On the Safety 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

Technical Report Frank-19
Research group for Artificial Intelligence and Software Technology,
Institut für Informatik,
J.W.Goethe-Universität Frankfurt,

30.10.2004

Abstract. This paper proves correctness of Nöcker's method of strictness analysis, implemented for Clean, 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, which addresses correctness of the abstract reduction rules. Our method also addresses the cycle detection rules, which are the main strength of Nöcker's strictness analysis.
We reformulate Nöcker's strictness analysis algorithm in a higher-order lambda-calculus with `case`, constructors, `letrec`, and a non-deterministic choice operator $\oplus$ used as a union operator. Furthermore, the calculus is expressive enough to represent abstract constants like `Top` or *Inf*. The operational semantics is a small-step semantics and equality of expressions is defined by a contextual semantics that observes termination of expressions. The correctness of several reductions is proved using a context lemma and complete sets of forking and commuting diagrams. The proof is based mainly on an exact analysis of the lengths of normal order reductions. However, there remains a small gap: Currently, the proof for correctness of strictness analysis requires the conjecture that our behavioral preorder is contained in the contextual preorder. The proof is valid without referring to the conjecture, if no abstract constants are used in the analysis.

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 [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 a strictness analysis algorithm in [SSPS95].
We will reformulate Nöcker's strictness analysis algorithm in a higher-order lambda-calculus with `case`, constructors, `letrec`, constructors, `seq` and a non-deterministic choice operator $\oplus$ used as a union operator. `letrec` can describe recursive definitions and also explicitly treats the sharing inherent in lazy functional languages. The choice-operator $\oplus$ serves in forming sets of expressions.
The main part of the proof is to exhibit the properties of the reduction rules and other reductions, in particular their influence on the length of normal order reduction of some expression. The induction is on the number of essential steps of the (unique) normal order reduction of a given $\oplus$-free expression.
Let us consider two examples.
An expression `f` is called *strict* in argument i for arity n, iff the evaluation starting with `f` $t_1 \ldots t_{i-1}$ `Bot` $t_{i+1} \ldots, t_n$ will never yield a weak head normal form, where `Bot` 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: reducing `K Bot Top` gives `Bot` which indicates that `K` is indeed strict in its first argument.
A nontrivial example 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 Bot` using the rules of the calculus results either in `Bot` or in an expression that is essentially the same as `length Top Bot`. 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 values like `Top` or `Inf`. However, the syntax is slightly different from Nöcker's since usually there is a global `letrec` environment including all relevant function definitions.

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,LPJ96]), 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 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 this 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 the 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 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 LRA, 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 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:

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 – 15)
3. a description of the strictness analysis algorithm, its data structures and a proof of its correctness (sections 16 – 19).

The first part is concerned with describing the calculus for a call-by-need functional core language using sharing and investigating equivalences and lengths of normal order reductions. The calculus is extended by a non-deterministic choice-primitive $\oplus$ that is used to describe (infinite) sets of expressions, and since the core language permits `letrec`s, it is possible to encode abstract constants like $\top$ or *Inf* in the extended core language LRA. (Non-deterministic) reduction allows extracting the elements from these sets.
The core language provides `letrec`, the usual primitives like a weakly typed `case`, constructors, lambda, application, and `seq`, since programs in Clean or Haskell often use a `seq` or a similar primitive which would otherwise not be representable in the core language. The core language and the first treatment is borrowed mainly from [SS03], since this choice appears to be the very natural way for proving all the required equivalences. The `case`-primitive is slightly changed insofar as it is typed, which 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 6.4).
Contrary to Moran, Sands and Carlsson [MSC99] we allow arguments other than variables. Their restriction is fine for working with an abstract machine, also for most of the proofs of equivalences, but awkward in proofs of the properties of behavioral preorder and behavioral equivalence. 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 LRA are defined without constraints, the normal order reduction is then defined as a specific strategy uniquely determining the next sub-expression for reduction. The non-determinism comes only from the non-deterministic (choice) rule. The contextual equivalence is defined as usual where the only observation is successful termination of the evaluation of an expression.
We show that contextual equivalence is stable w.r.t. all deterministic 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 reduc-

tion sequence. We then study how these measures are affected by normal order reducing a $\oplus$-free expression.

The algorithm SAL is a reformulation of Nöcker's algorithm. It uses the knowledge of already computed strictness of functions. The main data structure is a directed graph of expressions, where the directed edges correspond to reductions or to cycle-checks. The expressions in the graph represent sets of terms in the (concrete, i.e. $\oplus$-free) core language, and the reductions either modify the abstract expressions, or if a $\oplus$ pops up and has to be reduced, the case analysis leads to a forking in the graph. This leads to a concise representation of unions, and it indicates that the 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 a 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 19.1 and Corollaries 19.3, 19.2) relies on arguments on the number of essential normal order reduction steps of expressions after reductions and transformations. It requires the union-theorem stating that the execution of a choice-reduction corresponds to the union of the concretizations. A further base is a theorem on the correctness of copying parts of a term that are deterministic.

The correctness proof of the strictness analysis algorithm also requires a conjecture (14.5) on the containment of our behavioral preorder in the contextual preorder. Our behavioral preorder is a generalization of the behavioral preorder for deterministic lambda calculi. The correctness proof is valid without the conjecture if no abstract constants like $\top$ are used (see Corollary 19.4 and 19.5).

A proof of the conjecture is possible, if Howe's proof method can be used also for non-deterministic calculi. A first adaptation for a non-deterministic calculus, but without a `case`, without constructors, and with a non-recursive `let` was successful [Man04]. The conjecture is used in Proposition 14.6, Proposition 16.6, and finally via the Proposition 16.6 also in the theorems and corollaries in section 19.1. We are optimistic that this gap can be closed in the near future.

4 Syntax of the Abstract Functional Core Language LRA

Our language, LRA, 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:

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

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, for every constructor $c_{T,i}, i = 1, \ldots, |T|$ of type T, there is exactly one alternative with a pattern of the form $(c_{T,i}\ y_1 \ldots y_n)$. The expressions $(c\ E_1 \ldots E_{\mathrm{ar}(c)}) \mid (\texttt{seq}\ E_1\ E_2) \mid (\texttt{case}_T\ E\ Alt_1 \ldots Alt_N) \mid (E_1\ E_2) \mid (\lambda V.E) \mid (\texttt{letrec}\ V_1 = E_1, \ldots V_n = E_n\ \texttt{in}\ E)$, $(E_1 \oplus E_2)$ are called *constructor application*, `seq`-*expression*, `case`-*expression*, *application*, *abstraction*, `letrec`-*expression*, or $\oplus$-expression respectively.
The constructs `case`, `seq`, $\oplus$ and the constructors $c_{T,i}$ can only occur in special syntactic constructions. Thus expressions where `case`, `seq`, $\oplus$ 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 notion of closed, open expressions and α-renamings. 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 also use the convention to omit parentheses in denoting nested applications: $(s_1 \ldots s_n)$ denotes $(\ldots(s_1\ s_2)\ldots s_n)$.

Definition 4.1. *(concrete language)*
We define the concrete language, LR*, as the fragment of the language without $\oplus$-expressions. The corresponding terms are also called* concrete terms*.*

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 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.
In the following we define different context classes and contexts, where we use a different text style for context classes and individual contexts.

Definition 4.2. *The class $\mathcal{C}$ of all* contexts *is defined as follows.*

$$\begin{aligned}\mathcal{C} ::= \ & [\cdot] \mid (\mathcal{C}\ E) \mid (E\ \mathcal{C}) \mid \ (\texttt{seq}\ E\ \mathcal{C}) \mid (\texttt{seq}\ \mathcal{C}\ E) \mid \lambda x.\mathcal{C} \\ & \mid (\texttt{case}_T\ \mathcal{C}\ alts) \mid (\texttt{case}_T\ E\ Alt_1, \ldots, (Pat\ \rightarrow\ \mathcal{C}), \ldots, Alt_n) \\ & \mid (c\ E_1 \ldots E_{i-1}\ \mathcal{C}\ E_{i+1} \ldots E_{\mathrm{ar}(c)}) \\ & \mid (\mathcal{C} \oplus E) \mid (E \oplus \mathcal{C}) \mid \\ & \mid (\texttt{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ \mathcal{C}) \\ & \mid (\texttt{letrec}\ x_1 = E_1, \ldots, x_{i-1} = E_{i-1}, x_i = \mathcal{C}, x_{i+1} = E_{i+1}, \ldots, x_n = E_n\ \texttt{in}\ E)\end{aligned}$$

The main depth *of a context C is the depth of the hole in the context C.*

Definition 4.3. *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 $t = R^-[t_0]$, we say R^- is maximal (for t)*, iff there is no larger weak reduction context with this property. For a term t with $t = C[t_0]$, 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 index j of involved bindings is maximal.*

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 4.4. *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:

$$\begin{aligned}\mathcal{S} ::= \; & [\cdot] \mid (\mathcal{S}\ E) \mid (E\ \mathcal{S}) \\ & \mid (\mathtt{case}_T\ \mathcal{S}\ \mathit{alts}) \mid (\mathtt{case}_T\ E\ \ldots (p \to \mathcal{S}) \ldots) \\ & \mid (c\ E_1 \ldots E_{i-1}\ \mathcal{S}\ E_{i+1} \ldots E_{\mathrm{ar}(c)}) \\ & \mid (\mathtt{seq}\ \mathcal{S}\ E) \mid (\mathtt{seq}\ E\ \mathcal{S}) \\ & \mid (\mathcal{S} \oplus E) \mid (E \oplus \mathcal{S}) \\ & \mid (\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ \mathcal{S}) \mid (\mathtt{letrec}\ \ldots, x_i = \mathcal{S}, \mathit{Env}\ \mathtt{in}\ E)\end{aligned}$$

Note that every reduction context is also a surface context.

Definition 4.5. *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:*

$$\begin{aligned}\mathcal{AS} ::= \; & [\cdot] \mid (\mathcal{AS}\ E) \mid (E\ \mathcal{AS}) \\ & \mid (\mathtt{case}_T\ \mathcal{AS}\ \mathit{alts}) \\ & \mid (c\ E_1 \ldots E_{i-1}\ \mathcal{AS}\ E_{i+1} \ldots E_{\mathrm{ar}(c)}) \\ & \mid (\mathtt{seq}\ \mathcal{AS}\ E) \mid (\mathtt{seq}\ E\ \mathcal{AS}) \\ & \mid (\mathcal{AS} \oplus E) \mid (E \oplus \mathcal{AS}) \\ & \mid (\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ \mathcal{AS}) \mid (\mathtt{letrec}\ \mathit{Env}, x_i = \mathcal{AS}, \ldots\ \mathtt{in}\ E)\end{aligned}$$

Definition 4.6. *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`*:*

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

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

Definition 4.8. *The (base) reduction rules for the language* LRA *are defined in figures 1 and 2. The union of (llet-in) and (llet-e) is called (llet), the union of the rules (choice-l) and (choice-r) is called (choice), 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)	$((\lambda x.s)\ r) \rightarrow (\texttt{letrec}\ x = r\ \texttt{in}\ s)$
(cp-in)	$(\texttt{letrec}\ x_1 = v, x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \ \texttt{in}\ C[x_m])$
	$\rightarrow (\texttt{letrec}\ x_1 = v, x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \ \texttt{in}\ C[v])$
	where v is an abstraction
(cp-e)	$(\texttt{letrec}\ x_1 = v, x_2 = x_1, \ldots, x_m = x_{m-1}, Env, y = C[x_m]\ \texttt{in}\ r)$
	$\rightarrow (\texttt{letrec}\ x_1 = v, x_2 = x_1, \ldots, x_m = x_{m-1}, Env, y = C[v]\ \texttt{in}\ r)$
	where v is an abstraction
(llet-in)	$(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ (\texttt{letrec}\ y_1 = t_1, \ldots, y_m = t_m\ \texttt{in}\ r))$
	$\rightarrow (\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n, y_1 = t_1, \ldots, y_m = t_m\ \texttt{in}\ r)$
(llet-e)	$(\texttt{letrec}\ x_1 = s_1, \ldots, x_i =$
	$\quad (\texttt{letrec}\ y_1 = t_1, \ldots, y_m = t_m\ \texttt{in}\ s_i), \ldots, x_n = s_n\ \texttt{in}\ r)$
	$\rightarrow (\texttt{letrec}\ x_1 = s_1, \ldots, x_i = s_i, \ldots, x_n = s_n, y_1 = t_1, \ldots, y_m = t_m\ \texttt{in}\ r)$
(lapp)	$((\texttt{letrec}\ Env\ \texttt{in}\ t)\ x) \rightarrow (\texttt{letrec}\ Env\ \texttt{in}\ (t\ x))$
(lcase)	$(\texttt{case}_T\ (\texttt{letrec}\ Env\ \texttt{in}\ t)\ alts) \rightarrow (\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{case}_T\ t\ alts))$
(seq-c)	$(\texttt{seq}\ v\ t) \rightarrow \quad t \qquad$ if v is a value
(seq-in)	$(\texttt{letrec}\ x_1 = v, x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ C[(\texttt{seq}\ x_m\ t)])$
	$\rightarrow \quad (\texttt{letrec}\ x_1 = v, x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ C[t])$
	if v is a value
(seq-e)	$(\texttt{letrec}\ x_1 = v, x_2 = x_1, \ldots, x_m = x_{m-1}, Env, y = C[(\texttt{seq}\ x_m\ t)]\ \texttt{in}\ r)$
	$\rightarrow \quad (\texttt{letrec}\ x_1 = v, x_2 = x_1, \ldots, x_m = x_{m-1}, Env, y = C[t]\ \texttt{in}\ r)$
	if v is a value
(lseq)	$(\texttt{seq}\ (\texttt{letrec}\ Env\ \texttt{in}\ s)\ t) \rightarrow (\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{seq}\ s\ t))$
(choice-l)	$(s \oplus t) \rightarrow s$
(choice-r)	$(s \oplus t) \rightarrow t$

Fig. 1. Reduction rules, part a

Note that the reduction rules generate only syntactically correct expressions, since the context definitions are made in an appropriate way.

Remark 4.9. 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 x ...` is to

be evaluated, then the case and constructor application must cooperate. It is not possible to copy every constructor application into the position of x, since it may contain choice-expressions. 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 deterministic reductions are correct program transformations. The current definition of (case) is borrowed from FUNDIO [SS03].

(case-c)	$(\texttt{case}\ (c_i\ t_1 \ldots t_n)\ \ldots((c_i\ y_1 \ldots y_n) \to t)\ldots)$
	$\to \quad (\texttt{letrec}\ y_1 = t_1 \ldots y_n = t_n\ \texttt{in}\ t)$
	where $n = \mathrm{ar}(c_i) \geq 1$
(case-c)	for the case $\mathrm{ar}(c_i) = 0$
	$(\texttt{case}\ c_i\ \ldots\ (c_i \to t)\ldots) \to t$
(case-in)	$\texttt{letrec}\ x_1 = (c_i\ t_1 \ldots t_n),$ where $n = \mathrm{ar}(c_i) \geq 1$
	$x_2 = x_1, \ldots x_m = x_{m-1}, \ldots$
	$\texttt{in}\quad C[\texttt{case}\ x_m\ \ldots((c_i\ z_1 \ldots z_n)\ldots \to t)]$
	$\longrightarrow$
	$\texttt{letrec}\ x_1 = (c_i\ y_1 \ldots y_n), y_1 = t_1, \ldots y_n = t_n,$
	$x_2 = x_1, \ldots x_m = x_{m-1}, \ldots$
	$\texttt{in}\quad C[(\texttt{letrec}\ z_1 = y_1, \ldots, z_n = y_n\ \texttt{in}\ t)]$
	where y_i are fresh variables
(case-in)	for the case $\mathrm{ar}(c_i) = 0$
	$\texttt{letrec}\ x_1 = c_i, x_2 = x_1, \ldots x_m = x_{m-1}, \ldots$
	$\texttt{in}\quad C[\texttt{case}\ x_m\ \ldots\ (c_i \to t)\ldots]$
	$\longrightarrow$
	$\texttt{letrec}\ x_1 = c_i, x_2 = x_1, \ldots x_m = x_{m-1}, \ldots$
	$\texttt{in}\quad C[t]$
(case-e)	$\texttt{letrec}\ x_1 = (c_i\ t_1 \ldots t_n),$ where $n = \mathrm{ar}(c_i) \geq 1$
	$x_2 = x_1, \ldots x_m = x_{m-1}, \ldots$
	$u = C[\texttt{case}\ x_m\ \ldots((c_i\ z_1 \ldots z_n) \to r_1)\ldots]$
	$\texttt{in}\quad r_2$
	$\longrightarrow$
	$\texttt{letrec}\ x_1 = (c_i\ y_1 \ldots y_n), y_1 = t_1, \ldots y_n = t_n,$
	$x_2 = x_1, \ldots x_m = x_{m-1}, \ldots$
	$u = C[(\texttt{letrec}\ z_1 = y_1, \ldots, z_n = y_n\ \texttt{in}\ r_1)]$
	$\texttt{in}\quad r_2$
	where y_i are fresh variables
(case-e)	for the case $\mathrm{ar}(c_i) = 0$
	$\texttt{letrec}\ x_1 = c_i, x_2 = x_1, \ldots x_m = x_{m-1}, \ldots$
	$u = C[\texttt{case}\ x_m\ \ldots\ (c_i \to r_1)\ldots]$
	$\texttt{in}\quad r_2$
	$\longrightarrow$
	$\texttt{letrec}\ x_1 = c_i, x_2 = x_1, \ldots x_m = x_{m-1}, \ldots$
	$u = C[r_1]$
	$\texttt{in}\quad r_2$

Fig. 2. Reduction rules, part b

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 5.1.
The following labeling algorithm will detect the position where the normal order reduction will start evaluation by applying a reduction rule. Use three labels: $e0, e1, e$, where $e0$ means evaluation of the top term, $e1$ means evaluation of a subterm, and e may be $e0$ as well as $e1$. The label $e1$ prevents going deeper into `letrec`-expressions. For a term s the labeling algorithm starts with s^{e0}
The labeling algorithm:

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

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 the 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, then there is no normal order redex and hence no normal order reduction.

Definition 5.1. *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* $(\texttt{letrec}\ \mathit{Env}_1\ \texttt{in}\ t'')$*, 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) *$R = R_0[(\texttt{case}_T\ [\cdot]\ \mathit{alts})]$. Reduce $(\texttt{case}_T\ t'\ \mathit{alts})$ using (lcase).*
   (d) *$R = (\texttt{letrec}\ \mathit{Env}_2\ \texttt{in}\ [\cdot])$. Reduce t using (llet-in) by flattening t' resulting in $(\texttt{letrec}\ \mathit{Env}_1, \mathit{Env}_2\ \texttt{in}\ t'')$.*
   (e) *$R = (\texttt{letrec}\ x = [\cdot], \mathit{Env}_2\ \texttt{in}\ t''')$. Reduce t using (llet-e) by flattening t' resulting in $(\texttt{letrec}\ x = t'', \mathit{Env}_1, \mathit{Env}_2\ \texttt{in}\ t''')$.*
2. *t' is a value. There are the following cases:*
   (a) *$R = R_0[\texttt{case}_T\ [\cdot]\ \ldots]$, t' is a constructor application, and the top constructor of t' belongs to type T. Then apply (case-c) to $(\texttt{case}_T\ t'\ \ldots)$,*
   (b)

i. We have

$$R = \texttt{letrec } x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, Env \\ \texttt{in } R_0^-[\texttt{case}_T\ x_m\ (c\ y_1 \ldots y_n \to r)\ alts],$$

$t' = c\ t_1 \ldots t_n$, and the top constructor c belongs to T.
Then apply (case-in) resulting in

$$\texttt{letrec } x_1 = c\ z_1 \ldots z_n, x_2 = x_1, \ldots, x_m = x_{m-1}, z_1 = t_1, \ldots, z_n = t_n, Env \\ \texttt{in } R_0^-[(\texttt{letrec } y_1 = z_1, \ldots, y_n = z_n \texttt{ in } r)]$$

ii. We have

$$R = \texttt{letrec } x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, Env \\ \texttt{in } R_0^-[\texttt{case}_T\ x_m\ (c \to r)\ alts],$$

$t' = c$, and the top constructor c belongs to T.
Then apply (case-in) resulting in

$$\texttt{letrec } x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, Env \\ \texttt{in } R_0^-[r]$$

(c)

i. We have

$$R = \texttt{letrec } x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, Env \\ y = R_0^-[\texttt{case}_T\ x_m\ (c\ y_1 \ldots y_n \to r)\ alts]\ \texttt{in } r',$$

$t' = c\ t_1 \ldots t_n$, the top constructor c belongs to T, and y is in a reduction context.
Then apply (case-e) resulting in

$$\texttt{letrec } x_1 = c\ z_1 \ldots z_n, x_2 = x_1, \ldots, x_m = x_{m-1}, z_1 = t_1, \ldots, z_n = t_n, \\ Env, y = R_0^-[(\texttt{letrec } y_1 = z_1, \ldots, y_n = z_n \texttt{ in } r)]\ \texttt{in } r'$$

ii. We have

$$R = \texttt{letrec } x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, Env \\ y = R_0^-[\texttt{case}_T\ x_m\ (c \to r)\ alts]\ \texttt{in } r',$$

$t' = c$, the top constructor c belongs to T, and y is in a reduction context.
Then apply (case-e) resulting in

$$\texttt{letrec } x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, \\ Env, y = R_0^-[r]\ \texttt{in } r'$$

(d) $R = R_0[([\cdot]\ s)]$ *where R_0 is a reduction context and t' is an abstraction.*
Then apply (lbeta) to $(t'\ s)$.

(e) $R = (\texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ R_0^-[x_m])$ *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* $(\texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ R_0^-[t'])$.

(f) $R = (\texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, Env, y = R_0^-[x_m]\ \texttt{in}\ r)$ *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* $(\texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, Env, y = R_0^-[t']\ \texttt{in}\ r)$.

(g) $R = R_0[(\texttt{seq}\ [\cdot]\ r)]$. *Then apply (seq-c) to* $(\texttt{seq}\ t'\ r)$ *resulting in* r.

(h) $R = (\texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ R_0^-[(\texttt{seq}\ x_m\ r)])$, *and* t' *is a constructor application. Then apply (seq-in) resulting in* $(\texttt{letrec}\ x_1 = t', x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ R_0^-[r])$.

(i) $R = (\texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, Env, y = R_0^-[(\texttt{seq}\ x_m\ r)]\ \texttt{in}\ r')$ *where* y *is in a reduction context, and* t' *is a constructor application. Then apply (seq-e) resulting in* $(\texttt{letrec}\ x_1 = t', x_2 = x_1, \ldots, x_m = x_{m-1}, Env, y = R_0^-[r]\ \texttt{in}\ r')$.

3. t' *is a* $\oplus$*-expression. Then apply (choice-r) or (choice-l) to* t'.

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.
The notion of weak head normal form will be required.

Definition 5.2. *A* weak head normal form (WHNF) *is one of the two 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* $(\texttt{letrec}\ x_1 = c\ t_1 \ldots t_{\mathrm{ar}(c)}, x_2 = x_1, \ldots x_m = x_{m-1}, Env\ \texttt{in}\ x_m)$

Lemma 5.3. *For every term* t*:*
if t *has a normal order redex, then this redex is unique.*
If the normal order redex is not a $\oplus$*-expression, then the normal order reduction is unique.*

Definition 5.4. *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.
The set of terminating normal order reductions of an expression t *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 abbreviated as* `Bot`*.*

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 $(\mathtt{case}_T(\lambda x.x)\ Alts)$ 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" $(\mathtt{letrec}\ x = x\ \mathtt{in}\ x)$. The maximal reduction context for this term is $(\mathtt{letrec}\ x = [\cdot]\ \mathtt{in}\ x)$. It is easily seen that there is no normal order reduction defined for this term.
A term that has a non-terminating normal order reduction is $(\mathtt{letrec}\ z = \lambda x.x\ x\ \mathtt{in}\ (z\ z))$; the start of the infinite reduction being $(\mathtt{letrec}\ z = \lambda x.x\ x\ \mathtt{in}\ (z\ z)) \xrightarrow{n,cp} (\mathtt{letrec}\ z = \lambda x.x\ x\ \mathtt{in}\ ((\lambda x.x\ x)\ z)) \xrightarrow{n,lbeta} (\mathtt{letrec}\ z = \lambda x.x\ x\ \mathtt{in}\ (\mathtt{letrec}\ x_1 = z\ \mathtt{in}\ (x_1\ x_1))) \xrightarrow{n,llet} (\mathtt{letrec}\ z = \lambda x.x\ x, x_1 = z\ \mathtt{in}\ (x_1\ x_1))$.

6 Contextual Equivalence

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

Definition 6.1. *(contextual preorder and equivalence) Let s, t be terms. Then:*

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

Note that we permit contexts such that $C[s]$ may be an open term.
Later in Proposition 11.6 we will 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 6.2. *$\leq_c$ 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 notion from denotational semantics.

Definition 6.3. *An expression s is* strict, *iff* $(s\ \texttt{Bot}) \sim_c \texttt{Bot}$.
An expression s is strict in the i^{th} argument for arity n, *iff* $1 \leq i \leq n$ *and for all closed expressions* $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots t_n$*:*
$(s\ t_1 \ldots t_{i-1}\ \texttt{Bot}\ t_{i+1} \ldots t_n) \sim_c \texttt{Bot}$.
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* `Bot`*, we have* $s' \sim_c \texttt{Bot}$.
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 programs.

6.1 A Difference Between LR and an Untyped Core Language

Example 6.4. 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 CFP without types, but with `letrec`s, constructors, and abstractions. The language CFP 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 CFP.
Now define the following functions:

```
s0 = \f -> if (f True) then (if (f Nil) then Bot else True)
                       else Bot
t0 = \f -> if (f Nil)  then  (if (f True) then Bot else False)
                       else Bot
```

We claim that s_0, t_0 cannot be distinguished in LR, since `(f True)` and `(f 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 `case` is typed. That this reasoning implies $s_0 \sim_c t_0$ can be derived from the conjecture 14.5.
The expressions s_0, t_0 can be distinguished in CFP, 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 CFP is only the restricted typing. The reason is that typing restricts the number of contexts in LR in contrast to CFP.

7 Context Lemma

The Context Lemma restricts the criterion for contextual equivalence to reduction contexts. This 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]

Lemma 7.1. *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, and every occurrence of a hole is mentioned 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, or in the expressions that is removed by a reduction (choice), 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 itself 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. □

7.1 Deterministic Reductions: Easy Cases

Non-normal order reductions for the language LRA 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 $i\mathcal{S}$.

7.2 The reductions (case-c), (seq-c), (lbeta), (lapp), (lcase), (lseq)

Lemma 7.2. *Every a-reduction in a reduction context where* $a \in \{$*(case-c), (seq-c), (choice), (lbeta), (lapp), (lcase), (lseq)*$\}$ *is a normal order reduction.*

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

Proposition 7.3. *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 7.1. It is sufficient to consider $R[s]$ and $R[t]$. From $s \xrightarrow{a} t$ and Lemma 7.2 it follows that $R[s] \xrightarrow{n} R[t]$. Since the reductions are deterministic, it follows $R[s]{\Downarrow}$ iff $R[t]{\Downarrow}$. Now we apply the context lemma. □

The (choice)-reductions break contextual equivalence. 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.

7.3 Monotonicity of choice-Reductions

Theorem 7.4. *If* $s \xrightarrow{a} t$ *by a choice-reduction a then* $s \geq_c t$.

Proof. By Lemma 7.2, there are no internal (choice)-reductions in reduction contexts. The proof is similar to the proof in Proposition 7.3. We use the context lemma 7.1. It is sufficient to consider $R[s]$ and $R[t]$ for an arbitrary reduction context R. From $s \xrightarrow{choice-r} t$ and Lemma 7.2 it follows that $R[s] \xrightarrow{n,choice-r} R[t]$. It follows that $R[t]{\Downarrow} \Rightarrow R[s]{\Downarrow}$. Now we apply the context lemma and obtain $t \leq_c s$. □

7.4 A Macro-Step (llet)-Reduction

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

Lemma 7.5. *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* $(\texttt{letrec}\ Env_R, x = R'[(\texttt{letrec}\ Env\ \texttt{in}\ t')]\ \texttt{in}\ r) \xrightarrow{n,lll,+} (\texttt{letrec}\ Env_R, Env, x = R'[t']\ \texttt{in}\ r)$, *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 7.6. *We define the $\xrightarrow{n,lll,*}$-reduction as in Lemma 7.5 which shifts* `letrec`*-environments that are in reductions contexts to the top of the expression as (n,mll).*

8 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 4.8. In the following definition we describe transformations on reduction sequences. Therefore we use the notation

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

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 LRA.
The above transformation rule can be applied to the prefix of the reduction sequence RED, if the prefix is: $s \xrightarrow{iX,red} t_1 \xrightarrow{n,a_1} \ldots t_k \xrightarrow{n,a_k} t$. 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

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

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

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

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 8.1.

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

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

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*

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

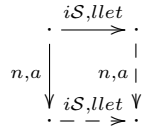
where $k, k' \geq 0, m \geq 1$*, such that for every reduction sequence* $t_h \xleftarrow{n} \ldots t_2 \xleftarrow{n} t_1 \xrightarrow{iX,red} t_0$*, where* t_h *is a WHNF, at least one of the transformation rules from the set is applicable to a suffix of the sequence. In the special case that* $h =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

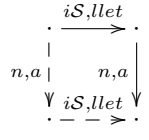
As a notation, we also use the * and +-notation of regular expressions for the diagrams. The interpretation is obvious and is intended to stand for an infinite set accordingly constructed.

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.

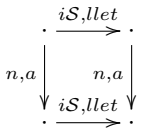
For example, the forking diagram $\xleftarrow{n,a} \cdot \xrightarrow{i\mathcal{S},llet} \rightsquigarrow \xrightarrow{i\mathcal{S},llet} \cdot \xleftarrow{n,a}$ is represented as



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.



Note, however, that in cases of reduction diagrams for (choice), the twofold interpretation as forking and commuting diagrams may be incorrect.

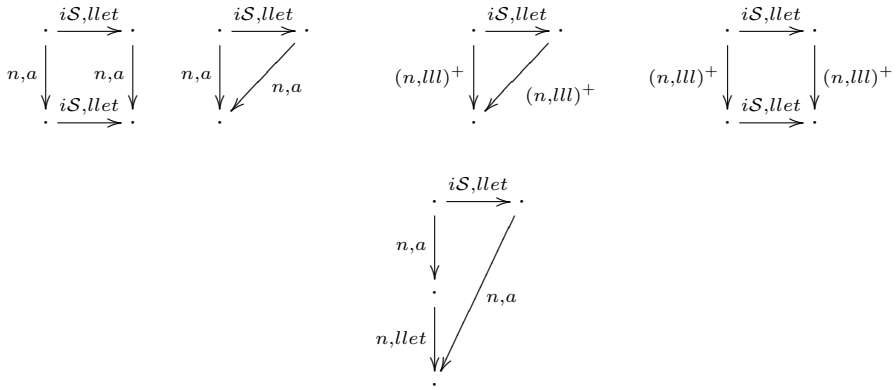
9 Diagrams for (llet), (seq) and (cp)

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.

9.1 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 9.1. *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:

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

The corresponding complete set of forking diagrams is:

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

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, (seq)-reduction or a (choice)-reduction. Lemma 7.5 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. Two typical cases are ($\texttt{letrec}\ x = (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ s)\ \texttt{in}\ t \oplus x$) and ($\texttt{letrec}\ x = (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ s)\ \texttt{in seq True}\ x$). □

Lemma 9.2. *If $s \xrightarrow{i,lll} t$, then s is a WHNF iff t is a WHNF.*

Proposition 9.3. *If $s \xrightarrow{llet} t$, then $s \sim_c t$.*

Proof. By the context lemma 7.1, 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 normal order reduction sequence to WHNF of $R[s]$, we can construct a normal order reduction of $R[t]$ to WHNF, and vice versa.

1. If $R[s]{\Downarrow}$, then we by induction on the length of the normal order reduction *Red* of $R[s]$ to a WHNF, that there is also a normal order reduction 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 9.2. If *Red* is not trivial, then the complete set of forking diagrams in lemma 9.1 provides all cases. Let $\mathit{Red} = R[s] \xrightarrow{n} s' \cdot \mathit{Red}'$. 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 $R[t]$.

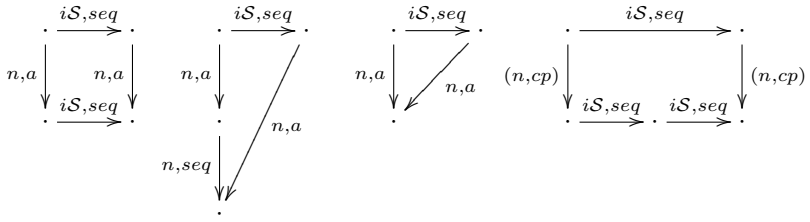
2. If $R[t]\Downarrow$, then we use similar methods. We apply induction on the number of normal order reduction steps of $R[t]$ to a WHNF using the complete set of commuting diagrams in lemma 9.1. In the base case we use Lemma 9.2.

□

9.2 Equivalence of (seq)

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

Lemma 9.4. *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, a (seq)-reduction or a (choice)-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 9.5. *If $s \xrightarrow{i\mathcal{S},seq} t$, then s is a WHNF iff t is a WHNF.*

Proposition 9.6. *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 9.4. 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 9.5.

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 9.4 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 9.5. and remove this reduction. Finally, we obtain a normal order reduction for $R[s]$.

□

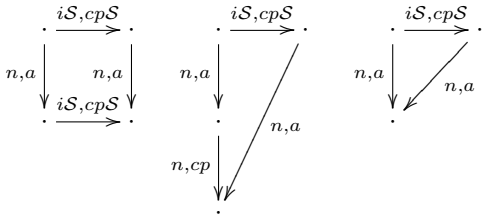
9.3 Correctness of (cp)

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.

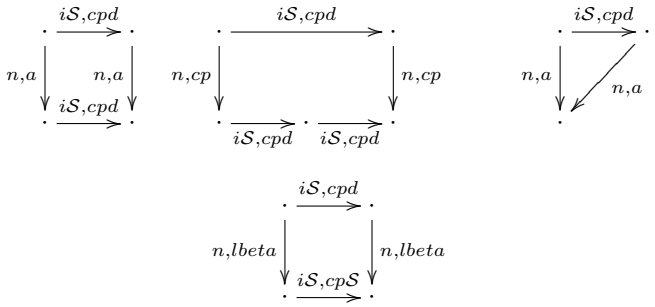
(cpd) = (cp) where the position of the replaced variable is not in a surface context.

Lemma 9.7. *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 9.8. *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 9.9. *If* $s \xrightarrow{i\mathcal{S},cp} t$*, then* s *is a WHNF iff* t *is a WHNF.*

Proposition 9.10. *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 9.8 and 9.7.
   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 9.9.

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 $s \sim_c t$.

□

10 Equivalence of Other Reductions and (case)

We show in this section that the extra reductions (gc), (cpx), (cpax), (cpcx), (xch), (ucp) and (lwas) are correct program transformations in the calculus LRA. 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.

Definition 10.1. *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).*

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 10.2. *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* $\texttt{lrdepth}(s,t)$ *of all* `letrec`*-subexpressions s of t, where* `lrdepth` *is defined as follows:*

$$\texttt{lrdepth}(s,s) = 0$$

$$\texttt{lrdepth}(s, C_1[C_2[s]]) = \begin{cases} 1 + \texttt{lrdepth}(s, C_2[s]) & \text{if } C_1 \text{ is a context of main depth 1, and not a } \texttt{letrec} \\ \texttt{lrdepth}(s, C_2[s]) & \text{if } C_1 \text{ is a context of main depth 1, and it is a } \texttt{letrec} \end{cases}$$

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

Proposition 10.3. *The reduction (lll) is terminating, I.e. there are no infinite reductions sequences consisting only of (lll)-reductions.*

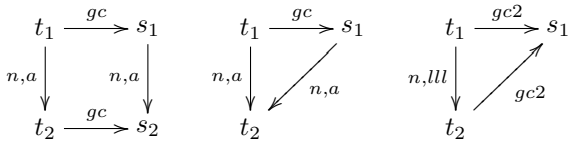
Proof. This holds, since $t_1 \xrightarrow{lll} t_2$ implies $\mu_{lll}(t_1) > \mu_{lll}(t_2)$, and the ordering induced by the measure is well-founded. □

10.1 Correctness of (gc)

Lemma 10.4. *A complete set of commuting and forking diagrams for $(\mathcal{S}, gc)$ can be read off the following set of graphical diagrams:*

(gc1)	$(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n, Env\ \texttt{in}\ t) \to (\texttt{letrec}\ Env\ \texttt{in}\ t)$
	if for all $i : x_i$ does not occur in Env nor in t
(gc2)	$(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t) \to t$
	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)	$(\texttt{letrec}\ x = c\ t_1 \ldots t_m, Env\ \texttt{in}\ C[x])$
	$\to (\texttt{letrec}\ x = c\ y_1\ \ldots\ y_m, y_1 = t_1, \ldots, y_m = t_m, Env\ \texttt{in}\ C[c\ y_1\ \ldots\ y_m])$
(cpcx-e)	$(\texttt{letrec}\ x = c\ t_1\ \ldots\ t_m, z = C[x], Env\ \texttt{in}\ t)$
	$\to (\texttt{letrec}\ x = c\ y_1\ \ldots\ y_m,$
	$y_1 = t_1, \ldots, y_m = t_m, z = C[c\ y_1\ \ldots\ y_m], Env\ \texttt{in}\ t)$
(abs)	$(\texttt{letrec}\ x = (c\ t_1 \ldots t_n), Env\ \ \texttt{in}\ s)$
	$\to (\texttt{letrec}\ x = c\ x_1 \ldots x_n, x_1 = t_1, \ldots x_n = t_n, Env\ \ \texttt{in}\ s)$
	where $n \geq 1$
(abse)	$(c\ t_1 \ldots t_n)$
	$\to (\texttt{letrec}\ x_1 = t_1, \ldots x_n = t_n\ \texttt{in}\ c\ x_1 \ldots x_n)$
	where $n \geq 1$
(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 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 4.6)

Fig. 3. Extra Reduction Rules



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 (choice) or 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 10.5. *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 10.6. *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 9.10, we have only to ensure that transforming a terminating normal order reduction of $R[t]$ using the diagrams in Lemma 10.4 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 10.5. In constructing from a reduction $R[t] \xrightarrow{gc} R[t'] \xrightarrow{n,*} t_0$ a terminating normal order reduction for $R[t']$, Proposition 10.3 shows that there there are only finitely many repeated applications of diagram 3.

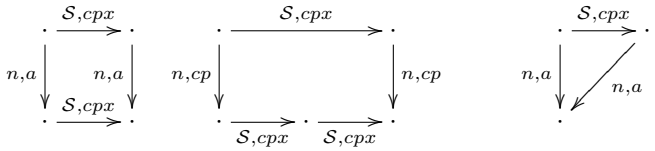
□

10.2 Equivalence of (cpx)

Note that the reduction $\xrightarrow{\mathcal{R},cpx}$ may not terminate:
$\texttt{letrec}\ x = y, y = x\ \texttt{in}\ C[x] \xrightarrow{\mathcal{R},cpx} \texttt{letrec}\ x = y, y = x\ \texttt{in}\ C[y] \xrightarrow{\mathcal{R},cpx} \texttt{letrec}\ x = y, y = x\ \texttt{in}\ C[x]$.

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

Lemma 10.7. *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), (seq), or (choice). An example for the last case is

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

□

Lemma 10.8. *If* $s \xrightarrow{\mathcal{S},cpx} t$*, then* s *is a WHNF iff* t *is a WHNF.*

Proposition 10.9. *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 10.7 and 10.8.
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. □

10.3 Equivalence of the reduction rule (xch)

Lemma 10.10. *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:

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

□

Lemma 10.11. *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 10.12. *If* $t \xrightarrow{xch} t'$*, then* t *is a WHNF iff* t' *is a WHNF.*

Proposition 10.13. *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

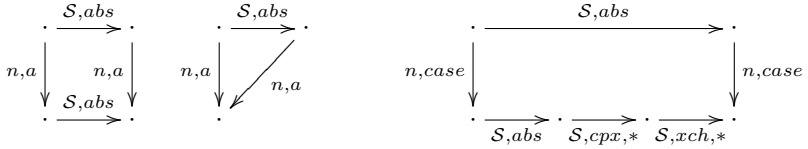
$$\begin{array}{ll} & (\texttt{letrec}\ x = t, y = x, Env\ \texttt{in}\ r) \\ \xrightarrow{cpax} & (\texttt{letrec}\ x = t[x/y], y = x, Env[x/y]\ \texttt{in}\ r[x/y]) \\ \xrightarrow{gc} & (\texttt{letrec}\ x = t[x/y], Env[x/y]\ \texttt{in}\ r[x/y]) \\ =_\alpha & (\texttt{letrec}\ y = t[y/x], Env[y/x]\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

□

and from the correctness of (gc) and (cpx); see Proposition 10.6, 10.9. □

10.4 Equivalence of (abs)

Lemma 10.14. *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:

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

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

Lemma 10.15. *If* $s \xrightarrow{\mathcal{S},abs} t$*, then* s *is a WHNF iff* t *is a WHNF.*

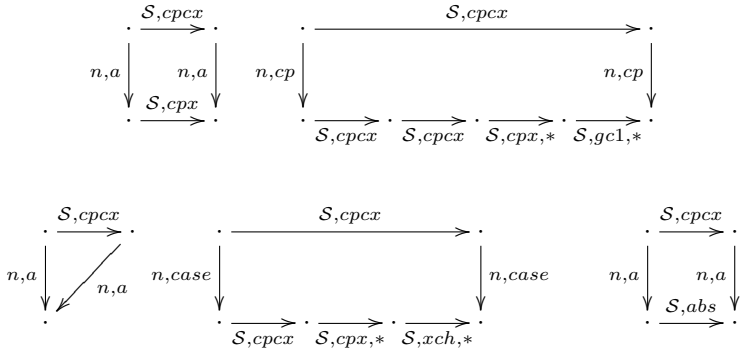
Proposition 10.16. *The reduction (abs) is a correct program transformation.*

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 10.14, 10.7, and 10.10.
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 10.14, 10.7, and 10.11. □

10.5 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 10.17. *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), (choice), or (seq).

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

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

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

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

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

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

□

Lemma 10.18. *If $s \xrightarrow{\mathcal{S},cpcx} t$, then s is a WHNF iff t is a WHNF.*

Proposition 10.19. *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 10.17, 10.10, 10.7, 10.4, and 10.14 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 10.4 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 10.8, 10.5, 10.15, and 10.18 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). □

10.6 Correctness of (case)-Reductions

Proposition 10.20. *The reductions (case-in) and (case-e) are correct program transformations.*

Proof. Propositions 7.3 and 9.3 show that (case-c) is a correct program transformation. From Lemmas 10.19, 10.9, 10.6, and 9.10, we obtain that (cpcx), (cpx), and (gc) 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 (cpS), (cpcx), (case-c), (cpx), and (gc). 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.
The base case is:

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

The result after a (case-in) is:

$$\xrightarrow{case-in} (\texttt{letrec}\ x = c\ y, y = t, Env\ \texttt{in}\ C[(\texttt{letrec}\ z = y\ \texttt{in}\ s)])$$

We show the induction for a short variable chain:

$$\begin{array}{ll} & (\texttt{letrec}\ x_1 = c\ t, x_2 = x_1, Env\ \texttt{in}\ C[\texttt{case}\ x_1\ (c\ z \to s)\ alts]) \\ \xrightarrow{cpcx} & (\texttt{letrec}\ x_1 = c\ y, y = t, x_2 = c\ y, Env\ \texttt{in}\ C[\texttt{case}\ x_n\ (c\ z \to s)\ \ alts]) \\ \xrightarrow{case-in} & (\texttt{letrec}\ x_1 = c\ y, y = t, x_2 = c\ y_2, y_2 = y, Env\ \texttt{in}\ C[(\texttt{letrec}\ z = y_2\ \texttt{in}\ s)]) \\ \xrightarrow{cpx,cpx,gc} & (\texttt{letrec}\ x_1 = c\ y, y = t, x_2 = c\ y, Env\ \texttt{in}\ C[(\texttt{letrec}\ z = y\ \texttt{in}\ s)]) \end{array}$$

The other case is:

$$\begin{array}{ll} \xrightarrow{case-in} & (\texttt{letrec}\ x_1 = c\ y, y = t, x_2 = x_1, Env\ \texttt{in}\ C[(\texttt{letrec}\ z = y\ \texttt{in}\ s)]) \\ \xrightarrow{cpcx} & (\texttt{letrec}\ x_1 = c\ y', y' = y, y = t, x_2 = c\ y', Env\ \texttt{in}\ C[(\texttt{letrec}\ z = y\ \texttt{in}\ s)]) \\ \xrightarrow{cpx,cpx,gc} & (\texttt{letrec}\ x_1 = c\ y, y = t, x_2 = c\ y, Env\ \texttt{in}\ C[(\texttt{letrec}\ z = y\ \texttt{in}\ s)]) \end{array}$$

□

Proposition 10.21. *The reduction (case) is a correct program transformation.*

Proof. Follows from Proposition 10.20 and 7.3. □

10.7 Summary of Properties

Theorem 10.22. *The reductions (cpcx), (cpx), (abs) and (gc) keep contextual equivalence. I.e. whenever* $t \xrightarrow{a} t'$*, with* $a \in \{cpcx, cpx, abs, gc\}$*, then* $t \sim_c t'$*.*

The remaining reduction will be treated below.

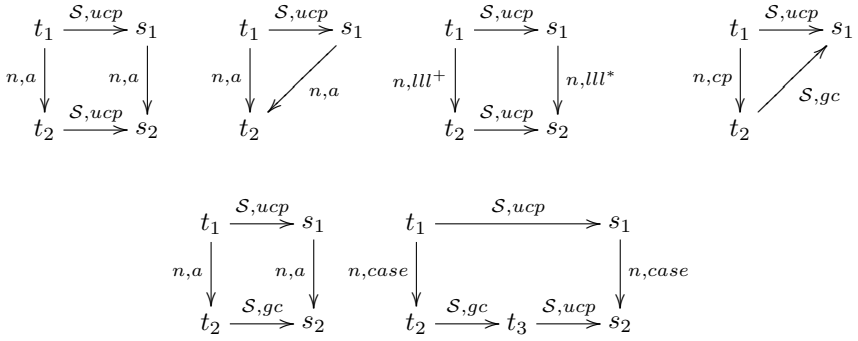
Theorem 10.23. *All the deterministic reductions in the base calculus keep contextual equivalence. I.e. whenever* $t \xrightarrow{a} t'$*, with* $a \in \{cp, lll, case, seq, lbeta\}$*, then* $t \sim_c t'$*.*

Proof. Follows from Propositions 9.3, 9.6, 9.10, 10.21 and 7.3. □

10.8 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 10.24. *The complete sets of forking and commuting diagrams for* $\xrightarrow{\mathcal{S},ucp}$ *can be read off the following graphical diagrams:*



Where $a \in \{(\text{seq}), (\text{choice-l}), (\text{choice-r})\}$ in the fifth diagram

Proof. We show the typical overlappings.

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

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

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

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

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

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

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

□

Lemma 10.25. *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 10.26. *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 9.10, 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 10.24 and 10.4 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 10.25.
- 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 10.24 and 10.4 for the transformation. First we shift the single $\xrightarrow{\mathcal{S},ucp}$-reduction to the right using the diagrams in Lemma 10.24. 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 10.25 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 10.4 and can use the same arguments as in Proposition 10.6 to transform this mixed sequence into a normal order reduction to a WHNF.

□

10.9 Correctness of (abse)

Proposition 10.27. *The reduction (abse) is a correct program transformation.*

Proof. This follows from Proposition 10.26 and Theorem 10.22, since (abse) can be undone by several (ucp)-and (gc)-reductions. □

10.10 Correctness of (cpax)

Proposition 10.28. *The reduction (cpax) is a correct program transformation.*

Proof. This follows from proposition 10.9, since (cpax) can also be performed by several (cpx) reductions. □

Proposition 10.29. *The reduction (cpax) is terminating.*

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

10.11 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 10.30. *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 Theorem 10.22, Theorem 10.23 and Proposition 10.26 show that the proposition holds.

□

10.12 Summary of Properties

Theorem 10.31. *The reductions (ucp), (cpx), (cpax), (gc), (lwas), (cpcx), (abs), (abse) and (xch) keep contextual equivalence. I.e. whenever* $t \xrightarrow{a} t'$, *with* $a \in \{ucp, cpx, cpax, gc, lwas, cpcx, abs, abse, xch\}$, *then* $t \sim_c t'$.

Proposition 10.32. *If* $t\Uparrow$ *and* $t \xrightarrow{a} t'$, *where* a *is any reduction (lll), (seq), (lbeta), (case), (gc) (cpx), (cpax), (ucp), (lwas), then* t' *is also non-terminating.*

Proof. This follows from the contextual equivalences (see Theorem 10.23 and Theorem 10.31). □

Theorem 10.33. *(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 10.23, 10.22 and 7.4. □

11 Simplifications

11.1 A Convergent Rewrite System of Simplifications

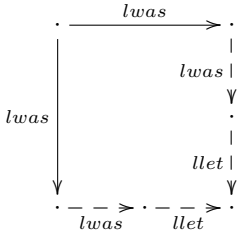
Definition 11.1. *As* simplification rules *we will use (lwas), (llet), (gc), (cpax).*

Note that the rule (lwas) includes (lseq), (lcase), (lapp), but not (llet). The simplification rules (lwas), (llet), (gc), (cpax) keep contextual equivalence, which is proved in Theorem 10.23 and Theorem 10.31. For definitions of confluence and local confluence see e.g. [BN98].

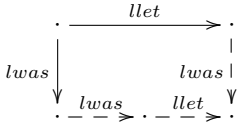
Theorem 11.2. *The set of reductions (lwas), (llet), (gc), (cpax) is confluent (up to α-renaming) and terminating.*

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:

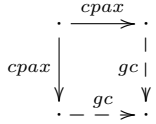
For (lwas) with (lwas):



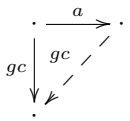
For (lwas) with (llet):



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 10.2 and used in the proof of Proposition 10.3, $\mu_2(t)$ 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]). □

Proposition 11.3. *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.*

11.2 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 `Bot` is the least element w.r.t. $\leq_c$.

Proposition 11.4. *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 7.5, 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 11.5.

1. *If t_1, t_2 are expressions with $t_1{\Uparrow}{\Uparrow}$ and $t_2{\Uparrow}{\Uparrow}$, then* $\texttt{Bot} \sim_c t_1 \sim_c t_2$.
2. *For all expressions s:* $\texttt{Bot} \leq_c s$.
3. $R[\texttt{Bot}] \sim_c \texttt{Bot}$.
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 11.4. Claim 3 and 4 follow using the arguments in the proof of Proposition 11.4. □

11.3 Another Definition of Contextual Equivalence

Proposition 11.6. *$s \leq_c t$ 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 $D' := (\texttt{letrec}\ x_1 = \texttt{Bot}, \ldots, x_n = \texttt{Bot}\ \texttt{in}\ D)$. Then $D'[s]\Downarrow$ follows from $D[s]\Downarrow$, and $D'[t]\Downarrow$ follows from the assumption. The terminating normal order reduction of $D'[t]\Downarrow$ does never put any x_i in a reduction context, since this contradicts Corollary 11.5. Hence the same method as in the proof of Proposition 11.4 shows that we can use the same normal order reduction to show that $D[t]\Downarrow$. □

11.4 Strict Subexpressions

We prove that a strict subexpression s of t, which is additionally 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 a (llet)-reduction that destroys the top level `letrec` of s, or s is eliminated.

Lemma 11.7. *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 $t \xrightarrow{n,*} t_0$, 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 s is in a reduction context.*

Proof. If 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 `Bot` and get the same reduction sequence to a WHNF. This contradicts the assumption that s is a strict subterm of t. □

Lemma 11.8. *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 ($\texttt{case}\ s'\ alts$),
   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 11.5, since ($\texttt{letrec}\ E\ \texttt{in}\ \texttt{Bot}$) $\sim_c$ ($\texttt{Bot}\ s''$) $\sim_c$ ($\texttt{seq}\ \texttt{Bot}\ s''$) $\sim_c$ ($\texttt{case}\ \texttt{Bot}\ alts$) $\sim_c$ `Bot`, since in each of the expressions `Bot` is in a reduction context
The second claim follows from the properties of a precongruence: If $t = C[D[s]]$ we have $C[D[\texttt{Bot}]] \sim_c \texttt{Bot}$. Since $\texttt{Bot} \leq_c D[\texttt{Bot}]$, we obtain $C[\texttt{Bot}] \leq_c C[D[\texttt{Bot}]] \sim_c \texttt{Bot}$, hence $C[\texttt{Bot}] \sim_c \texttt{Bot}$.
The third claim can be proved using Lemma 11.7 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 11.9. *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 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 $t[\texttt{Bot}/s] \sim_c \texttt{Bot}$, and so also $t[\texttt{Bot}/s'] \sim_c \texttt{Bot}$. 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 $t[\texttt{Bot}/s] \xrightarrow{abs} t'[\texttt{Bot}/s']$, hence by Proposition 10.16, we obtain $t'[\texttt{Bot}/s] \sim_c \texttt{Bot}$,
If the reduction does not change s, then also $t[\texttt{Bot}/s] \to t'[\texttt{Bot}/s]$ by a reduction from the base calculus. In this case Theorems 10.23 and7.4 show that $t'[\texttt{Bot}/s] \sim_c \texttt{Bot}$.
The other case is that s is not changed, but eliminated by a (choice), (seq), or (case). In this case we have $t[\texttt{Bot}/s] \to t'[\texttt{Bot}/s] = t'$ and we reach the contradiction $t' \sim_c \texttt{Bot}$ using Theorems 10.23 and7.4.
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.

□

12 Length of Normal Order Reduction

In the following we develop properties of the length of normal order reduction sequences. The lengths are mainly used for normal order reductions of concrete expressions, but we require the claims also for certain induction arguments later for abstract terms.
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.

Definition 12.1. *We define the lengths of normal order reductions.*
Let t be a closed term and Red be a normal order reduction that terminates with a WHNF. Then

1. $\mathrm{rl}\sharp\sharp(Red)$ *is defined to be the number of (choice),(case), (lbeta), (seq)-reductions in Red.*
2. $\mathrm{rl}\sharp(Red)$ *is defined to be the number of (choice), (case), (lbeta), (seq), and (cp)-reductions in 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.*

In the case that the normal order reduction is infinite, we define $\mathrm{rl}\sharp\sharp(Red) = \mathrm{rl}\sharp(Red) = \infty$. *This is consistent with the property of normal order reduction sequences.*

Proposition 12.2. *Let t be a closed term and Red be an infinite normal order reduction sequence for t, then Red contains infinitely many (cp)-reductions as well as infinitely many (lbeta)-reductions.*

Proof. Suppose *Red* has finitely many (cp)-reductions. Then it is sufficient to treat the situation where *Red* has no (cp)-reductions. Then there are only finitely many (lbeta)-reductions, since the number of abstractions is strictly decreased by (lbeta), but not increased by non-(cp)-reductions. The same for (case)-reductions, since a (case)-reduction removes a `case`-expression. Similar for `seq`-reductions. Then we have a normal order reduction that consists only of (lll)-reductions. This, however, contradicts Proposition 10.3.
Now suppose that *Red* has finitely many (lbeta)-reductions. Then the number of surface positions of variables remains bounded, and hence there are finitely many (cp)-reductions. This contradicts the first part. □

Definition 12.3. *Let t be a closed concrete term, and let Red be the unique and maximal normal order reduction of t.*

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. $\mathrm{rl}(t) := \mathrm{rl}\sharp(t) + \mathrm{rl}\flat(t)$.

Note that since the terms are concrete, there are no (choice)-reductions.
In the case that t does not terminate with a WHNF, we define $\mathrm{rl}\sharp\sharp(t) = \mathrm{rl}\sharp(t) = \infty$.

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

Proposition 12.4. *Let t be a closed expression and $Red_1 \in nor(t)$.*

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 12.5. *Let t_1, s_1 be closed expressions and $Red_1 \in nor(t_1)$. Then*

1. *If* $t_1 \xrightarrow{a} s_1$ *with* $a \in \{case, seq, lbeta, cp\}$, *then there exists a* $Red_2 \in nor(s_1)$ *with* $\mathrm{rl}(Red_1) \geq \mathrm{rl}(Red_2)$, $\mathrm{rl}\sharp(Red_1) \geq \mathrm{rl}\sharp(Red_2)$ *and* $\mathrm{rl}\sharp\sharp(Red_1) \geq \mathrm{rl}\sharp\sharp(Red_2)$.

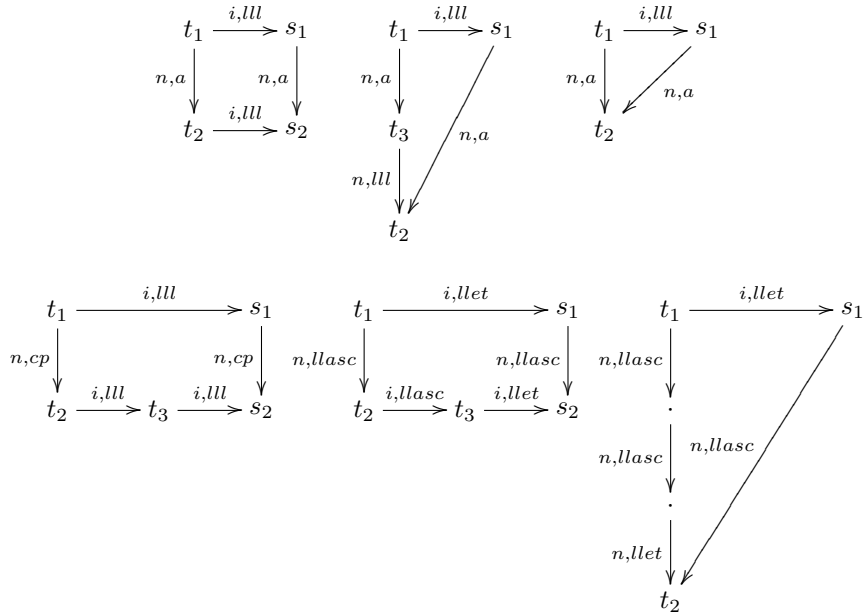
2. *If* $t_1 \xrightarrow{\mathcal{S},a} s_1$ *with* $a \in \{caseS, seqS, lbeta, cpS\}$, *then there exists a* $Red_2 \in nor(s_1)$ *with* $\mathrm{rl}\sharp(Red_1) \geq \mathrm{rl}\sharp(Red_2) \geq \mathrm{rl}\sharp(Red_1) - 1$ *and* $\mathrm{rl}\sharp\sharp(Red_1) \geq \mathrm{rl}\sharp\sharp(Red_2) \geq \mathrm{rl}\sharp\sharp(Red_1) - 1$. *If* $a = cpS$, *then there exists a* $Red_2 \in nor(s_1)$ *with* $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$.
3. *If* $t_1 \xrightarrow{a} s_1$ *with* $a \in \{lll, gc\}$, *then there exists a* $Red_2 \in nor(s_1)$ *with* $\mathrm{rl}(Red_1) \geq \mathrm{rl}(Red_2)$, $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ *and* $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$. *If* $a = gc1$, *then we can select* Red_2 *such that in addition* $\mathrm{rl}(Red_1) = \mathrm{rl}(Red_2)$.
4. *If* $t_1 \xrightarrow{a} s_1$ *with* $a \in \{cpx, cpax, xch\}$, *then there exists a* $Red_2 \in nor(s_1)$ *with* $\mathrm{rl}(Red_1) = \mathrm{rl}(Red_2)$, $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ *and* $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$.
5. *If* $t_1 \xrightarrow{ucp} s_1$, *then*
there exists a $Red_2 \in nor(s_1)$ *with* $\mathrm{rl}(Red_1) \geq \mathrm{rl}(Red_2)$, $\mathrm{rl}\sharp(Red_1) \geq \mathrm{rl}\sharp(Red_2)$ *and* $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$.
6. *If* $t_1 \xrightarrow{lwas} s_1$, *then there exists a* $Red_2 \in nor(s_1)$ *with* $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$.
7. *If* $t_1 \xrightarrow{a} s_1$ *with* $a \in \{cpcx, abs\}$ *then there exists a* $Red_2 \in nor(s_1)$ *with* $\mathrm{rl}(Red_1) = \mathrm{rl}(Red_2)$, $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ *and* $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$.

Proof. The proof is done below in this section in Propositions 12.20, 12.22, 12.8, 12.11, 12.12, 12.14, 12.17, 12.24, and 12.16. □

12.1 Reductions lengths for (lll) and (gc)

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

Lemma 12.6. *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 or an expression lost by a (choice)-reduction.
- 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.

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

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

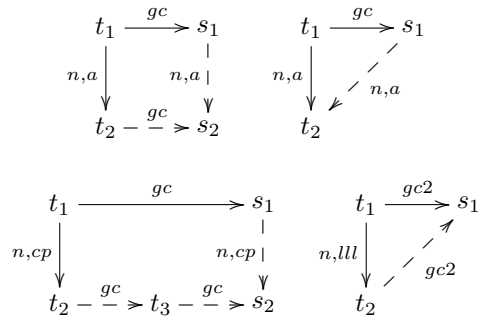
This is covered in the diagram number 5.
A slight variation is the case:

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

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

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



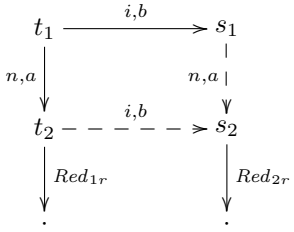
Proof. We omit the arguments for the cases 1,2,3.

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 10.4). □

Now we can prove claim 3 of theorem 12.5.

Proposition 12.8. *Let t_1, s_1 be closed expressions with $Red_1 \in nor(t_1)$ and $t_1 \xrightarrow{a} s_1$ where $a \in \{lll, gc\}$. Then there exists a $Red_2 \in nor(s_1)$ with* $\mathrm{rl}(Red_1) \geq \mathrm{rl}(Red_2)$, $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ *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 9.2 and 10.5.
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 12.6 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 12.7 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 12.9. *Let t_1, s_1 be a closed expressions with $Red_2 \in nor(s_1)$ and $t_1 \xrightarrow{lll} s_1$. Then there exists a $Red_1 \in nor(t_1)$ with $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ and $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$.*

Proof. Using Lemma 12.6 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 $\mu_{lll}(r_1)$.

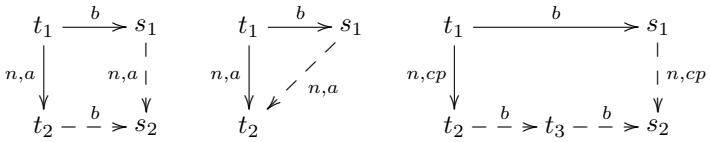
1. Diagram 1 strictly reduces in one triple either μ_1, or leaves μ_1 and strictly decreases μ_2.
2. Diagram 2,3,6 remove one triple from the multiset.
3. Diagram 4 replaces a triple by two strictly smaller triples, where the first component is strictly smaller.
4. Diagram 5 replaces a triple by two 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 (choice), (seq), (case), (lbeta), (cp)-reductions is not modified. Hence the claim holds. □

12.2 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 10.7 have to be reconsidered, since now all positions in a term have to be covered.

Lemma 12.10. *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 12.11. *Let t_1 be a closed expression with $Red_1 \in nor(t_1)$, and $t_1 \xrightarrow{b} s_1$ where $b \in \{cpx, xch\}$. Then there exists a $Red_2 \in nor(s_1)$ with Then $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$, $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ and $\mathrm{rl}(Red_1) = \mathrm{rl}(Red_2)$.*

Proof. This follows by induction on $\mathrm{rl}(Red_1)$ from Lemma 12.10, Lemma 10.8 and 10.12. □

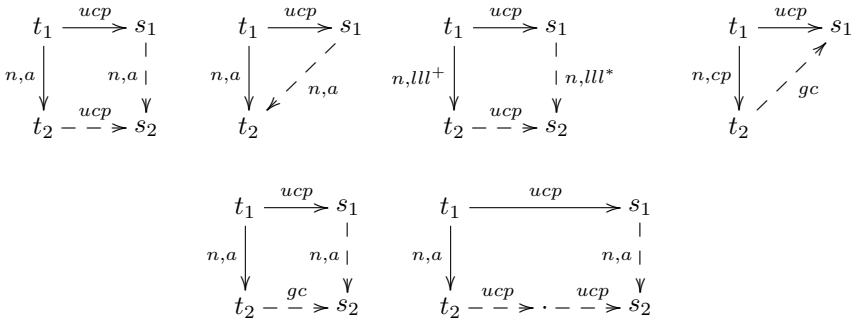
We have to treat the length-modifications by (cpax)-reductions:

Proposition 12.12. *Let t_1 be a closed expression with $Red_1 \in nor(t_1)$, and $t_1 \xrightarrow{cpax} s_1$. Then there exists a $Red_2 \in nor(s_1)$ with $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$, $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ 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 12.11, since the (cpax)-reduction can be simulated by several (cpx) reductions. □

12.3 Reduction Length for ucp-Reductions

Lemma 12.13. *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 10.24, the 6^{th} diagram covers the same case as the 6^{th} case in Lemma 10.24 and in addition the case that the (ucp) takes place in the body of an abstraction. □

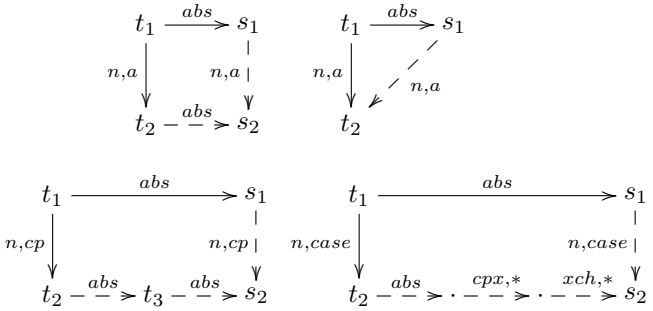
Proposition 12.14. *Let t_1 be a closed expression with $Red_1 \in nor(t_1)$ and $t_1 \xrightarrow{ucp} s_1$. Then there exists a $Red_2 \in nor(s_1)$ with $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$ and $\mathrm{rl}\sharp(Red_1) \geq \mathrm{rl}\sharp(Red_2)$.*

Proof. This follows by induction on $\mathrm{rl}\sharp(Red_1)$ and then on $\mathrm{rl}(Red_1)$ from Lemma 10.25, Lemma 12.13 and Proposition 12.8. □

12.4 Reduction Length for (abs)

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

Lemma 12.15. *The forking diagrams for (abs) in arbitrary contexts are as follows.*



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

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

□

Proposition 12.16. *Let t_1, s_1 be closed expressions with $Red_1 \in nor(t_1)$ and $t_1 \xrightarrow{abs} s_1$. Then there exists a $Red_2 \in nor(s_1)$ with $\mathrm{rl}(Red_1) = \mathrm{rl}(Red_2)$, $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ 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 12.15 are used, and part 4 in Theorem 12.5, and since (abs) transforms WHNFs into WHNFs and vice versa. □

12.5 Reduction Length for (lwas)-Reductions

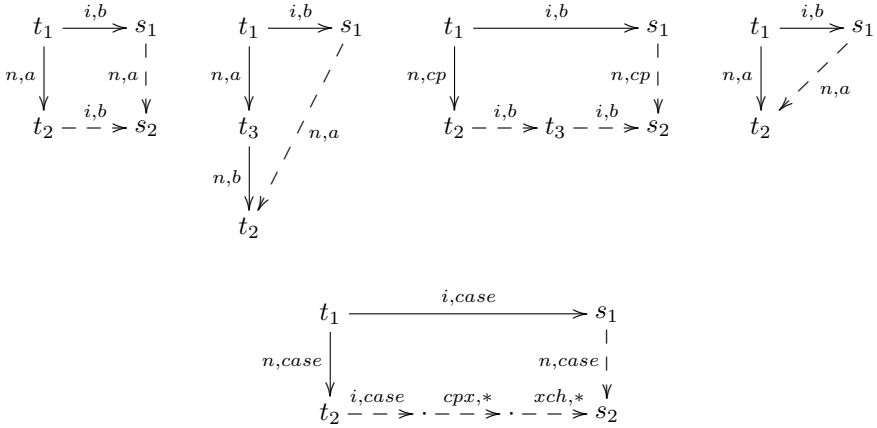
Proposition 12.17. *Let t_1 be a closed expression with $Red_1 \in nor(t_1)$ and $t_1 \xrightarrow{lwas} s_1$. Then there exists a $Red_2 \in nor(s_1)$ with $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$.*

Proof. Since (lwas) can be simulated using (ucp) and (llet)-reductions in both directions (see proof of Lemma 10.30), Propositions 12.14, and 12.8 show the claim. □
It would also be possible to sharpen this proposition, however, this is not necessary for the further development.

12.6 Using Diagrams for Internal Base Reductions

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

Lemma 12.18. *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 (choice) or (seq).
The exceptional diagram is a (case)-(case)-overlapping:

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

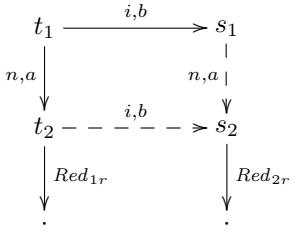
□

Lemma 12.19. *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 12.5

Proposition 12.20. *Let t_1, s_1 be closed expressions with $Red_1 \in nor(t_1)$ and $t_1 \xrightarrow{a} s_1$ where $a \in \{case, seq, lbeta, cp\}$. Then there exists a $Red_2 \in nor(s_1)$ with* $\mathrm{rl}(Red_1) \geq \mathrm{rl}(Red_2)$, $\mathrm{rl}\sharp(Red_1) \geq \mathrm{rl}\sharp(Red_2)$ *and* $\mathrm{rl}\sharp\sharp(Red_1) \geq \mathrm{rl}\sharp\sharp(Red_2)$.

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 12.19 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 12.18 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:



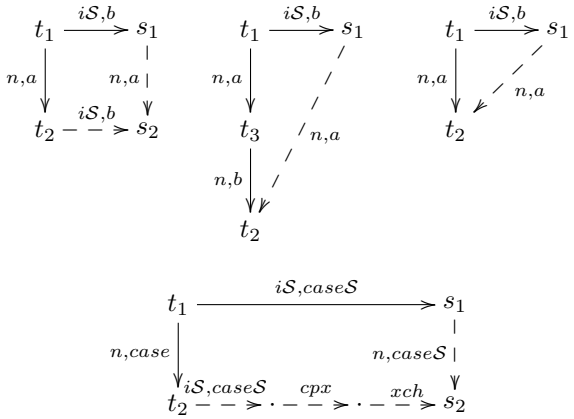
In case 1, we obtain by induction that there exists a reduction Red_{2r} of t_2 with $\mathrm{rl}\sharp(Red_{1r}) \geq \mathrm{rl}\sharp(Red_{2r})$, $\mathrm{rl}(Red_{1r}) \geq \mathrm{rl}(Red_{2r})$, and $\mathrm{rl}\sharp\sharp(Red_{1r}) \geq \mathrm{rl}\sharp\sharp(Red_{2r})$.
In case 3, we have to 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 12.11.

□

12.7 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 12.21. *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 12.18 can be used. See also Lemma 9.7. 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 12.5

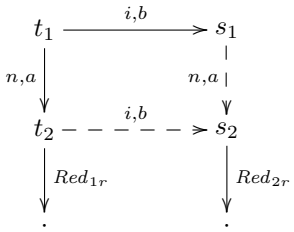
Proposition 12.22. *Let t_1, s_1 be a closed expression with $Red_1 \in nor(t_1)$ and $t_1 \xrightarrow{\mathcal{S},a} s_1$ where $a \in \{caseS, seqS, lbeta, cpS\}$. Then there exists a $Red_2 \in nor(s_1)$ with $\mathrm{rl}\sharp(Red_1) \geq \mathrm{rl}\sharp(Red_2) \geq \mathrm{rl}\sharp(Red_1) - 1$ 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 12.20 already shows that there exists a $Red_2 \in nor(s_1)$ 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 12.19 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 12.21 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 12.11.

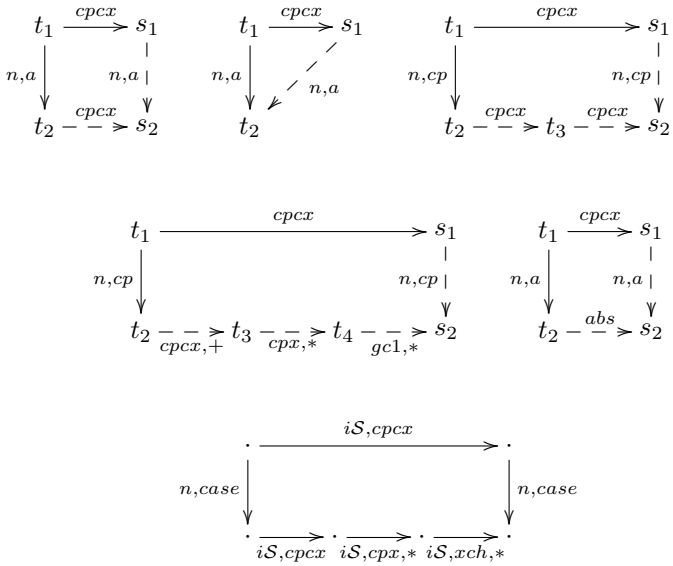
In the case that $a =$ cpS, the equation $\mathrm{rl}\sharp\sharp(Red_1) = \mathrm{rl}\sharp\sharp(Red_2)$ follows by induction using the diagrams 1,2,3.

□

12.8 Reduction Length for (cpcx)

The reduction (cpcx) is defined as follows (see also Definition 10.1).

Lemma 12.23. *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 10.17. We give a further prototypical example for diagram 6:

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

The following case is covered by diagram 5:

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

□

Proposition 12.24. *Let s_1, t_1 be closed expressions with $Red_1 \in nor(t_1)$ and $t_1 \xrightarrow{cpcx} s_1$. Then there exists a $Red_2 \in nor(s_1)$ with $\mathrm{rl}(Red_1) = \mathrm{rl}(Red_2)$, $\mathrm{rl}\sharp(Red_1) = \mathrm{rl}\sharp(Red_2)$ 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 10.18 is used for the base case, and the diagrams in the following Lemmas are used: 12.23, 12.7, 12.15, and 12.10. □

12.9 Length of Normal Order Reductions for Concrete Expressions

We specialize the claims on the lengths of reduction sequences to concrete expressions.

Theorem 12.25. *Let t_1, s_1 be closed and terminating concrete expressions. Then*

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 $\mathrm{rl}\sharp\sharp(t_1) = \mathrm{rl}\sharp\sharp(s_1)$.*
4. *If $t_1 \xrightarrow{a} s_1$ with $a \in \{cpx, cpax, xch\}$, 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 $\mathrm{rl}\sharp\sharp(t_1) = \mathrm{rl}\sharp\sharp(s_1)$.*
7. *If $t_1 \xrightarrow{a} s_1$ with $a \in \{cpcx, abs\}$ then $\mathrm{rl}(t_1) = \mathrm{rl}(s_1)$, $\mathrm{rl}\sharp(t_1) = \mathrm{rl}\sharp(s_1)$ and $\mathrm{rl}\sharp\sharp(t_1) = \mathrm{rl}\sharp\sharp(s_1)$.*

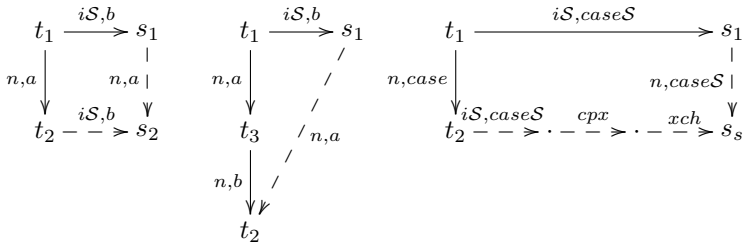
Proof. Follows from Theorem 12.5 by specializing it to concrete expressions, where normal order reductions are uniquely determined. □

12.10 Length of Normal Order Reduction in Concrete Terms Using Strictness Optimization

In this subsection we show 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.

Proposition 12.26. *Let t_1 be a closed concrete (*LR*-) expression 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 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 $a \neq (cpt)$, *then* $\mathrm{rl}\sharp\sharp(t_1) = 1 + \mathrm{rl}\sharp\sharp(s_1)$ *and if* $a = (cpt)$, *then* $\mathrm{rl}\sharp\sharp(t_1) = \mathrm{rl}\sharp\sharp(s_1)$.

Proof. We only consider the case that $t_1 \Downarrow$.
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 $t_1[\mathtt{Bot}/t_0] \sim_c \mathtt{Bot}$ cannot hold simultaneously.
Let $t_1 \xrightarrow{n} t_2$.
Lemma 11.9 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 12.21).



The short triangle-diagram from Lemma 12.21 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 12.25) for (xch) and (cpx)-reductions. We obtain that the claim of the proposition holds.

□

12.11 Local Evaluation and Deep Subterms

In this subsection we restrict considerations to concrete terms $\in$ LR and show that the reduction lengths for deep and strict subterms is strictly smaller than the reduction length of the top term.

Definition 12.27. *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 $(\texttt{letrec}\ Env\ \texttt{in}\ x)$, where only the reductions are included that make modifications in Env, i.e., a perhaps 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 $(\texttt{letrec}\ Env\ \texttt{in}\ x)$ terminates with a WHNF, then the length of a local evaluation is denoted as $\mathrm{rl}\sharp_{loc}(\texttt{letrec}\ Env\ \texttt{in}\ x)$, otherwise $\mathrm{rl}\sharp_{loc}(\texttt{letrec}\ Env\ \texttt{in}\ x) := \infty$.

Definition 12.28. *In the closed concrete term $(\texttt{letrec}\ x = t, y = s, Env\ \texttt{in}\ r)$, we say x* requires *y, iff the local evaluation of x in $(\texttt{letrec}\ x = t, y = \texttt{Bot}, Env\ \texttt{in}\ r)$ does not produce a WHNF for x, i.e., results in $\texttt{Bot}$ for x.*

Lemma 12.29. *Let $t = (\texttt{letrec}\ x = s_x, y = s_y, Env\ \texttt{in}\ r)$ be a closed concrete term, where x requires y, and let $t \xrightarrow{n} t'$. Then in t' the variable x also requires y.*

Proof. First assume that the reduction is not a (llet)-reduction
If $t \xrightarrow{n} t'$ modifies only r, then the Lemma holds, since there is no difference in the local evaluations of x wrt. t and t'. If the reduction modifies a `case`-expression in r, where the constructor application is in the top environment, let $t' = (\texttt{letrec}\ x = s'_x, y = s'_y, Env'\ \texttt{in}\ r')$. Then one of the two relations $(\texttt{letrec}\ x = s_x, y = \texttt{Bot}, Env\ \texttt{in}\ x) = (\texttt{letrec}\ x = s'_x, y = \texttt{Bot}, Env'\ \texttt{in}\ x)$ or $(\texttt{letrec}\ x = s_x, y = \texttt{Bot}, Env\ \texttt{in}\ x) \xrightarrow{abs} (\texttt{letrec}\ x = s'_x, y = \texttt{Bot}, Env'\ \texttt{in}\ x)$ holds, which implies that the Lemma holds.
If $t \xrightarrow{n} t'$ modifies the top environment, then similar arguments show that the Lemma holds.
Now assume that the reduction is a (llet)-reduction. If the (llet)-reduction does not change the top level structure of t, then again the same arguments suffice for a proof.
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 11.8. □

Lemma 12.30. *Let the closed concrete term $t = (\texttt{letrec}\ x_1 = t_1, \ldots, x_n = t_n, Env\ \texttt{in}\ r)$ be such that there is 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. Assume w.l.o.g. that some local evaluation of x_1 terminates with success. Moreover assume, that this is the rl$\sharp(\cdot)$-shortest successful local evaluation for all x_i.
Every x_i is bound to a term that is not a value, since otherwise we have a shorter evaluation. W.l.o.g. we can ignore the (lll)-reductions. Let the first normal order reduction step of $t \xrightarrow{n} t'$ be a non-(lll)-normal order reduction step. The cyclic dependency remains as before the reduction (see Lemma 12.29). The term t' is a counterexample with a shorter 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 $i = 1, \ldots, n$.
□

Proposition 12.31. *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 rl$\sharp(t_1) \geq$ rl$\sharp_{loc}(\texttt{letrec}\ Env\ \texttt{in}\ x)$ *and* rl$\sharp\sharp(t_1) \geq$ 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 rl$\sharp_{loc}(\texttt{letrec}\ Env\ \texttt{in}\ x)$. If t_x is in WHNF, then rl$\sharp_{loc}(\texttt{letrec}\ Env\ \texttt{in}\ x) = 0$, and the claim holds. Now let t_x be a non-WHNF. Let $t_1 \xrightarrow{a} t_2$ 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 12.25. 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 12.26 and induction on the number of local evaluations shows the claim. □

Definition 12.32. *Let* $t = (\texttt{letrec}\ Env\ \texttt{in}\ t_1)$. *Let* t_1 *be either an application, a* `seq`*-expression, a* `case`*-expression, or a variable, say* x_1. *In the latter case there must be a part of the environment 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. Let* $x \in LV(Env)$ *such that* $x \notin \{x_1, \ldots, x_n\}$.
Then we say that t_x *in the binding* $x = t_x$ *is a* deep subterm *in* t.

The next proposition shows that for deep and strict proper subexpressions, the number rl$\sharp\sharp(\cdot)$ of local normal order reductions to a WHNF is less than the corresponding number for the top term.

Proposition 12.33. *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 rl$\sharp\sharp(t_1) >$ 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 11.9.
We have to show that t_x is also a deep subterm in t_2. The subterm t_1' is not modified. If it 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 12.30. 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 rl$\sharp\sharp$ does mot change. If the next reduction in the local evaluation is a (cpS)-reduction, then apply apply Proposition 12.26. 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 12.26. We obtain that $\mathrm{rl}\sharp\sharp(t_1) = 1 + \mathrm{rl}\sharp\sharp(t_2)$.
Hence the induction shows that $\mathrm{rl}\sharp\sharp(t_1) > \mathrm{rl}\sharp\sharp((\texttt{letrec}\ Env\ \texttt{in}\ x))$. □

13 Contextual Least Upper Bounds

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

Definition 13.1. *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 13.2. *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 13.3. *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 13.4. *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 $s = club((s_i)_i)$.

Proof. We prove that the conditions of Lemma 13.3 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 7.1. 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 7.1). 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. □

14 Behavioral Preorder and Equivalence

In addition to the context lemma we require the notion and tool of behavioral equivalence $\leq_b$ in LRA for proving $\leq_c$-relations
We generalize the behavioral preorder (see [Abr90]), which avoids the counterexamples to a naïve generalization given in [SS03], and which is compatible with the preorder in [Man04].

14.1 Variant of (case)-Rules

Definition 14.1. *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.*
The extra reduction rule (ibot) is required by the definition of the behavioral preorder.

(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)	$(\texttt{letrec}\ x = (c\ x_1 \ldots x_n), Env\ \texttt{in}\ C[\texttt{case}\ x\ ((c\ y_1 \ldots y_n) \to s)\ alts])$
	$\to \texttt{letrec}\ x = (c\ x_1 \ldots x_n), Env$
	$\quad \texttt{in}\ C[(\texttt{letrec}\ y_1 = x_1, \ldots, y_n = x_n\ \texttt{in}\ s)]$
(case-cx)	$\texttt{letrec}\ x = (c\ x_1 \ldots x_n), Env,$
	$\quad y = C[\texttt{case}\ x\ ((c\ y_1 \ldots y_n) \to s)\ alts]\ \texttt{in}\ r$
	$\to\ \texttt{letrec} x = (c\ x_1 \ldots x_n), Env,$
	$\quad y = C[(\texttt{letrec}\ y_1 = x_1, \ldots, y_n = x_n\ \texttt{in}\ s)]\ \texttt{in}\ r$
(case-cx)	*like (case) in all other cases*
(ibot)	$t \to \bot$

Lemma 14.2. *The following holds:*

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

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. □

The number of normal order reduction steps is not changed due to Theorem 12.5, and since (case-cx) can be simulated by a (case) and subsequent (cpx) and (gc)-reductions.

14.2 Behavioral Preorder

Definition 14.3. *Let t be a closed expression. If t can be reduced to t' using the reduction rules of the base calculus, and additionally (ibot), (gc), (cpx), (cpcxnoa), (abse) and all reductions are in surface contexts, then we write $t \xrightarrow{bhv,*} t'$.*
A B-value *is either an abstraction, or a constructor application $(c\ x_1 \dots x_n)$, where x_i are variables.*
A term t is in bhv weak head normal form (BWHNF)*, iff it is a B-value or of the form* (`letrec` *Env* `in` *v*)*, where v is a B-value, and $Env = \{x_1 = t_1, \dots, x_n = t_n\}$, where for all $i : t_i$ is either a B-value, or $\bot$.*
We partition BWHNFs into FBWHNFs and CBWHNFs: A BWHNF (`letrec` *Env* `in` *v*)*, where v is an abstraction, is called a FBWHNF, and if v is a constructor application it is called a CBWHNF.*

Definition 14.4. *We define the behavioral preorder $\leq_b$ on closed terms.*
For a relation η define the relation $[\eta]$ for closed terms s, t:
$s\ [\eta]\ t$ iff: If $s \xrightarrow{bhv,*} s_1$, and s_1 is a FBWHNF, then there is a FBWHNF t_1 with $t \xrightarrow{bhv,*} t_1$, and
for all closed r : (`letrec` $x = r$ `in` $(s_1\ x)$) η (`letrec` $x = r$ `in` $(t_1\ x)$) for a fresh variable x,
and
if $s \xrightarrow{bhv,*} s_1 =$ (`letrec` Env_s `in` $(c\ x_1 \dots x_n)$), i.e., s_1 is a CBWHNF, then there is a CBWHNF t_1 with $t \xrightarrow{bhv,*} t_1 =$ (`letrec` Env_t `in` $(c\ y_1 \dots y_n)$), and for all i : (`letrec` Env_s `in` x_i) η (`letrec` Env_t `in` y_i).
If c is a 0-ary constant, then the recursive condition is omitted.

The behavioral preorder $\leq_b$ on closed terms is defined as the greatest fixpoint on relations on closed abstract terms of the operator $[\cdot]$.

We extend this order to open terms s, t as follows:
$s \leq_b^o t$ iff for $FV(s,t) = \{x_1, \dots, x_n\}$ and for all closed $t_1, \dots, t_n$:

$$(\texttt{letrec}\ x_1 = t_1, \dots x_n = t_n\ \texttt{in}\ s) \leq_b (\texttt{letrec}\ x_1 = t_1, \dots x_n = t_n\ \texttt{in}\ t).$$

Conjecture 14.5. We conjecture that the behavioral preorder $\leq_b$ is contained in the contextual preorder $\leq_c$ on closed terms.

A proof of this conjecture for a weaker language is in [Man04]. We are working on generalizing and extending this proof also for LRA.
We will have to use this conjecture in the proof of correctness of the strictness analysis. Wherever the conjecture is used, we will make the reference explicit.

Proposition 14.6. *The following equivalence holds: for all $s, t : s \leq_b^o t$ implies $s \leq_c t$.*

Proof. We use the Conjecture 14.5.
Assume that $s \leq_b^o t$ holds. Let $\{x_1, \ldots, x_n\} = FV(s, t)$. Then by definition, for all closed $t_1, \ldots, t_n$: ($\texttt{letrec}\ x_1 = t_1, \ldots x_n = t_n\ \texttt{in}\ s$) $\leq_b$ ($\texttt{letrec}\ x_1 = t_1, \ldots x_n = t_n\ \texttt{in}\ t$). By the definition of behavioral preorder, this implies that $\lambda x_1, \ldots, x_n.s \leq_b \lambda x_1, \ldots, x_n.t$. Using Conjecture 14.5, renaming variables and plugging the expressions into the context $((\lambda y_1, \ldots, y_n.[\cdot])\ x_1 \ldots x_n)$ implies $(\lambda y_1, \ldots, y_n.s[y_i/x_i, i = 1, \ldots, n])\ x_1 \ldots x_n \leq_c (\lambda y_1, \ldots, y_n.t[y_i/x_i, i = 1, \ldots, n])\ x_1 \ldots x_n$. By Theorem 10.23 reduction implies contextual equivalence, so (lbeta)-reducing both sides we obtain $s \leq_c t$. □

Lemma 14.7. *If $t \xrightarrow{bhv,*} t'$, then $t' \leq_c t$.*

Proof. By Theorems 10.23 and 7.4, $\xrightarrow{bhv}$-reductions (except (ibot)) imply contextual equivalence. The reduction (ibot) in turn makes the expression smaller w.r.t. $\leq_c$, which is a consequence of Proposition 11.4. □

14.3 Iterative Deepening Evaluations

Due to optimzations in the proofs, this subsection is no longer directly required in later proofs.

Lemma 14.8. *Let $t = (\texttt{letrec}\ Env, x = r\ \texttt{in}\ t')$ be a concrete expression, such that the local evaluation of x (see Def. 12.27) does not terminate. Then $t \sim_c (\texttt{letrec}\ Env, x = \bot\ \texttt{in}\ t')$*

Proof. ($\texttt{letrec}\ Env, x = \bot\ \texttt{in}\ t') \leq_c t$ follows from Corollary 11.5.
We use the context lemma for the other direction: Let R be a reduction context and let $R[t]\Downarrow$. If the normal order reduction starts a local evaluation of x, then t is strict in r, and the Proposition 12.31 shows that the normal order reduction does not terminate. Hence the normal order reduction never starts a local evaluation of r, and hence it can be replaced by `Bot` without changing the normal order reduction. Hence $R[(\texttt{letrec}\ Env, x = \bot\ \texttt{in}\ t')]\Downarrow$. Now the context lemma shows that $(\texttt{letrec}\ Env, x = \bot\ \texttt{in}\ t') \leq_c t$, and the claim holds. □

Definition 14.9. *Let s be a closed concrete expression. We define two series s_i and $s_i^\perp$ of expressions as follows: If s has no WHNF, then $s_0 = \perp$. Otherwise, s_0 is constructed from s by applying normal order reduction to WHNF. Then $s_0 = (\texttt{letrec}\ Env_0\ \texttt{in}\ s_0')$. Using the reductions (abse), (llet) and (cpcxnoa), we can assume that s_0' is a B-value after the reduction.*
The expressions $s_i = (\texttt{letrec}\ Env_{i,1}, Env_{i,2}\ \texttt{in}\ s_0')$ for $i \geq 1$ have a fixed partitioning of their top level environment into two parts $Env_{i,1}, Env_{i,2}$, where $Env_{i,1}$ represents the already evaluated bindings in the i^{th} step and $Env_{i,2}$ the not yet evaluated bindings. If there is no top level environment, then these parts are empty. The bound terms in $Env_{i,1}$ are B-values or $\perp$, if the corresponding local evaluation does not terminate.
The construction of s_{i+1} from s_i is as follows:
If $Env_{i,2}$ is empty, then stop the sequence.
Otherwise, let $Env_{i,2} = \{x_{i,2,1} = s_{i,2,1}, \ldots\}$.
For all j: if the local evaluation of $x_{i,2,j}$ does not terminate, then replace $s_{i,2,j}$ by $\perp$. Otherwise make a local evaluation of the variables $x_{i,2,j}$ in $Env_{i,2}$, where (case) is replaced by (case-cx). If there is a bound term that is a constructor application that is not a B-value, then proceed as follows: for constructor applications that have arguments that are not variables, apply (abse), and then (llet); for variables apply (cpx). We collect the new bindings at top level in an environment $Env_{i+1,2}$. The modified environment $Env_{i,2}$ after the evaluations is denoted as $Env'_{i,2}$. The result is the term s_{i+1} where $Env_{i,1}$ is unchanged, since we have used (case-cx) instead of (case), and $Env_{i+1,1} := Env_{i,1} \cup Env'_{i,2}$ and $Env_{i+1,2} = Env_{i,3}$.
The expression $s_i^\perp$ is constructed from s_i by replacing all bound terms in the environment $Env_{i,2}$ by $\perp$ which gives the environment $Env_{i,2}^\perp$

Lemma 14.10. *For the terms s, s_i as defined above in Definition 14.9, we have $s \sim_c s_i$ for all i.*

Proof. This follows from the correctness of reductions as program transformations, i.e. from Theorems 10.23,10.22, Lemma 14.8 and 14.2. □

Lemma 14.11. *The expressions $s_i, s_i^\perp$ for $i \geq 1$ can also be represented as $s_i = (\texttt{letrec}\ Env_1^{\text{seq}}, \ldots, Env_i^{\text{seq}}, Env_{i,2}\ \texttt{in}\ s_0')$, and $s_i^\perp = (\texttt{letrec}\ Env_1^{\text{seq}}, \ldots, Env_i^{\text{seq}}, Env_{i,2}^\perp\ \texttt{in}\ s_0')$. We have $\bigcup_{j=1}^{i} Env_i^{\text{seq}} = Env_{i,1}$. The bound terms in Env_j^{seq} are B-values or $\perp$ for all j. The variables $LV(Env_i^{\text{seq}})$ may be contained in $FV(Env_{i-1}^{\text{seq}})$ for $i \geq 1$, but not in $FV(Env_k^{\text{seq}})$ for $0 \leq k < i-1$.*

The intention in the following is to show that the sequence $(s_i^\perp)_i$ is an ascending chain with $s = club((s_i^\perp)_i)$.

Lemma 14.12. *Let $s, s_i, s_i^\perp$ be as in Definition 14.9. Then $s_i^\perp$ are BWHNFs with $s \xrightarrow{bhv,*} s_i^\perp$ for all i*

Proof. This holds, since the expressions $s_i^\perp$ are BWHNFs, and the reductions are permitted by $\xrightarrow{bhv}$.

Lemma 14.13. *Let $s, s_i, s_i^\perp$ be as in Definition 14.9. Then the sequence $s_i^\perp$ is a $\leq_c$-ascending chain with $s_i^\perp \leq_c s$ for all i.*

Proof. It is clear that $s_i^\perp \leq_c s$, by Lemma 14.7 and 14.10
We omit the trivial cases.
Let $s_i = (\texttt{letrec}\ Env_{i,1}, Env_{i,2}\ \texttt{in}\ s_0')$ with $Env_{i,2} = \{x_1 = t_1, \ldots, x_n = t_n\}$, where s_0' and the bound terms in $Env_{i,1}$ are B-values. The expression s_{i+1} as constructed in definition 14.9 has the form $s_{i+1} = (\texttt{letrec}\ Env_{i,1}, Env_{i,2}', Env_{i+1,2}\ \texttt{in}\ s_0')$. We have $s_{i+1}^\perp = (\texttt{letrec}\ Env_{i,1}, Env_{i,2}', Env1_{i+1,2}^\perp\ \texttt{in}\ s_0')$, where the bound terms in $Env_{i+1,2}'$ are all equal to $\perp$. Since $LV(Env_{i,2}) = LV(Env_{i,2}')$, and the variables in $LV(Env_{i+1,2})$ do not occur in $Env_{i,1}$ nor in s_0', the term $s_i^\perp$ can be reached from $s_{i+1}^\perp$ by the following transformations: replace in $s_{i+1}^\perp$ the terms that are bound to the variables in $LV(Env_{i,2})$ by $\perp$, and then use perhaps several (gc)-reductions. This implies $s_i^\perp \leq_c s_{i+1}^\perp$ by Theorem 10.31 and Proposition 11.4. □

Lemma 14.14. *Let s be a concrete closed term and let s_i be defined as in Definition 14.9, and let R be a reduction context. Given a normal order reduction $R[s] \xrightarrow{n,*} t$, where t is a WHNF, then there exists the following reduction sequence:*
$R[s] \xrightarrow{\mathcal{AS},*} R[s_i] \xrightarrow{n,*} t'$, *where $t \xrightarrow{*} t'$ and t' is a WHNF, and the $\xrightarrow{\mathcal{AS},*}$- reduction may use base reductions and the extra reductions (cpx), (gc), (cpcxnoa). I.e.,*

$$\begin{array}{ccc} R[s] & \xrightarrow{\mathcal{AS},*} & R[s_i] \\ {\scriptstyle n,*}\downarrow & & \downarrow{\scriptstyle n,*} \\ t & \xrightarrow[*]{} & t' \end{array}$$

Moreover, the number of reductions of $R[s] \xrightarrow{n,*} t$ is greater than or equal to the number of reductions of $R[s_i] \xrightarrow{n,*} t'$.

Proof. This follows easily from Theorem 12.5. □

Lemma 14.15. *Let s be a concrete closed term and let $s_i, s_i^\perp$ be defined as in Definition 14.9, and let R be a reduction context. If $R[s_i]\Downarrow$, and the reduction requires less than i normal order reductions of type (case), (lbeta), or (seq), then $R[s_i^\perp]\Downarrow$.*

Proof. Select a normal order reduction sequence as a witness of $R[s_i]\Downarrow$. This reduction may put bound expressions of the top level of s_i into a reduction context. Using Lemma 14.11, it is easy to see using induction, that bound terms in Env_j^{seq} are required not before at least j (case) or (lbeta)-normal order reductions were performed. Since the difference between s_i and $s_i^\perp$ is in the bound

terms of $Env_{i,2}$, at least i normal order reductions are required before the B-values of these bound terms are demanded. Hence, If $R[s_i]\Downarrow$ using at most i normal order reductions of type (case), (lbeta), or (seq), then also $R[s_i^\perp]\Downarrow$ using at most i normal order reductions of type (case), (lbeta), or (seq). □

Proposition 14.16. *Let s be a concrete closed expression. Let the terms $s_i, s_i^\perp$ be defined as in Definition 14.9. Then $s = club((s_i^\perp)_i)$.*

Proof. We use the criterion in Lemma 13.4. The chain is $\leq_c$-ascending by Lemma 14.13. To show the second condition, let R be any reduction context, let $R[s]\Downarrow$, select a terminating normal order reduction for $R[s]$, and let i be the number of normal order reductions of type (case), (lbeta), or (seq). Since $s \xrightarrow{\mathcal{AS},*} s_i$ (where extra reductions are allowed), and thus $R[s] \xrightarrow{\mathcal{AS},*} R[s_i]$, Lemma 14.14 shows that $R[s_i]\Downarrow$ with a reduction using at most i normal order reductions of type (case), (lbeta), or (seq). Lemma 14.15 now shows that $R[s_i^\perp]\Downarrow$.
Finally, Lemma 13.4 implies that $s = club((s_i^\perp)_i)$. □

14.4 Union Theorem

The main result in this subsection is the union-theorem, which states that the (choice)-rule constructs a union, i.e. if $s \leq_c t$, where s is a closed concrete expression, and t_l, t_r are the two reducts of a surface choice-redex, then $s \leq_c t_l$ or $s \leq_c t_r$.

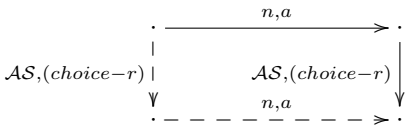
Theorem 14.17. *(Union-Theorem)*
Let s be a concrete closed term and t be an abstract term such that $s \leq_c t$. If for some $\oplus$-$\mathcal{AS}$-redex in t, we have the two different reductions $t \xrightarrow{\mathcal{AS},choice-r} t_r$ and $t \xrightarrow{\mathcal{AS},choice-l} t_l$, then $s \leq_c t_r$ or $s \leq_c t_l$.

Proof. Assume that the theorem does not hold. Then there is a closed term $t = C[r_1 \oplus r_2]$, where C is a AS-context, $t \xrightarrow{\mathcal{AS},choice-l} t_l = C[r_1]$, $t \xrightarrow{\mathcal{AS},choice-r} t_r = C[r_2]$, and $s \not\leq_c t_l$ as well as $s \not\leq_c t_r$. By the context-lemma 7.1, there are reduction contexts R_l, R_r with $R_l[s]\Downarrow, R_r[s]\Downarrow$ and $R_l[t_l]\Uparrow\Uparrow$, $R_r[t_r]\Uparrow\Uparrow$. Let $S' := (\texttt{letrec}\ x = [\cdot]\ \texttt{in seq}\ R_l[x]\ R_r[x])$. Then S' is an AS-context. We have to argue that $S'[s]\Downarrow$: We have to show that $(\texttt{letrec}\ x = s\ \texttt{in seq}\ R_l[x]\ R_r[x])\Downarrow$. The term $(\texttt{letrec}\ x = s\ \texttt{in}\ R_l[x])$ terminates. A terminating normal order reduction can be transferred and yields a term of the form $(\texttt{letrec}\ Env_s, Env_R\ \texttt{in seq}\ r_0\ R_r[x])$, such that `seq` is applicable and reduces to $(\texttt{letrec}\ Env_s, Env_R\ \texttt{in}\ R_r[x])$. Applying (gc) and all the reductions on the Env_s backwards produces the term $(\texttt{letrec}\ x = s\ \texttt{in}\ R_r[x])$. Since s is deterministic, we can apply the Theorems 10.23 and 10.31 and obtain that $S'[s]\Downarrow$.
Now we have to argue that $S'[t]$ does not terminate. So assume that there is a terminating normal order reduction of $S'[t]$. There are two possibilities:

1. The normal order reduction does not reduce the subterm $r_1 \oplus r_2$, i.e. this subterm is never in a reduction context in the normal order reduction. Then

the same terminating normal order reduction is possible for $S'[t_r], S'[t_l]$, which contradicts our assumption.

2. The normal order reduction includes a reduction step (choice) with redex $r_1 \oplus r_2$. W.l.o.g. we assume it is a (choice-r)-reduction. It is easy to see that a $\xrightarrow{\mathcal{AS},choice-r}$-reduction commutes with normal order reductions, if the (choice)-redex is already in an AS-context in the upper left corner of the diagram, i.e. before the normal order reduction



The subterm $r_1 \oplus r_2$ is in an $\mathcal{AS}$-context at the start of the reduction, and since normal order reduction steps leave terms in $\mathcal{AS}$-contexts, this holds during all reduction steps until the subterm is the normal order redex. The commuting diagram then shows that the (choice-r)-reduction can be shifted to the left in the reduction. Finally we obtain $S'[C[r_1 \oplus r_2]] \xrightarrow{\mathcal{AS},choice-r} S'[t_r]$ and there is a terminating normal order reduction for $S'[t_r]$.
This is a contradiction to the assumption. Hence the theorem holds.

□

Note that Theorem 14.17 does in general not hold if s is not a concrete expression.

15 Deterministic 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.
For a concrete subterm t of ($\texttt{letrec}\ x = t, Env\ \texttt{in}\ S[x]$) where $FV(t) = \emptyset$ and S is a surface context, an instance of the problem is to show that ($\texttt{letrec}\ x = t, Env\ \texttt{in}\ S[x]$) and ($\texttt{letrec}\ x = t, Env\ \texttt{in}\ S[t]$) are contextually equivalent.

Definition 15.1. *A closed concrete term is also called* deterministic*.*
Let s be a subterm in a surface context of the abstract term t.
Then s is called non-deterministic *, if it contains a $\oplus$-expression, or if there is a let-bound free variable in s that is bound to a non-deterministic subterm.*
A subterm s of t, which is in a surface context is deterministic*, if it is not a non-deterministic subterm.*

A direct consequence of the definition of a deterministic subterm s is that the local evaluations of s will not use (choice)-reductions.

We treat the following situation:
Let $t = (\texttt{letrec}\ x_1 = y_1, x_2 = y_1, Env_1, Env_{\text{rest}}\ \texttt{in}\ s)$, where y_1, and all subterms in Env_1 are deterministic, and $FV(Env_1) \cup \{y_1\} \subseteq LV(Env_1)$. We want to show

that $t \sim_c t'$, where $t' = (\texttt{letrec}\ x_1 = y_1, x_2 = z_1, Env_1, Env_2, Env_{\text{rest}}\ \texttt{in}\ s)$, where Env_2 is a renamed copy of Env_1, including the let-bound variables. More precisely, let $Env_1 = \{y_1 = r_1, \ldots, y_n = r_n\}$, and let $\rho = \{y_1 \mapsto z_1, \ldots, y_n \mapsto z_n\}$, where z_i are new variables. Then $Env_2 = \{z_1 = \rho(r_1), \ldots, z_n = \rho(r_n)\}$.

Proposition 15.2. *Under the above conditions, we have $t \sim_c t'$.*

Proof. We use the context lemma to show that $R[t]{\Downarrow} \Leftrightarrow R[t']{\Downarrow}$. It is obvious from the definition of normal order reduction that the first reduction steps of $R[t]$ as well as $R[t']$ are to shift the top environment of t (or t') to the top environment of $R[t]$, or $R[t']$, respectively. Then the part $x_1 = y_1, x_2 = y_1, Env_1$ is the same in the top environment. The rest of the top environment is denoted in the following as Env_3.

Let $R[t]{\Downarrow}$. Assume first that y_1 is not used in the reduction, i.e. it is never in a reduction context in any term in the reduction sequence, and we also have $R[(\texttt{letrec}\ x_1 = \texttt{Bot}, x_2 = \texttt{Bot}, Env_1, Env_3\ \texttt{in}\ s)]{\Downarrow}$. Then $R[(\texttt{letrec}\ x_1 = \texttt{Bot}, x_2 = \texttt{Bot}, Env_1, Env_2, Env_3\ \texttt{in}\ s)]{\Downarrow}$, since (gc) can be used as program transformation to remove Env_2. From $R[(\texttt{letrec}\ x_1 = \texttt{Bot}, x_2 = \texttt{Bot}, Env_1, Env_2, Env_3\ \texttt{in}\ s)] \leq_c R[(\texttt{letrec}\ x_1 = y_1, x_2 = z_1, Env_1, Env_2, Env_3\ \texttt{in}\ s)]$, we derive $R[(\texttt{letrec}\ x_1 = y_1, x_2 = z_1, Env_1, Env_2, Env_3\ \texttt{in}\ s)]{\Downarrow}$.
The remaining case is that y_1 is used in the reduction. We fix a normal order reduction *Red* to WHNF. This reduction sequence is modified by replacing every (case)-reduction by $\xrightarrow{abs} \cdot \xrightarrow{case-cx}$ having the same effect as the (case). The constructed reduction sequence is denoted as Red'. Then we first distinguish the reduction steps in Red' 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 the following reduction sequence of $R[t']$, where every Env_1-related reduction step in Red' is also done (as a renamed copy) as a corresponding reduction in Env_2 after the Env_1-reduction. In Red' there may be (cp)-reductions copying abstraction from Env_1 to positions in Env_3 or s. These may be modified by copying from Env_2. Other reductions remain the same perhaps up to the names of the variables y_i, z_i. The invariant is that after merging y_i, z_i for all i and successive reductions on Env_1, Env_2, we obtain the reduction for t.
We have constructed a reduction sequence of $R[t']$ to a WHNF using reductions of the calculus and external reductions. From Lemma 14.2 and Theorem 10.33 we obtain $R[t']{\Downarrow}$.

In order to prove the other direction, let $R[t']{\Downarrow}$.
We fix a normal order reduction *Red* of $R[t']$ to a WHNF. The idea is to synchronize the reduction steps that occur in the environments Env_1, Env_2, perhaps adding reductions if necessary, and translating (case)-reductions into (abs) and (case-cx) if necessary.

Since the environments Env_1, and Env_2 are equal up to a variable renaming, we speak of *corresponding* terms, positions and reductions.
Our first observation is that we can recognize the successor environments of the environments Env_1 and Env_2 in the expressions in the normal order reduction sequence of $R[t']$. The construction will keep this correspondence property. Note that these environments may have more bindings than the original environments.
We want to argue that we can construct a reduction sequence to a WHNF with the following property:
Every reduction making a modification in Env_1 is immediately followed by a reduction that makes the corresponding modification in Env_2 and vice versa, and all reductions are in surface contexts, are base-reductions, (case-cx)-reductions or (abs)-reductions.
We show this by induction on the pair $(\mathrm{rl}\sharp(\cdot), \mu_{lll}(t))$ of the reduction Red, where t is the first term.
If the reduction does not modify the environments Env_1, Env_2, then we leave the reduction where it is and treat the next reduction step to the right.
Consider the first normal order reduction step that modifies a part of Env_1 or Env_2 (or their successor-environments).

1. If the reduction is a (lll)-reduction, then do the same for the other environment. It is only possible that an inner part of Env_1 may add bindings to Env_1. The same for Env_2, respectively.
2. If the reduction is completely within, say, Env_1, make the same reduction for Env_2. Theorem 12.5 shows that we can use induction on the length.
3. A (cp) into Env_1 or Env_2 is not possible due to the conditions on variable occurrences.
4. If the reduction is a (case) where the inner redex is in Env_1, but the redex is external, then split it into an (abs) followed by a (case-cx), and make the corresponding (case-cx) also for Env_2. Since a (case-cx) can be simulated by (case) with following (cpx) and (gc), Theorem 12.5 shows that we can use induction on the length $\mathrm{rl}\sharp(\cdot)$.
5. If the effect in Env_1 is an (abs), which comes from an external (case), then perform the corresponding (abs) also in Env_2. The same also for Env_2.

Note that there is no choice-reduction in the environments Env_1 nor Env_2.
Finally, we obtain a reduction sequence to a WHNF, using only surface reduction in the base calculus, some extra reductions, and (case-cx), where the reductions in Env_1, Env_2 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 only one of the 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, extra reductions and (case-cx). Now Lemma 14.2 and Theorem 10.33 show that $R[t]{\Downarrow}$. □

Corollary 15.3. *Let S be a surface context, and t be a deterministic subterm of* $(\texttt{letrec}\ x = t, Env\ \texttt{in}\ S[x])$ *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 15.2 with $(\texttt{letrec}\ x = y_1, x_2 = y_1, y_1 = t, Env\ \texttt{in}\ S[x_2])$, since the preconditions are satisfied. We obtain $(\texttt{letrec}\ x = y_1, x_2 = y_1, y_1 = t, Env\ \texttt{in}\ S[x_2]) \sim_c (\texttt{letrec}\ x = y_1, y_1 = t, x_2 = z_1, z_1 = t, Env\ \texttt{in}\ S[x_2]) \xrightarrow{cpx} \ldots \xrightarrow{gc} \ldots \xrightarrow{ucp} (\texttt{letrec}\ x = y_1, y_1 = t, Env\ \texttt{in}\ S[t])$. Now the proof can be finished using (gc) and (ucp). □

Corollary 15.4. *Let* $t = (\texttt{letrec}\ Env_1, Env_3\ \texttt{in}\ t_0)$, *such that* $FV(Env_1) \subseteq LV(Env_1)$ *and all expressions in* Env_1 *are deterministic, where* $Env_1 = \{y_1 = s_1, \ldots, y_n = s_n\}$, *and let* $t' = (\texttt{letrec}\ y_1 = (\texttt{letrec}\ Env_1'\ \texttt{in}\ y_1'), Env_2, Env_3\ \texttt{in}\ t_0)$, *where* Env_1' *is* Env_1 *where* y_i *are renamed into* y_i' *and* Env_2 *is* Env_1 *where* y_1 *is renamed into* y_1''. *Then* $t \sim_c t'$.

Proof. Proposition 15.2 is used as follows: $t = (\texttt{letrec}\ x_1 = y_1, x_2 = y_1, y_1 = s_1, \ldots, y_n = s_n, Env_3\ \texttt{in}\ t_0)$, such that x_1 does not occur in any subterm, and y_1 does not occur in Env or t_0. Then $t'' = (\texttt{letrec}\ x_1 = y_1, x_2 = y_1', y_1 = s_1, \ldots, y_n = s_n, y_1' = s_1', \ldots, y_n' = s_n', Env_3\ \texttt{in}\ t_0)$, and we obtain $t \sim_c t''$.
Since y_i' for $i = 1, \ldots, n$ occurs only in s_j', we can use (llet) backwards and obtain $t'' \sim_c (\texttt{letrec}\ x_1 = y_1, y_1 = s_1, \ldots, y_n = s_n, x_2 = (\texttt{letrec}\ y_1' = s_1', \ldots, y_n' = s_n'\ \texttt{in}\ y_1'), Env_3\ \texttt{in}\ t_0)$. Using (gc) and an appropriate renaming, we obtain the claim of the Corollary. □

Corollary 15.5. *Let* $t = (\texttt{letrec}\ Env_1, Env_3\ \texttt{in}\ t_0)$, *such that* $FV(Env_1) \subseteq LV(Env_1)$, *where* $Env_1 = \{y_1 = s_1, \ldots, y_n = s_n\}$ *is deterministic, and let* $t' = (\texttt{letrec}\ y_1 = (\texttt{letrec}\ Env_{1,i}\ \texttt{in}\ y_1'), \ldots, y_n = (\texttt{letrec}\ Env_{1,n}\ \texttt{in}\ y_n'), Env_3\ \texttt{in}\ t_0)$, *where* $Env_{1,i}$ *is* Env_1 *where* y_i *are renamed into* y_i'.
Then $t \sim_c t'$.

Proof. This follows from Corollary 15.4 by repeated application: After one application, the part $y_1 = (\texttt{letrec}\ Env_1'\ \texttt{in}\ y_1')$ can be seen as a part of the next rest-environment Env_3'. □
This corollary describes a technique of duplicating environments and isolating variables if the environment is deterministic and certain restrictions are satisfied.

Example 15.6. Consider the expressions

```
s := letrec repeat = ...  x = repeat 1 in (x,x)
t := letrec ...   in (repeat 1, repeat 1)
```

where the definition of `repeat` is done using a fixpoint combinator Y. Then $s \sim_c t$ follows easily from Corollary 15.3.
The corollary as it stands cannot be applied if the definition of `repeat` is recursive.

16 Abstract Terms

Abstract terms are expressions from LRA. Most of the extra constants (the abstract constants) used in strictness analysis algorithms, like $\bot$, $\top$, *Inf* etc. can be simulated within this language, so we use them only as abbreviations. These constants were already used in [Nöc92,Nöc93,vEGHN93,Sch00]. The notation for sets of terms was coined "demands" in [Sch00]. Note that only the subset of down-closed demands can be represented in our calculus.

16.1 Abstract Sets

We define structured terms in the abstract language that are used to represent sets of concrete expressions. These terms are able to represent for example all expressions ($\top$), or all infinite lists (*Inf*). In the following we will use () as a (dummy) constant.

Definition 16.1. *A closed* LRA*-term t is called an* ac-labeled term*, if it is of the form* (`letrec` $Env_{ac}, Env_{acg}, Env_{up}$ `in` r)*, where the environment for* abstract constant generators *is $Env_{acg} = \{y_1 = s_1, \ldots, y_m = s_m\}$, and the environment for abstract constants is $Env_{ac} = \{x_1 = t_1, \ldots, x_n = t_n\}$. The variables $LV(Env_{ac})$ are labelled "ac", and called ac-variables, and the variables in $LV(Env_{acg})$ are labeled "acg" and called acg-variables. The variables in $LV(Env_{up})$ are unlabeled and called up-variables.*
The following should hold:

- *The terms t_i must be of one of the forms*

$$(c\ (y_{i_1}\ ()) \ldots (y_{i_k}\ ())), y_i\ (), \bot$$

  where the variables y_{i_1}, y_i are in $LV(Env_{acg})$, i.e., are acg-variables.
- *The terms s_i, i.e., the abstract constant generators, are of the form $\lambda x.s_i'$, where $x \notin FV(s_i')$, and s_i' are nested $\oplus$-expressions where the inner terms are of one of three forms*

$$(c\ (y_{i_1}\ ()) \ldots (y_{i_k}\ ())), \lambda x.y_i\ (), \bot,$$

  where the variables y_{i_1}, y_i are in $LV(Env_{acg})$.
- *Env_{up} does not contain acg-variables*
- *Env_{acg} and Env_{ac} do not contain up-variables*
- *There may be occurrences of ac-variables in Env_{up} and r, but there are no occurrences of acg-variables y_i in Env_{up} or r.*

An abstract constant *a is a closed ac-labeled abstract term with an abstract set environment and is of the form* (`letrec` Env_{acg}, Env_{ac} `in` x_i)*, where x_i is an ac-variable.*

This means abstract constants do not contain `case`-expressions and `seq`-expressions, and every application is basically $((\lambda x.r)\ ())$, where r does not contain the variable x. They can be seen as a recursive description of sets of expressions.

Definition 16.2. *The set of concrete terms (i.e. in* LR*) of an abstract constant a is defined as:*

$$ctac(a) := \{t \mid t \in \mathcal{L}_{\mathrm{LR}}, t \leq_c a\}$$

Note that all the terms in $ctac(a)$ are deterministic.

Definition 16.3. *We give the definition in our formalism of some abstract constants that are also used in other papers on strictness analysis.*
We write the $\oplus$-expressions without brackets.

$$\begin{array}{lll}
\top := & \texttt{letrec} & \\
& \quad \text{topg} = & \lambda x.\ (\lambda y.\text{topg}\ ()) \\
& & \quad \oplus \bot \\
& & \quad \oplus\ (c_1\ \text{topg}\ ()\ldots\text{topg}\ ()) \\
& & \quad \oplus\ \ldots \\
& & \quad \oplus\ (c_N\ \text{topg}\ ()\ldots\text{topg}\ ()) \\
& \texttt{in}\ \text{topg}\ () &
\end{array}$$

$$\begin{array}{lll}
\mathit{Inf} := & \texttt{letrec} & \\
& \quad \text{topg} = & \lambda x.\ (\lambda y.\text{topg}\ ()) \\
& & \quad \oplus \bot \\
& & \quad \oplus\ (c_1\ \text{topg}\ ()\ldots\text{topg}\ ()) \\
& & \quad \oplus\ \ldots \\
& & \quad \oplus\ (c_N\ \text{topg}\ ()\ldots\text{topg}\ ()) \\
& \quad \text{infg} = & \lambda x.\ \bot \\
& & \quad \oplus\ (\text{topg}\ () : \text{infg}\ ()) \\
& \texttt{in}\ \text{infg}\ () &
\end{array}$$

where $c_1, c_2, \ldots c_N$ are all the constructors in the language and where : *denotes the binary list-constructor.*

Note that $(\lambda y.\text{topg}\ ())$ cannot be used for a representation of abstractions in LRA, but it works if restricted to LR.
It follows from Definition 16.1 that the ac-variables do not reference each other via bindings.
We have to define the effect of reduction rules on the ac-labeling:

Definition 16.4. *If a reduction rule is applied to an expression, then in general the labeling does not change with the following exception. If a (case)- or (abs)-reduction in the ac-part, or a (case)-reduction in the upper part generates a binding in the top* `letrec` $x_0 = c\ x_1 \ldots x_n$*, where x_0 is an ac-variable and the variables $x_i, i = 1, \ldots, n$ are fresh ones, then remove the ac-label from x_0, i.e. make it into an up-variable, label the variables $x_i, i = 1, \ldots, n$ as ac-variables and thus the binding is shifted into the upper part. A (cp)-reduction that copies an abstraction from the ac-part into the upper part is forbidden. A base reduction that is not forbidden is also called* ac-reduction.

The special (cp)-reduction above is forbidden, since it is not compatible with the intention that ac-variables represent sets of deterministic terms.

Lemma 16.5. *Let t be an ac-labeled term. According to Definition 16.1 the following reduction sequences $t \xrightarrow{*} t'$ transform t into an ac-labeled term.*

1. *A reduction step (seq), (lbeta), (case), (cp), (lll) that only modifies the upper part.*
2. *A reduction step (case) in the upper part with an (abs)-effect in the ac-part with subsequent relabeling.*
3. *A reduction (n,cp) copying an acg-function into the ac-part, followed by appropriate (n,lbeta), (n,lll), (n, choice)*.*

Proof. This follows by a case analysis.
We only go through the reductions that may change Env_{ac}:
If the effect in the ac-part is an (abs), then a binding $x = (c\ t_1 \dots t_n)$ may be transformed into several bindings $x = (c\ x_1 \dots x_n), x_1 = t_1, \dots, x_n = t_n$, where we let x_i be new ac-variables. In this case, $x = (c\ x_1 \dots x_n)$ is moved into the upper part, and x is no longer an ac-variable, but an up-variable. If there is a sequence (cp), (lbeta) within Env_{ac}, then subsequent (lll) and choice-reduction will transform the environment into a form that conforms to the requirements. Note that the reductions may leave redundant bindings $x = ()$ in the upper part.
□
The following shows that the definition of $\top$ is indeed contextually greater than any (possibly open) expression.

Proposition 16.6. *Let $\top$ be defined as above in Definition 16.3. Then the following holds:*

1. *For all expressions t: $t \leq_c \top$*
2. *For all concrete expressions t: If t has a CWHNF with top constructor c of arity n, then $t \leq_c (c\ \top \dots \top)$.*
3. (a) *If a concrete expression t allows to construct a finite sequence $t_i, i = 1, 2 \dots, n$ of expressions as follows $t = t_1$, $t_i \xrightarrow{n,*} (s_{i+1} : t_{i+1})$, and $t_n \Uparrow\Uparrow$ then $t \leq_c Inf$.*
   (b) *If a concrete expression t allows to construct an infinite sequence $t_i, i = 1, 2 \dots$ of expressions as follows $t = t_1$, $t_i \xrightarrow{n,*} (s_{i+1} : t_{i+1})$, then $t \leq_c Inf$.*

Proof. We use Conjecture 14.5 and Proposition 14.6. We can assume that t is closed by possibly treating all the terms ($\texttt{letrec}\ x_1 = t_1, \dots, x_n = t_n\ \texttt{in}\ t$) where t_i are closed. Then we use the definition of $\leq_b$ and co-induction: For reduction we have to use the bhv-reductions and BWHNFs. If t has no WHNF, then it also has no BWHNF and the claim holds. If t has a WHNF, then we can argue for every BWHNF of t as follows: In every case, we bhv-reduce the expression for $\top$ by (lbeta), (llet) and some (choice)-reductions and the other reductions as deep as necessary until it is in BWHNF. The BWHNF can be seen as representing a

tree built from constructors, where the leaves are $\bot$, abstractions or constructor constants. We can reduce $\top$ to a tree that is at least as deep as the corresponding tree for the BWHNF for t. In every case we use the definition of $\leq_b$, and then obtain the same situations as before, in general with a different t', but the term for $\top$ is reproduced. Hence we can use co-induction, and obtain that $t \leq_b \top$, and hence $t \leq_c \top$.
The proof for *Inf* follows the same co-induction scheme as for $\top$. □

Remark 16.7. The definition of abstract constants does not cover the non-down-closed demands in [Sch00] like Fin, the abstract constant representing all finite lists.
We conjecture that the Moran-Sands-preorder [MSC99] which has an extra condition on non-termination in the definition of the contextual order, together with a behavioral preorder may be an alternative mechanism to provide Fin.

17 The Calculus for Strictness Detection

Intuitively, strictness of a function f is detected if normal order reduction of $(f\ \bot)$ in the abstract language can only yield $\bot$ or nontermination. This may be represented by the explicit abstract constant $\bot$ itself, or by a proof that normal order reduction will not terminate. Reduction of expressions (`case` $\top$ ...) will require a case analysis, which is done 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.
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 abstract constants apart from $\top$ and $\bot$, even more complicated analyses are possible like a test for tail-strictness, or strictness under certain conditions.
We also want to use strictness of built-in functions and the results of previous analyses. Therefore we assume that there are already 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 $x = f$ in the top level `letrec`, and the variable x is not labeled as ac or acg.
Note that `seq` allows explicitly making any argument of a function a strict argument.

17.1 Reductions on Abstract Terms

Definition 17.1. *The reduction rules that treat the constant $\bot$ are defined in figure 4. Note that these reductions are permitted in all contexts.*

Proposition 17.2. *If $t \to t'$ by a $\bot$-reduction as defined in Figure 4, then*

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

Fig. 4. Reduction rules for $\bot$

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

Proof. Contextual equivalence follows from Corollary 11.5 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 $\bot$ nor x is required. The values of x or $\bot$ 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 $\bot$. Thus the context lemma shows contextual equivalence.

The claim on the lengths of reductions can be proved as follows. In a terminating normal order reduction to WHNF, the subterm $\bot$, 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.

□

In the following, the bot-reduction rules as defined in Definition 17.1 (figure 4) are used where instead of $\bot$, we allow a variable bound to $\bot$ in the top environment. This does not really make a difference.

17.2 Concretizations of ac-Labeled Terms

We define concretizations s of ac-labeled terms t as deterministic 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 ($\texttt{letrec}\ \mathit{Env}, \mathrm{top} = \mathrm{topg}(), \mathrm{inf} = \mathrm{infg}()\ \texttt{in}\ \mathrm{top} : \mathrm{inf}$).

Definition 17.3. *Let $t = (\texttt{letrec}\ \mathit{Env}_{t,ac}, \mathit{Env}_{t,up}\ \texttt{in}\ t')$ be an ac-labeled closed abstract term. We assume that $\mathit{Env}_{t,ac} = \mathit{Env}_{t,acb} \uplus \mathit{Env}_{t,acg}$ where $\mathit{Env}_{t,acb}$ are the ac-bindings.*
Then the set of concretizations *$\gamma(t)$ is defined as follows:*
Let $s = (\texttt{letrec}\ \mathit{Env}_s\ \texttt{in}\ s')$ be a closed concrete term. Then $s \in \gamma(t)$ iff the following holds:

- *There is a split of the environment Env_s into $\mathit{Env}_s = \mathit{Env}_{s,ac} \uplus \mathit{Env}_{s,up}$, with $FV(\mathit{Env}_{s,ac}) = \emptyset$, such that the properties below hold. Note that s is not an ac-labeled term, and that $\mathit{Env}_{s,ac}$ is any appropriate part of the environment.*
- *Let $s_1 := (\texttt{letrec}\ \mathit{Env}_{s,up}\ \texttt{in}\ s')$, $t_1 := (\texttt{letrec}\ \mathit{Env}_{t,up}\ \texttt{in}\ t')$. Let $\{y_1, \ldots, y_k\} = FV(t_1)$ be the ac-variables occurring in terms of the upper part of t.*
  The set $FV(s_1)$ consists of k variables $\{x_1, \ldots, x_k\}$. For ρ defined by $\rho(x_i) = y_i$ for $i = 1, \ldots, k$, the equation $\rho(s_1) = t_1$ holds, i.e. s_1 and t_1 are equal up to a renaming of variables.
- *For every i: $(\texttt{letrec}\ \mathit{Env}_{s,ac}\ \texttt{in}\ x_i) \in \mathit{ctac}(\texttt{letrec}\ \mathit{Env}_{t,ac}\ \texttt{in}\ y_i)$, i.e., $(\texttt{letrec}\ \mathit{Env}_{s,ac}\ \texttt{in}\ x_i) \leq_c (\texttt{letrec}\ \mathit{Env}_{t,ac}\ \texttt{in}\ y_i)$.*

Proposition 17.4. *Let t be an ac-labeled closed term and let $s \in \gamma(t)$. Then $s \leq_c t$.*

Proof. We use the notation of Definition 17.3. Then $t = (\texttt{letrec}\ \mathit{Env}_{t,ac}, \mathit{Env}_{t,up}\ \texttt{in}\ t'), s = (\texttt{letrec}\ \mathit{Env}_{s,ac}, \mathit{Env}_{s,up}\ \texttt{in}\ s')$. For all $i = 1, \ldots, k$ the term $(\texttt{letrec}\ \mathit{Env}_{s,ac}\ \texttt{in}\ x_i)$ is closed and is deterministic, since it contains no $\oplus$-expressions.
For all $i = 1, \ldots, k$ the term $(\texttt{letrec}\ \mathit{Env}_{t,ac}\ \texttt{in}\ y_i)$ is closed, and for every $i = 1, \ldots, k$, there is a binding $y_i = u_i$ in $\mathit{Env}_{t,ac}$. The term t can be transformed into the following form: $(\texttt{letrec}\ y_1 = r_1, \ldots, y_k = r_k, \mathit{Env}_{rest}\ \texttt{in}\ t')$, where $r_i = (\texttt{letrec}\ \mathit{Env}_{acg}\ \texttt{in}\ u_i)$ for $i = 1, \ldots, k$ are closed, using (gc) and by moving (i.e. appropriately copying) the acg-environment completely into r_i, which are correct program transformations. We have for all i: $(\texttt{letrec}\ \mathit{Env}_{t,ac}\ \texttt{in}\ y_i) \sim_c (\texttt{letrec}\ \mathit{Env}_{t,acg}, y_i = u_i\ \texttt{in}\ y_i) \sim_c (\texttt{letrec}\ \mathit{Env}_{t,acg},\ \texttt{in}\ u_i) = r_i$.
Now we use the condition on *ctac*, and that $\leq_c$ is a precongruence, and obtain (after an appropriate renaming ρ):

$$
\begin{array}{ll}
 & \texttt{letrec}\ y_1 = (\texttt{letrec}\ \mathit{Env}_{s,ac}\ \texttt{in}\ x_1), \ldots, y_k = (\texttt{letrec}\ \mathit{Env}_{s,ac}\ \texttt{in}\ x_k), \\
 & \qquad \mathit{Env}_{t,up} \\
 & \texttt{in}\ t' \\
\leq_c & \texttt{letrec}\ y_1 = r_1, \ldots, y_k = r_k, \mathit{Env}_{t,up}\ \texttt{in}\ t'
\end{array}
$$

Corollary 15.5, the correctness of (gc) and and applying the assumptions and the renaming ρ shows that

$$
\begin{array}{ll}
& \texttt{letrec } y_1 = (\texttt{letrec } Env_{s,ac} \texttt{ in } x_1), \ldots, y_k = (\texttt{letrec } Env_{s,ac} \texttt{ in } x_k), \\
& \qquad Env_{t,up} \texttt{ in } t' \\
\sim_c & (\texttt{letrec } Env'_{s,ac}, Env_{t,up} \texttt{ in } t') \\
& \quad \text{where } Env'_{s,ac} \text{ is } \{y_1 = \rho^*(q_1), \ldots, y_n = \rho(q_n)\} \\
& \qquad \text{if } Env_{s,ac} = \{x_1 = q_1, \ldots, x_n = q_n\} \\
& \qquad \text{and } \rho^* = \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\} \\
\sim_c & (\texttt{letrec } Env_{s,ac}, Env_{s,up} \texttt{ in } s') \qquad \text{(by renaming)} \\
= & \ s
\end{array}
$$

If we put the $\sim_c$-relations and $\leq_c$-relations together, we obtain $s \leq_c t$. □

Example 17.5. In this example we use a short-hand notation for expressions by omitting top-level definitions of functions and of infg and topg.
To show

$$
\begin{array}{c}
(\texttt{letrec } x_1 = 1, x_3 = (\texttt{repeat } x_1) \texttt{ in } (x_1 : x_3)) \\
\in \gamma((\texttt{letrec } \text{top0} = \text{topg } (), \text{inf0} = \text{infg } () \texttt{ in } (\text{top0} : \text{inf0})))
\end{array}
$$

requires checking whether $1 \in ctac((\texttt{letrec } \text{top0} = \text{topg } () \texttt{ in } \text{top0}))$ and $(\texttt{letrec } x_1 = 1, x_3 = (\texttt{repeat } x_1) \texttt{ in } x_3) \leq_c (\texttt{letrec } \text{inf0} = \text{infg } () \texttt{ in } \text{inf0})$. The former follows from Proposition 16.6, for the latter we use the definition of the behavioral preorder, Conjecture 14.5 and co-induction.

Example 17.6. Let $s = (\texttt{letrec } x = (y : x), y = 1 \texttt{ in } (c\ x\ x\ y))$, and let $t = (\texttt{letrec } \ldots \texttt{ in } (c \text{ inf inf top}))$. Since inf, top are ac-variables, we have to compare using $\rho = \{x \mapsto \text{inf}, y \mapsto \text{top}\}$, whether $\rho(c\ x\ x\ y) = (c \text{ inf inf top})$, which holds.
There are three further checks required:
The first is membership of $(\texttt{letrec } z = x, x = (y : x), y = 1 \texttt{ in } z)$ in $ctac(\texttt{letrec } z = \text{inf}, \ldots \texttt{ in } z)$, which is true, and can be verified using $\leq_b$, using Conjecture 14.5 and co-induction. However, we give no algorithm for this test in this paper.
The second is the same, whereas the third is membership of $(\texttt{letrec } z = y, x = (y : x), y = 1 \texttt{ in } z)$ in $ctac(\texttt{letrec } z = \text{top}, \ldots \texttt{ in } z)$, which also holds.

Example 17.7. There may be abstract terms with a reduction to WHNF, but without concretizations:
$t = (\texttt{letrec } otg = \lambda x.(1 \oplus 2), f = \lambda x.otg\ () \texttt{ in if } (f\ 0) + (f\ 0) == 3 \texttt{ then } 1 \texttt{ else } \bot)$, where we assume that the variable f is ac-labeled. There is a reduction to WHNF, however, there is no concretization s of t that leads to a WHNF: the possible concrete expressions bound to f are $\bot, \lambda x.\bot, \lambda x.1, \lambda x.2$, but every possibility leads to nontermination.

17.3 Subset Relationship for Abstract Terms

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

Definition 17.8. *Let s, t be two ac-labeled closed abstract terms.*
Then $s \subseteq_\gamma t$ iff $\gamma(s) \subseteq \gamma(t)$.

We give a sufficient condition for $\subseteq_\gamma$:

Lemma 17.9. *Let $s = (\texttt{letrec}\ Env_s\ \texttt{in}\ s')$ and $t = (\texttt{letrec}\ Env_t\ \texttt{in}\ t')$ be two closed abstract terms that are ac-labeled. Then $s \subseteq_\gamma t$ if the following holds:*

- *The environment Env_s is split into $Env_s = Env_{s,ac} \uplus Env_{s,up}$, where $Env_{s,ac}$ is the ac-labeled part.*
- *The environment Env_t is split into $Env_t = Env_{t,ac} \uplus Env_{t,up}$, where $Env_{t,ac}$ is the ac-labeled part.*
- *Let $s_1 = (\texttt{letrec}\ Env_{s,up}\ \texttt{in}\ s')$, $t_1 = (\texttt{letrec}\ Env_{t,up}\ \texttt{in}\ t')$, Let $\{y_1, \ldots, y_k\} = FV(t_1)$ be the ac-variables occurring in terms of the upper part of t.*
  The set $FV(s_1)$ consists of k variables $\{x_1, \ldots, x_k\}$. Let $\rho(x_i) = y_i$ for $i = 1, \ldots, k$. The equation $\rho(s_2) = t_2$ must hold.
- *For every i: $ctac(\texttt{letrec}\ Env_{s,ac}\ \texttt{in}\ x_i)) \subseteq ctac((\texttt{letrec}\ Env_{t,ac}\ \texttt{in}\ y_i)$.*

Proof. The conditions can directly be matched with the conditions in Definition 17.3. □

18 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.

Definition 18.1. *The algorithm SAL uses a finite family of finite sets, the strict functions, $SF^{n,i}$, that 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. The edges may be labeled or not.
The algorithm SAL starts with a directed graph consisting only of one node labeled with the 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 18.4 below. For every node added, we assume that the simplification rules (i.e. (lwas), (llet), (gc), (cpax)) and the bot-reduction rules (see 17.1) 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.

We identify syntactically the strict and *proper* strict positions w.r.t. the sets $SF^{n,i}$.

Definition 18.2. *Assume given the sets $SF^{n,i}$. Then 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 $t \equiv S[f\ t_1 \dots t_n]$, $S \in \mathcal{SFS}(t)$ and $f \in SF^{n,i}$, then $S[f\ t_1 \dots t_{i-1}[\cdot]t_{i+1} \dots t_n] \in \mathcal{SFS}(t)$*
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 $S[(\texttt{letrec}\ Env\ \texttt{in}\ [\cdot])] \in \mathcal{SFS}(t)$.*

We will use the term *strict position ac-reduction* (*spac-reduction*), if the reduction is an ac-reduction and the inner redex is in an SF-strict context. This reduction is not unique, since there may be several SF-strict positions where a spac-reduction may be possible.
Note that a subterm in an SF-strict context is also a strict subterm. The following condition is used to syntactically detect *proper* strict contexts.

Definition 18.3. *Given the sets $SF^{n,i}$, the* proper SF-strict *contexts $\mathcal{PSFS}(t)$ of a term t are defined as follows:*

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

The following rules use the effective test $\subseteq\subseteq_\gamma$, which must be an algorithm with the property that $s \subseteq\subseteq_\gamma t \Rightarrow s \subseteq_\gamma t$.
For an abstract term t, let $\mathrm{simp}(t)$ be the result of exhaustively applying simplification rules and bot-reduction rules.

Definition 18.4. *The non-deterministic construction rules of SAL are:*

nred *Let a leaf L be labeled with t, and let t have a spac-reduction to t', where the reduction is not a choice-reduction and not an (lll)-reduction. Let $t'' := \mathrm{simp}(t')$. Then generate a new node L' with term t'' and add a directed edge from L to L'. If the reduction is in the upper part, and if it is a (case), (lbeta), or a (seq), then the edge is to be labeled with the kind of reduction.*

nchoice *Let a leaf L be labeled with t, where t has a spac-reduction (choice-r) to t_r, and (choice-l) to t_l for the same redex, then let $t'_r := \mathrm{simp}(t_r)$ and $t'_l := \mathrm{simp}(t_l)$ and generate two nodes L_r, L_l which are labeled with t'_r, t'_l, respectively. Add directed edges from L to L_l, L_r that are not labeled.*

ichoice *The same operation as* nchoice*, but for a non-spac-reduction, where the redex must be on the application surface.*

ired *Let a leaf L be labeled with t and let t have a non-spac-reduction to t', where the reduction must be an ac-reduction, and may be a (case), (seq), (lbeta), or (cp). Let $t'' := \mathrm{simp}(t')$. Then generate a new node L' with term t'' and add an unlabeled directed edge from L to L'.*

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 $t_1 \subseteq\subseteq_\gamma t_2$, 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 $\bot \neq t = (\texttt{letrec}\ Env\ \texttt{in}\ t_{11})$, 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 $C = (\texttt{letrec}\ Env\ \texttt{in}\ C') \in \mathcal{PSFS}(t)$ with $t = C[t_{12}]$ we have $(\texttt{letrec}\ Env\ \texttt{in}\ t_{12}) \subseteq\subseteq_\gamma t_2$, or*
2. *For $C = (\texttt{letrec}\ y = C', Env\ \texttt{in}\ t_{11}) \in \mathcal{PSFS}(t)$ with $t = C[t_{12}]$ we have $(\texttt{letrec}\ y = C'[t_{12}][x/t_{12}], Env[x/t_{12}], x = t_{12}\ \texttt{in}\ x) \subseteq\subseteq_\gamma t_2$.*

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

generalize *Given a leaf L with label t, construct a new term t' as follows: add a binding* top = topg () *to the top* `letrec`*-environment, select a subterm of t on a surface position and in the upper part of t and replace this subterm by* top*, where* top *is a new variable. Add an unlabeled directed edge from L to the node L' with term 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 the abstract reduction sequence $(\texttt{letrec}\ \ldots, x = \lambda y.topg\ ()\ \texttt{in}\ C[x\ s]) \to (\texttt{letrec}\ \ldots, x = \lambda y.top\ \texttt{in}\ C[(\lambda y.topg\ ())\ s])$ is forbidden. Instead, an application of generalize has to produce the desired effect: $(\texttt{letrec}\ \ldots, x = \lambda y.topg\ (), z = topg\ ()\ \texttt{in}\ C[z])$.

18.1 Correspondence between Concrete and Abstract Terms

Lemma 18.5. *Let t be an abstract ac-labeled term and let x be a strict subterm at position p in the upper part of t. Let $s \in \gamma(t)$ and let x be the subterm in s corresponding to position p. Then x is also a strict subterm of s.*

Proof. The definition of strictness implies that $t[\bot/p] \sim_c \bot$. Since the definition of concretization implies that s and t have syntactically the same upper part up to variable renaming, the term $s[\bot/p]$ is a concretization of $t[\bot/p]$. □

Theorem 18.6. *Let s be a closed concrete term, and let t be a closed ac-labeled abstract term, such that $s \in \gamma(t)$ and $s{\Downarrow}$.*

1. (nred) *Let* $t \xrightarrow{sp,a} t'$ *be a spac-reduction with* $a \in \{(case), (seq), (lbeta), (cp)\}$. *Then there are two cases:*
   (a) *The reduction is in the upper part. Then there is a term* s', *such that* $s \xrightarrow{sp,a} s'$, $s' \in \gamma(t')$, *and* $\mathrm{rl}\sharp\sharp(s) > \mathrm{rl}\sharp\sharp(s')$ *if* $a \in \{(case), (seq), (lbeta)\}$ *and* $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s')$ *if* $a = (cp)$.
   (b) *The reduction is in the ac-part. Then* $s' = s \in \gamma(t')$ *or* $s \sim_c s' \in \gamma(t)$ *and* $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s')$.
2. *If the rule (*nchoice*) or (*ichoice*) was applied to the term* t, *then* $t \xrightarrow{choice-l} t_1$ *and* $t \xrightarrow{choice-r} t_2$, *and there exists an* s' *such that* $s \sim_c s'$ *and either* $s' \in \gamma(t_1)$ *or* $s' \in \gamma(t_2)$, *and* $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s')$.
3. (ired) *Let* $t \xrightarrow{a} t'$ *with an a-reduction that is (case), (seq), (lbeta), or a (cp), and that is a non-spac-reduction, but an ac-reduction. Then there is some* s' *with* $s \xrightarrow{*} s'$, $s' \in \gamma(t')$ *and* $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s')$.
4. *If* $t \xrightarrow{a} t'$ *with a simplifying reduction a, then* $s \in \gamma(t')$, *or there is an* s' *with* $s \xrightarrow{a} s'$, *and* $s' \in \gamma(t')$ *and* $\mathrm{rl}\sharp\sharp(s) = \mathrm{rl}\sharp\sharp(s')$.
5. *If* $t \xrightarrow{a} t'$ *with a bot-reduction rule, then there is a concretization* s' *that is a closed concrete term with* $s' \sim_c s$, $\mathrm{rl}\sharp\sharp(s) = \mathrm{rl}\sharp\sharp(s')$ *and* $s' \in \gamma(t')$.
6. *If* t' *is the result of a generalization applied to* t, *then there is a concretization* s' *with* $s' \xrightarrow{ucp} s$, $s' \in \gamma(t')$, *and* $\mathrm{rl}\sharp\sharp(s) = \mathrm{rl}\sharp\sharp(s')$.

Proof. 1.

(a) If the reduction of t is a (case), (seq), (lbeta), or (cp) in the upper part, then the reduction of the abstract term can also be performed in the concretization, since in the upper part, the terms must be α-equal. Since the functions SF are in the upper part, the reduction is also an spac-reduction $s \xrightarrow{a} s'$ iff it is one in t. Note that the rules $\{(case), (seq), (cp)\}$ can only be $\{(case\mathcal{S}), (seq\mathcal{S}), (cp\mathcal{S})\}$ for spac-reductions. If $a \in \{(case\mathcal{S}), (seq\mathcal{S}), (lbeta)\}$, then Proposition 12.26 shows that $\mathrm{rl}\sharp\sharp(s) > \mathrm{rl}\sharp\sharp(s')$. Theorem 12.25 shows that $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s')$ if the reduction is a $(cp\mathcal{S})$.

(b) Let the reduction be in the ac-part.

Then there are two cases: if there is no change in the boundary between ac-part and upper part, then s remains a concretization, since *ctac* is invariant under non-choice-reduction, hence $s' = s$.

If there is a change in the boundaries between ac-part and upper part in t, then the conditions on the concretization show that there is a variable, say with index 1 such that $(\texttt{letrec}\ Env_{s,ac}\ \texttt{in}\ x_{s,1}) \leq_c (\texttt{letrec}\ Env_{t,ac}\ \texttt{in}\ x_{t,1})$, and after one reduction step the binding for $x_{t,1}$ in the reduct is of the form $x_{t,1} = r_t$ where r_t is a constructor application. Since $x_{t,1}$ is in a reduction context in t, the variable $x_{s,1}$ in $Env_{s,ac}$ is also in a reduction context. The assumption $s{\Downarrow}$ implies that $(\texttt{letrec}\ Env_{s,ac}\ \texttt{in}\ x_{s,1})$ will normal order reduce to a term where the binding for $x_{s,1}$ is of the form $x_{s,1} = r_s$, where r_s is a value or bound to a value. In the case that it is bound to a value, we can use (cpx),

(abs), (cp) and (cpcx) to copy the value to this position resulting in s'. Theorem 12.25 shows that we still have $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s')$.
Now it remains to show what happens when the boundaries are moved. The values r_t, r_s must have the same top-level constructor. The syntactic restrictions on the ac-part of t ensure that $r_t = c\ y_{t,1} \ldots y_{t,n}$. We also have $r_s = c\ r_{s,1} \ldots r_{s,n}$. Using (abs), we get $r_s = c\ y_{s,1} \ldots y_{s,n}$ with an additional binding part $\{y_{s,1} = r_{s,1}, \ldots y_{s,n} = r_{s,n}\}$. Since $\leq_c$ is a congruence, we obtain that the $\leq_c$-conditions hold.

2. If the reduction is a (choice)-reduction in a surface context, then it is in the ac-part, and s is also a concretization of one of the reducts by Theorem 14.17. If the boundary between ac-part and upper part changes, then the same reasoning as above shows that there is some s' with $s \sim_c s'$, $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s')$, and either $s' \in \gamma(t_l)$ or $s' \in \gamma(t_r)$.
3. In the case of an (ired)-reduction, the same reasoning as for (nred) applies with the difference that it is only possible to derive $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s')$ using Theorem 12.25.
4. Let the reduction be a simplification, i.e., (llet), (gc), (lwas) or (cpax). If it is completely in the upper part, then it can be performed in t as well as in s. Theorem 12.25 shows that the claim on $\mathrm{rl}\sharp\sharp(\cdot)$ holds. If it is in the ac-part, then Theorem 10.23 and 10.31 show that *ctac* is not changed.
5. Let $t \xrightarrow{a} t'$ with a bot-reduction rule. From Lemma 17.2 it follows that $\sim_c$ is stable w.r.t. this reduction, and also that $\mathrm{rl}\sharp\sharp(.)$ is unchanged. It is not possible that the corresponding subterms are used in a terminating reduction of a concretization, since the reduction of the subterm would either not terminate or produce an error, which is treated as equivalent to non-termination.
6. Let t' be a term. If t' is 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 top environment. Note that the replaced term has only free variables that are bound in the top `letrec`. Since $\top$ is maximal by Proposition 16.6, we obtain $s' \in \gamma(t')$. From Proposition 12.14 we derive that $\mathrm{rl}\sharp\sharp(s) = \mathrm{rl}\sharp\sharp(s')$.

□

Proposition 18.7. *Let (N, N') be an edge introduced by one of the subsume-rules. Let t be the term at N and t' be the term at N'. Let $s \in \gamma(t)$. Then the following holds:*

1. *If the node is generated by the rule (*subsume*), then there is a concretization $s' \in \gamma(t')$ with $\mathrm{rl}\sharp\sharp(s') \leq \mathrm{rl}\sharp\sharp(s)$.*
2. *If the node is generated by the rule (*subsume2*), then there is a concretization $s' \in \gamma(t')$ with $\mathrm{rl}\sharp\sharp(s') < \mathrm{rl}\sharp\sharp(s)$.*

Proof. For the rule (subsume), this holds, since $t' \subseteq \subseteq_\gamma t$. For the rule (subsume2), we apply Proposition 12.33.

□

Corollary 18.8. *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 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 neither (*nchoice*) nor (*ichoice*), then there is a concretization $s' \in \gamma(t')$ and $\mathrm{rl}\sharp\sharp(s) \geq \mathrm{rl}\sharp\sharp(s')$.*
3. *If the edge was generated using (*nchoice*) or (*ichoice*), let (N, N'') be the other edge generated by the rule, where t'' is the corresponding term of N''. Then $s \in \gamma(t')$ or $s \in \gamma(t'')$.*

19 Correctness of Strictness Detection

19.1 Main Theorems

Theorem 19.1. *Let t be a closed abstract term. If t leads to successful termination using SAL, then $s \in \gamma(t) \Rightarrow s{\Uparrow}$.*

Proof. Assume that there is a closed concrete term $s \in \gamma(t)$ that has a terminating normal order reduction.
Theorem 18.6 and Corollary 18.8 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' and term $t_{N'}$ with a concretization $s_{N'}$ 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 18.8. It is not possible that the initial concretization 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 a terminating concretization $s_{N,min}$. Since there is an outgoing edge, minimality shows that the edge cannot be labeled. However, since the graph is finite, we will find a cycle without labels, which does not exist due to the construction. Hence we have a contradiction. □

Corollary 19.2. *If (*`letrec` *$bot = \bot$* `in` *$f\ bot$) leads to successful termination using SAL, then f is strict in its argument.*

Proof. Since the term is already a concretization of itself, Theorem 19.1 implies that (`letrec` $bot = \bot$ `in` $f\ bot$) $\sim_c \bot$. Proposition 17.2 and correctness of (gc) show that (`letrec` $bot = \bot$ `in` $f\ bot$) $\sim_c f\ \bot \sim_c \bot$, hence by the definition of strictness, f is strict in its argument. □

Corollary 19.3. *If the term $t :=$*

$$\begin{array}{l}\texttt{letrec}\ topg = \ldots, bot = \bot, top_1 = topg\ (), \ldots, top_n = topg\ () \\ \quad \texttt{in}\ (f\ top_1\ \ldots\ top_{i-1}\ bot\ top_{i+1}\ \ldots\ top_n)\end{array}$$

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

Proof. The definition of strictness requires that for every deterministic expression $t_j, j = 1, \ldots, n$: $f\ t_1\ \ldots t_{i-1}\ \bot\ t_{i+1} \ldots t_n\ \sim_c\ \bot$. The term $f\ t_1\ \ldots t_{i-1}\ \bot\ t_{i+1} \ldots t_n$ is contextually equivalent to

$$s := \texttt{letrec}\ x_1 = t_1, \ldots, x_{i-1} = t_{i-1}, x_i = \bot, x_{i+1} = t_{i+1}, \ldots, x_n = t_n \\ \in\ f\ x_1\ \ldots x_{i-1}\ x_i\ x_{i+1} \ldots x_n\ ,$$

which follows using correctness of (lwas), (ucp), (gc) (see Theorem 10.31). We also have $s \in \gamma(t)$ by Proposition 16.6. Theorem 19.1 implies that $s \sim_c \bot$.
Now, by the definition of strictness, we obtain that f is strict in its i^{th} argument.
□
The following corollaries can be proved without using the conjecture 14.5:

Corollary 19.4. *Independent of Conjecture 14.5, the following holds. If* (`letrec` *bot* = ⊥ `in` *f bot*) *leads to successful termination using SAL without using any abstract constants, then f is strict in its argument.*

Proof. This follows from Corollary 19.2 and since the conjecture is only used in the correctness proof via Proposition 16.6, which is not required if no abstract constants are used in the analysis. □

Corollary 19.5. *Independent of Conjecture 14.5, the following holds. If* (`letrec` *bot* = ⊥ `in` *f bot*) *leads to successful termination using SAL without using the rule* generalize*, then f is strict in its argument.*

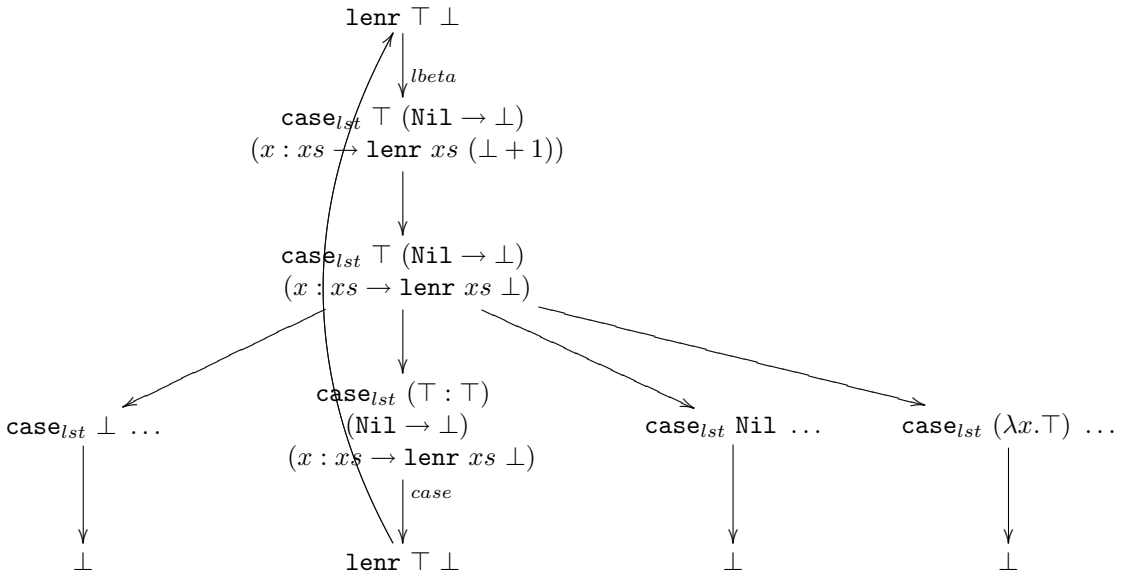
Proof. Only the rule generalizemay introduce the abstract constant ⊤, hence this follows from Corollary 19.4. □

20 Examples

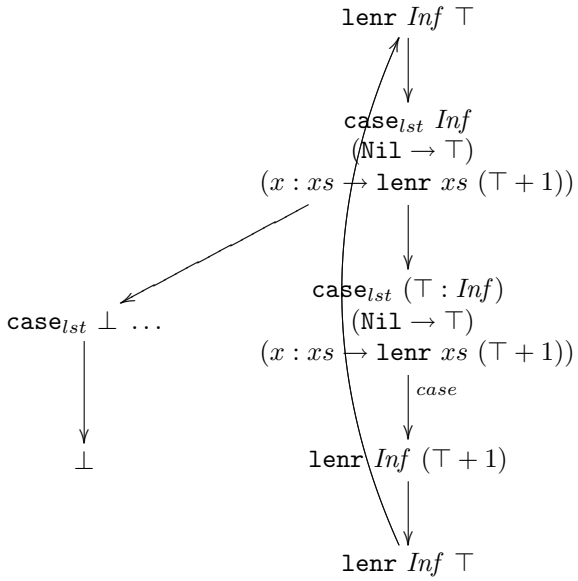
Example 20.1. We show that the tail-recursive length function (lenr) is strict in its second argument:

```
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 (`letrec` *top* = *topg* (), *bot* = ⊥ `in` (`lenr` *top bot*)) using an enclosing acg-environment.
We do not write the `letrec`-acg-environment, and we write ⊤ for a variable that is bound to an expression topg (), ⊥ for a variable that has a binding = ⊥, and *Inf* for a variable that is bound to infg (). For definition and properties see Definition 16.3 and Proposition 16.6.

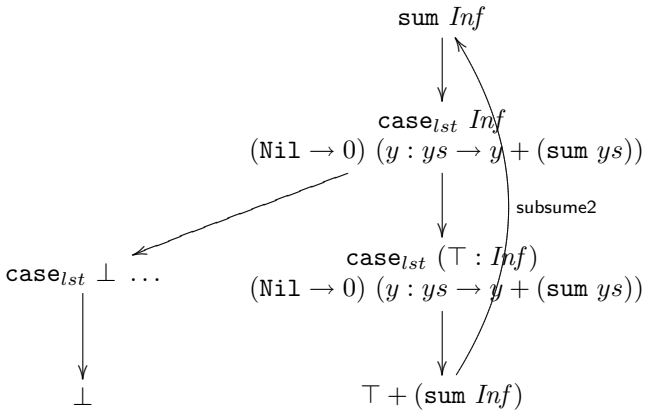


Example 20.2. Further properties of `lenr`.
The function `lenr` is tail-strict in the first argument. For definition of *Inf* and the properties see Definition 16.3 and and Proposition 16.6.



Example 20.3. 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.

Example 20.4. We want to show that sharing of abstract constants is 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 (`letrec` $top = \top, bot = \bot, \ldots$ `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.

21 Conclusion and Future Research

The paper gives a correctness proof for all the essential steps in the strictness analyzer based on abstract reduction as 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`, a non-deterministic $\oplus$ to represent sets and unions, and a contextual preorder. However, for applying it analyses that use abstract constants, we have to rely on the conjecture that our behavioral preorder is contained in the contextual preorder.
We are looking forward to see a proof of $\leq_b \ \subseteq \ \leq_c$ for LRA based on an extension of [Man04], since this would fully prove the correctness of the application of SAL and hence of Nöcker's algorithm.
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.
The representation of sets in the language may have other applications than abstract constants, for example it may lay a new foundation for a lazy lambda calculus with constraints [Man95] or for a representation of ambiguity in a lambda calculus which has potential applications in the representation of the semantics of natural languages.

References

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

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.

LPJ96. John Launchbury and Simon Peyton Jones. State in Haskell. *Journal of functional programming*, 1996. to appear.

Man95. L. Mandel. *Constrained Lambda Calculus*. Verlag Shaker, Aachen, Germany, 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.

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. Master's 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 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.

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.