is a reduction context, hence there is some i_0 such that $C[s_{1,i_0},s_{2,i},\ldots,s_{n,i}]\Downarrow$. Since $(s_{j,k})_k$ are ascending chains, we can choose the maximum i_1 of i_0 and i and obtain $C[s_{1,i_1},s_{2,i_1},\ldots,s_{n,i_1}]\Downarrow$, which is a contradiction. □