Improvements in a Functional Core Language with Call-By-Need Operational Semantics

Manfred Schmidt-Schauß and David Sabel

Goethe-University, Frankfurt, Germany

Technical Report Frank-55

Revision V2 from November 30, 2015[1]

Abstract. An improvement is a correct program transformation that optimizes the program, where the criterion is that the number of computation steps until a value is obtained is decreased. This paper investigates improvements in both – an untyped and a polymorphically typed – call-by-need lambda-calculus with letrec, case, constructors and seq. Besides showing that several local optimizations are improvements, the main result of the paper is a proof that common subexpression elimination is correct and an improvement, which proves a conjecture and thus closes a gap in Moran and Sands' improvement theory. We also prove that several different length measures used for improvement in Moran and Sands' call-by-need calculus and our calculus are equivalent.

1 Introduction

Motivation and State of the Art Functional programming languages with lazy evaluation like Haskell [9] support declarative programming, i.e. they allow a definition of the intended results while the exact sequence of operations is left unspecified, and they provide a high-level of abstraction [6].

While there is no official formal semantics of Haskell, it is often loosely identified with an extended lazy lambda-calculus with call-by-name evaluation. However, all real implementations of Haskell use call-by-need evaluation – i.e. lazy evaluation extended by sharing to avoid duplicated evaluation of subexpressions.

For reasoning about program semantics, it is indifferent whether a call-by-name or call-by-need semantics is used, since both induce the same (equational) semantics. However, for reasoning on the resource consumption, the call-by-name evaluation does not fit to the real implementations and thus a call-by-need has to be used. Thus, call-by-need program calculi provide a good model of both: the correctness of the programs as well as the amount of required work for executing programs. Analyzing these calculi and providing tools for proving transformations to be correct and/or to be optimizations is cumbersome, since sharing complicates reasoning, but it is worth the effort.

There are several works on analyzing and proving correctness of program transformations (e.g. [11,7,22]). However, there seems to be little research on whether the (correct) program transformations are also optimizations – i.e. while preserving the meaning of the programs they also do not increase the runtime or the space behavior of the programs. Having such results is e.g. useful in automated tools for program transformation like Hermit [23].

A theory of optimizations or improvements in extended lambda calculi was developed in [10] for an untyped higher-order language with call-by-need evaluation, and for a call-by-value variant in [16]. In [10] the resource model counts the steps of an abstract machine for call-by-need evaluation which is

---

[1] This version is a revision of Version V1 from March 28, 2015

a variant of Sestoft's abstract machine [24]. The work of Moran and Sands [10] provides a foundation for program improvements which leads to several results exhibiting program transformations that are improvements and also provides techniques for showing program transformations being improvements. In [10] it is also remarked that the reductions used in any context (a form of partial evaluation) are improvements, but the efficiency gain has a limit: it is at most polynomial. A detailed analysis on this topic can be found in [4] for a call-by-value lambda calculus. Clearly, other program transformations (which are not calculus reductions) have a higher potential to improve efficiency. One such rule is common subexpression elimination which identifies equal subexpressions of the program and replaces them by references to a single copy of the subexpression. Common subexpression elimination is treated in [10], but not proved to be an improvement (but it is conjectured).

Recently, Hackett and Hutton [5] rediscovered the improvement theory of [10] to argue that optimizations are indeed improvements, with a particular focus on worker/wrapper transformations (see e.g. [2] for more examples). The work of [5] is based on the call-by-need abstract machine of [10] with a slightly modified measure for the improvement relation.

Goals and Results The goal of this paper is to develop an improvement theory for the LR-calculus [22], an extended higher-order lambda calculus which models the core language of Haskell. The LR-calculus extends the lambda calculus with `letrec`-expressions (to express recursive and shared-bindings), data constructors and `case`-expressions (which act as selectors), and Haskell's `seq`-operator. The operational semantics of the LR-calculus is a small-step call-by-need evaluation.

Differences to the work of Moran and Sands are (i) that the LR-calculus uses a small-step operational semantics expressed by rewriting rules and a strategy, (ii) that it does not restrict the syntax of arguments to be variables (i.e. in LR arbitrary expressions are allowed as arguments), and (iii) that it includes the `seq`-operator for strict evaluation of expressions which is indispensable to model the semantics of Haskell (see e.g. [7,17]).

We use previous results and techniques for the LR-calculus to establish new improvement laws, in particular we show that common subexpression elimination is an improvement. Here we can build upon a detailed analysis of reduction lengths (performed in [22] in the context of a strictness analysis); the method of using diagrams to compute and join overlappings between reductions and transformations which we developed and applied in several works [8,22,14,15] to show correctness of program transformations; and correctness of inlining (or common subexpression elimination) via infinite expressions (to unfold and remove the `letrec`-expressions) established in [21]. We prefer analyzing reductions in LR, due to the success of the diagram method in LR. For example, the letrec calculus mentioned in [1] is related, but does not model case, constructors nor seq, and its deref-rule complicates diagram proofs.

Since our improvement relation is different from [10] (it uses a different measure and operational semantics), we compare our measures with those in [10,5]. The result is that our improvement theory can be transferred to the abstract machine of [10] using our measure, and that our calculus and Moran and Sand's calculus together with their measures are equivalent w.r.t. resources.

Analyzing untyped calculi covers a large amount of program transformations, however, sometimes typing arguments are required for showing that interesting program transformations are improvements (see e.g. [5]). Due to cyclic bindings, using monomorphic typing and monomorphising a polymorphic calculus is insufficient. Hence we adapt ideas from system-F polymorphism [3,12], in particular from an intermediate language in a Haskell compiler [9,25], and extend our improvement theory for the calculus LRP – a polymorphically typed variant of LR with `let`-polymorphism.

Since the type erasure of reduction sequences in LRP exactly leads to the untyped reduction sequences in LR, the connection between the untyped and the typed calculi is quite close and indeed it is rather simple to transfer the results and the techniques from LR into LRP. We also show that a type-dependent transformation (called (caseId)) is an improvement.

Outline In Sect. 2 we recall the untyped calculus LR, and in Sect. 3 we introduce improvement for LR and prove a context lemma. In Sect. 4 we show that common subexpression elimination is an

improvement. In Sect. 5 we compare our length measure with the measures used by Moran and Sands' improvement theory. In Sect. 6 we consider the polymorphically typed variant of LR. We conclude in Sect. 7.

2   The Call-by-Need Lambda Calculus LR

We recall the calculus LR [22], which is an untyped call-by-need lambda calculus that extends the lambda calculus by recursive `letrec`, data constructors, case-expressions, and the `seq`-operator. We recall results from previous investigations: from [22] we reuse a counting theorem for reduction lengths and correctness of several program transformations. From [21] we reuse correctness of copying arbitrary expressions.

We employ the syntax of the calculus LR [22]. Let Var be a countable infinite set of variables. We assume that there is a fixed set of type constructors $\mathcal{K}$, where every type constructor $K \in \mathcal{K}$ has an arity $ar(K) \geq 0$, and there is a finite, non-empty set $D_K = \{c_{K,1}, \ldots, c_{K,|D_K|}\}$ of data constructors. Every data constructor has an arity $ar(c_{K,i}) \geq 0$.

Example 2.1. $\mathcal{K}$ may include the 0-ary type constructor `Bool` and the 1-ary type constructor `List` with $D_{\texttt{Bool}} = \{\texttt{True}, \texttt{False}\}$, $D_{\texttt{List}} = \{\texttt{Nil}, \texttt{Cons}\}$, $ar(\texttt{True}) = ar(\texttt{False}) = ar(\texttt{Nil}) = 0$, and $ar(\texttt{Cons}) = 2$.

The syntax of expressions $r, s, t \in Expr$ of LR is defined in Fig. 1. We write $FV(s)$ for the set of free variables of an expression s. Besides *variables* x, *abstractions* $\lambda x.s$, and *applications* $(s\ t)$ the syntax of LR comprises the following constructs: *Constructor applications* $(c_{K,i}\ s_1 \ldots s_{ar(c_{K,i})})$ always occur fully saturated. In our meta-notation we sometimes omit the index of the constructor or use vector notation and thus write e.g. $(c\ \vec{s})$ instead of $(c_{K,i}\ s_1 \ldots s_{ar(c_{K,i})})$. In a `letrec`-*expression* `letrec` $x_1 = s_1, \ldots, x_n = s_n$ `in` t all variables $x_1, \ldots, x_n$ must be pairwise distinct, the scope of x_i is all s_i and t. The bindings $x_1 = s_1, \ldots, x_n = s_n$ are called the `letrec`-*environment* and t is called the `in`-*expression*. Sometimes the environment or parts of it are abbreviated by Env (e.g. we write `letrec` Env `in` s or `letrec` Env_1, Env_2 `in` s). For a `letrec`-environment $Env = \{x_1 = t_1, \ldots, x_n = t_n\}$, we define $LV(Env) := \{x_1, \ldots, x_n\}$ and sometimes write $\{x_i = t_i\}_{i=1}^n$ as abbreviation for such an environment. For a chain of variable-to-variable bindings $x_j = x_{j-1}, x_{j+1} = x_j, \ldots, x_m = x_{m-1}$ we use the abbreviation $\{x_i = x_{i-1}\}_{i=j}^m$. A `seq`-*expression* $(\texttt{seq}\ s\ t)$ can be used for strict evaluation of expressions, since the expression s must be successfully evaluated before t is evaluated. For every $K \in \mathcal{K}$ there is a `case`-*expression* $(\texttt{case}_K\ s\ (c_{K,1}\ x_1 \ldots x_{ar(c_{K,1})}\ \texttt{->}\ t_1) \ldots (c_{K,|D_K|}\ x_1 \ldots x_{ar(c_{K,|D_K|})}\ \texttt{->}\ t_{|D_K|}))$ with exactly one `case`-*alternative* $((c_{K,i}\ x_1 \ldots x_{ar(c_{K,i})})\ \texttt{->}\ t_i)$ for every data constructor $c_{K,i} \in D_K$. The variables $x_1, \ldots, x_{ar(c_{K,i})}$ in the `case`-*pattern* $((c_{K,i}\ x_1 \ldots x_{ar(c_{K,i})})\ \texttt{->}\ t_i)$ must be pairwise distinct and the scope of the variables $x_1, \ldots, x_{ar(c_{K,i})}$ is the expression t_i. We sometimes use the meta-symbol *alts* to abbreviate the `case`-alternatives and thus e.g. write $(\texttt{case}_K\ s\ alts)$.

Example 2.2. The `case`-expression $\texttt{case}_{\texttt{Bool}}\ s_1\ (\texttt{True}\ \texttt{->}\ s_2)\ (\texttt{False}\ \texttt{->}\ s_3)$ corresponds to the conditional `if` s_1 `then` s_2 `else` s_3. The function *map* which maps a function to all elements of a list can be defined as `letrec` $map = \lambda f, xs.\texttt{case}_{\texttt{List}}\ xs\ (\texttt{Nil}\ \texttt{->}\ \texttt{Nil})\ (\texttt{Cons}\ y\ ys\ \texttt{->}\ \texttt{Cons}\ (f\ y)\ (map\ f\ ys))\ \texttt{in}\ map$.

We assume the convention that bound variables are only introduced once in expressions, and that these are different from free variables, which can always be achieved by renaming bound variables. We assume that reduction rules and transformations are followed by a renaming to fulfill this distinct variable convention. In our meta-notations we assume that variable names that appear twice are the same object, if not mentioned otherwise. E.g., the notation $(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C[x^{\mathsf{sub}}])$ assumes that C does not capture x in its hole.

Definition 2.3. *A* context C *is an expression with a hole (denoted by* $[\cdot]$*) at expression position. Surface contexts* S *are contexts where the hole is not in an abstraction, top contexts* T *are surface contexts where the hole is not in an alternative of a* `case`*, and* weak top contexts *are top contexts*

$$r, s, t \in \mathit{Expr} := x \mid \lambda x.s \mid (s\ t) \mid (c_{K,i}\ s_1\ \ldots\ s_{ar(c_{K,i})}) \mid (\mathtt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \mathtt{in}\ t) \mid (\mathtt{seq}\ s\ t)$$
$$\mid (\mathtt{case}_K\ s\ (c_{K,1}\ x_1 \ldots x_{ar(c_{K,1})}\ \mathtt{->}\ t_1) \ldots (c_{K,|D_K|}\ x_1 \ldots x_{ar(c_{K,|D_K|})}\ \mathtt{->}\ t_{|D_K|}))$$

Fig. 1: Expressions of the language LR where $x, x_i \in \mathit{Var}$ are term variables

$$
\begin{array}{ll}
(s\ t)^{\mathsf{sub} \vee \mathsf{top}} & \to (s^{\mathsf{sub}}\ t)^{\mathsf{vis}} \\
(\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ s)^{\mathsf{top}} & \to (\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ s^{\mathsf{sub}})^{\mathsf{vis}} \\
(\mathtt{letrec}\ x = s, \mathit{Env}\ \mathtt{in}\ C[x^{\mathsf{sub}}]) & \to (\mathtt{letrec}\ x = s^{\mathsf{sub}}, \mathit{Env}\ \mathtt{in}\ C[x^{\mathsf{vis}}]) \\
(\mathtt{letrec}\ x = s, y = C[x^{\mathsf{sub}}], \mathit{Env}\ \mathtt{in}\ t) & \to (\mathtt{letrec}\ x = s^{\mathsf{sub}}, y = C[x^{\mathsf{vis}}], \mathit{Env}\ \mathtt{in}\ t),\ \text{if}\ C \neq [\cdot] \\
(\mathtt{letrec}\ x = s, y = x^{\mathsf{sub}}, \mathit{Env}\ \mathtt{in}\ t) & \to (\mathtt{letrec}\ x = s^{\mathsf{sub}}, y = x^{\mathsf{nontarg}}, \mathit{Env}\ \mathtt{in}\ t) \\
(\mathtt{seq}\ s\ t)^{\mathsf{sub} \vee \mathsf{top}} & \to (\mathtt{seq}\ s^{\mathsf{sub}}\ t)^{\mathsf{vis}} \\
(\mathtt{case}\ s\ \mathtt{of}\ \mathit{alts})^{\mathsf{sub} \vee \mathsf{top}} & \to (\mathtt{case}\ s^{\mathsf{sub}}\ \mathtt{of}\ \mathit{alts})^{\mathsf{vis}} \\
\mathtt{letrec}\ x = s^{\mathsf{vis} \vee \mathsf{nontarg}}, y = C[x^{\mathsf{sub}}], \mathit{Env}\ \mathtt{in}\ t & \to \mathtt{Fail} \\
(\mathtt{letrec}\ x = C[x^{\mathsf{sub}}], \mathit{Env}\ \mathtt{in}\ s) & \to \mathtt{Fail}
\end{array}
$$

Fig. 2: Computing reduction positions using labels (see Definition 2.4), where $a \vee b$ means label a or label b. The algorithm does not overwrite non-displayed labels.

(lbeta) $C[((\lambda x.s)^{\mathsf{sub}}\ r)] \to C[(\mathtt{letrec}\ x = r\ \mathtt{in}\ s)]$

(cp-in) $(\mathtt{letrec}\ x_1 = (\lambda x.s)^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}\ \mathtt{in}\ C[x_m^{\mathsf{vis}}])$
  $\to (\mathtt{letrec}\ x_1 = (\lambda x.s), \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}\ \mathtt{in}\ C[\lambda x.s])$

(cp-e) $(\mathtt{letrec}\ x_1 = (\lambda x.s)^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}, y = C[x_m^{\mathsf{vis}}]\ \mathtt{in}\ r)$
  $\to (\mathtt{letrec}\ x_1 = (\lambda x.s), \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}, y = C[\lambda x.s]\ \mathtt{in}\ r)$

(llet-in) $(\mathtt{letrec}\ \mathit{Env}_1\ \mathtt{in}\ (\mathtt{letrec}\ \mathit{Env}_2\ \mathtt{in}\ r)^{\mathsf{sub}}) \to (\mathtt{letrec}\ \mathit{Env}_1, \mathit{Env}_2\ \mathtt{in}\ r)$

(llet-e) $(\mathtt{letrec}\ \mathit{Env}_1, x = (\mathtt{letrec}\ \mathit{Env}_2\ \mathtt{in}\ s_x)^{\mathsf{sub}}\ \mathtt{in}\ r) \to (\mathtt{letrec}\ \mathit{Env}_1, \mathit{Env}_2, x = s_x\ \mathtt{in}\ r)$

(lapp) $C[((\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ t)^{\mathsf{sub}}\ s)] \to C[(\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ (t\ s))]$

(lcase) $C[\mathtt{case}_K\ (\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ t)^{\mathsf{sub}}\ \mathit{alts}] \to C[(\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ \mathtt{case}_K\ t\ \mathit{alts})]$

(lseq) $C[\mathtt{seq}\ (\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ s)^{\mathsf{sub}}\ t] \to C[(\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ \mathtt{seq}\ s\ t)]$

(seq-c) $C[\mathtt{seq}\ v^{\mathsf{sub}}\ t] \to C[t]$, if v is a value

(seq-in) $\mathtt{letrec}\ x_1 = (c\ \overrightarrow{t})^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}\ \mathtt{in}\ C[\mathtt{seq}\ x_m^{\mathsf{vis}}\ t]$
  $\to \mathtt{letrec}\ x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}\ \mathtt{in}\ C[t]$

(seq-e) $\mathtt{letrec}\ x_1 = (c\ \overrightarrow{t})^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}, y = C[\mathtt{seq}\ x_m^{\mathsf{vis}}\ t]\ \mathtt{in}\ r$
  $\to \mathtt{letrec}\ x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}, y = C[t]\ \mathtt{in}\ r$

(case-c) $C[\mathtt{case}_K\ (c\ \overrightarrow{t})^{\mathsf{sub}} \ldots (c\ \overrightarrow{y}\ \mathtt{->}\ t) \ldots] \to C[\mathtt{letrec}\ \{y_i = t_i\}_{i=1}^{ar(c)}\ \mathtt{in}\ t]$, if $ar(c) \geq 1$

(case-c) $C[\mathtt{case}_K\ c^{\mathsf{sub}} \ldots (c\ \mathtt{->}\ t) \ldots] \to C[t]$, if $ar(c) = 0$

(case-in) $\mathtt{letrec}\ x_1 = (c\ \overrightarrow{t})^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}\ \mathtt{in}\ C[\mathtt{case}_K\ x_m^{\mathsf{vis}} \ldots (c\ \overrightarrow{z}\ \mathtt{->}\ t) \ldots]$
  $\to \mathtt{letrec}\ x_1 = (c\ \overrightarrow{y}), \{y_i = t_i\}_{i=1}^{ar(c)}, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}\ \mathtt{in}\ C[\mathtt{letrec}\ \{z_i = y_i\}_{i=1}^{ar(c)}\ \mathtt{in}\ t]$,
  where $ar(c) \geq 1$ and y_i are fresh variables

(case-in) $\mathtt{letrec}\ x_1 = c^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}\ \mathtt{in}\ C[\mathtt{case}_K\ x_m^{\mathsf{vis}} \ldots (c\ \mathtt{->}\ t) \ldots]$
  $\to \mathtt{letrec}\ x_1 = c, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}\ \mathtt{in}\ C[t]$, if $ar(c) = 0$

(case-e) $\mathtt{letrec}\ x_1 = (c\ \overrightarrow{t})^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, u = C[\mathtt{case}_K\ x_m^{\mathsf{vis}} \ldots (c\ \overrightarrow{z}\ \mathtt{->}\ r_1) \ldots], \mathit{Env}\ \mathtt{in}\ r_2$
  $\to \mathtt{letrec}\ x_1 = (c\ \overrightarrow{y}), \{y_i = t_i\}_{i=1}^{ar(c)}, \{x_i = x_{i-1}\}_{i=2}^m, u = C[\mathtt{letrec}\ \{z_i = y_i\}_{i=1}^{ar(c)}\ \mathtt{in}\ r_1], \mathit{Env}\ \mathtt{in}\ r_2$
  where $ar(c) \geq 1$ and y_i are fresh variables

(case-e) $\mathtt{letrec}\ x_1 = c^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, u = C[\mathtt{case}_K\ x_m^{\mathsf{vis}} \ldots (c\ \mathtt{->}\ r_1) \ldots], \mathit{Env}\ \mathtt{in}\ r_2$
  $\to \mathtt{letrec}\ x_1 = c, \{x_i = x_{i-1}\}_{i=2}^m \ldots, u = C[r_1], \mathit{Env}\ \mathtt{in}\ r_2$, if $ar(c) = 0$

Fig. 3: Reduction rules

(gc1)	$\texttt{letrec}\ \{x_i = s_i\}_{i=1}^n, Env\ \texttt{in}\ t \to \texttt{letrec}\ Env\ \texttt{in}\ t,\ \text{if for all}\ i : x_i \notin FV(t, Env)$
(gc2)	$\texttt{letrec}\ \{x_i = s_i\}_{i=1}^n\ \texttt{in}\ t \to t,\ \ \text{if for all}\ i : x_i \notin FV(t)$
(cpx-in)	$\texttt{letrec}\ x = y, Env\ \texttt{in}\ C[x] \to \texttt{letrec}\ x = y, Env\ \texttt{in}\ C[y],\ \text{if}\ x \neq y \in Var$
(cpx-e)	$\texttt{letrec}\ x = y, z = C[x], Env\ \texttt{in}\ t \to \texttt{letrec}\ x = y, z = C[y], Env\ \texttt{in}\ t,\ \text{if}\ x \neq y \in Var$
(cpax)	$\texttt{letrec}\ x = y, Env\ \texttt{in}\ s \to \texttt{letrec}\ x = y, Env[y/x]\ \texttt{in}\ s[y/x],\ \text{if}\ x \neq y \in Var,\ y \in FV(s, Env)$
(cpcx-in)	$\texttt{letrec}\ x = c\,\overrightarrow{t}, Env\ \texttt{in}\ C[x] \to \texttt{letrec}\ x = c\,\overrightarrow{y}, \{y_i = t_i\}_{i=1}^{ar(c)}, Env\ \texttt{in}\ C[c\,\overrightarrow{y}]$
(cpcx-e)	$\texttt{letrec}\ x = c\,\overrightarrow{t}, z = C[x], Env\ \texttt{in}\ t \to \texttt{letrec}\ x = c\,\overrightarrow{y}, \{y_i = t_i\}_{i=1}^{ar(c)}, z = C[c\,\overrightarrow{y}], Env\ \texttt{in}\ t$
(abs)	$\texttt{letrec}\ x = c\,\overrightarrow{t}, Env\ \texttt{in}\ s \to \texttt{letrec}\ x = c\,\overrightarrow{x}, \{x_i = t_i\}_{i=1}^{ar(c)}, Env\ \texttt{in}\ s,\ \text{where}\ ar(c) \geq 1$
(abse)	$(c\,\overrightarrow{t}) \to \texttt{letrec}\ \{x_i = t_i\}_{i=1}^{ar(c)}\ \texttt{in}\ c\,\overrightarrow{x},\ \text{where}\ ar(c) \geq 1$
(xch)	$\texttt{letrec}\ x = t, y = x, Env\ \texttt{in}\ r \to \texttt{letrec}\ y = t, x = y, Env\ \texttt{in}\ r$
(ucp1)	$\texttt{letrec}\ Env, x = t\ \texttt{in}\ S[x] \to \texttt{letrec}\ Env\ \texttt{in}\ S[t]$
(ucp2)	$\texttt{letrec}\ Env, x = t, y = S[x]\ \texttt{in}\ r \to \texttt{letrec}\ Env, y = S[t]\ \texttt{in}\ r$
(ucp3)	$\texttt{letrec}\ x = t\ \texttt{in}\ S[x] \to S[t]$
	where in the (ucp)-rules, x has at most one occurrence in $S[x]$ and no occurrence in Env, t, r; and S is a surface context

Fig. 4: Extra transformation rules

where the hole is not in a $\texttt{letrec}$*-expression. With* $C[s]$ *we denote the substitution of the hole in the context* C *by expression* s*. A* multicontext M *is an expression with zero or more (different) holes at expression positions.*

2.1 Normal Order Reduction

A *value* in LR is an abstraction $\lambda x.s$ or a constructor application $(c\,\overrightarrow{s})$. The reduction rules of LR are defined in Fig. 3, where the role of the labels $\mathsf{sub}, \mathsf{top}, \mathsf{vis}, \mathsf{nontarg}$ will be explained below in Definition 2.4. The rule (lbeta) is the sharing variant of classical β-reduction. The rules (cp-in) and (cp-e) copy abstractions. The rules (llet-in) and (llet-e) join two $\texttt{letrec}$-environments. The rules (lapp), (lcase), and (lseq) float-out a $\texttt{letrec}$ from the first argument of an application, a $\texttt{case}$-, or a $\texttt{seq}$-expression. The rules (seq-c), (seq-in), and (seq-e) evaluate a $\texttt{seq}$-expression, provided that the first argument is a value (or a variable that is bound (via indirections) to a constructor application). The rules (case-c), (case-in), and (case-e) evaluate a $\texttt{case}$-expression provided that the first argument is (or is a variable which is bound to) a constructor application of the right type.

The normal order reduction strategy of the calculus LR is a call-by-need strategy. It applies the reduction rules at specific positions which will be defined by an algorithm to find the position of a redex[2].

Definition 2.4 (Labeling Algorithm). *The labeling algorithm detects the position to which a reduction rule will be applied according to normal order. It uses the labels* $\mathsf{top},\mathsf{sub},\mathsf{vis},\mathsf{nontarg}$ *where* top *means reduction of the top term,* sub *means reduction of a subterm,* vis *marks already visited subexpressions, and* $\mathsf{nontarg}$ *marks visited variables that are not target of a (cp)-reduction. For a term* s *the labeling algorithm starts with* s^{top}*, where no other subexpression in* s *is labeled and proceeds by applying the rules given in Fig. 2 exhaustively.*

Note that the labeling algorithm does not descend into sub-labeled $\texttt{letrec}$-expressions. If the labeling algorithm does not fail, then a potential normal order redex is found, which can only be a superterm of the sub-marked subexpression. However, it is possible that there is no normal order reduction, if the evaluation is already finished, or no rule is applicable.

Definition 2.5 (Normal Order Reduction of LR**).** *Let* t *be an expression. Then a single* normal order reduction step $t \xrightarrow{no} t'$ *is defined by first applying the labeling algorithm to* t*, and if the labeling algorithm terminates successfully, then one of the rules in Fig. 3 has to be applied, if possible, where*

---

[2] This is equivalent to a a syntactic definition of reduction contexts (using a context free grammar) which is omitted in this paper, but can be found for the calculus LR in [22].

the labels sub, vis *must match the labels in the expression t (t may have more labels), and t' is the result after erasing all labels.*

Example 2.6. For (letrec $w = \lambda x.x, y = w, z = (y\ y)$ in z), the labeling algorithm sucessfully ends with (letrec $w = (\lambda x.x)^{\mathsf{sub}}, y = w^{\mathsf{nontarg}}, z = (y^{\mathsf{vis}}\ y)^{\mathsf{vis}}$ in $z^{\mathsf{vis}})^{\mathsf{vis}}$. Only the rule (cp-e) matches the subexpressions and the labels, i.e. (letrec $w = \lambda x.x, y = w, z = (y\ y)$ in z) $\xrightarrow{no}$ (letrec $w = \lambda x.x, y = w, z = ((\lambda x'.x')\ y)$ in z) where the α-renaming is performed implicitly to fulfill the distinct variable convention. Note that the label nontarg at w prevents the normal order reduction to copy the sub-labeled abstraction into this position.

For letrec $x = (y\ y), y = x$ in y the labeling fails and thus the expression has no normal order reduction. For (Cons True Nil) $(\lambda x.x)$, the labeling terminates with ((Cons True Nil)$^{\mathsf{sub}}$ $(\lambda x.x))^{\mathsf{vis}}$ and also for letrec $x =$ True in x, the labeling ends successful with (letrec $x =$ True$^{\mathsf{sub}}$ in $x^{\mathsf{vis}})^{\mathsf{vis}}$. However, for both expressions no normal order reduction is applicable (since the labels do not match the reduction rules). The former expression is irreducible, since it has a "dynamic type error". The latter expression is irreducible since it is successfully evaluated, i.e. it is a weak head normal form (see Definition 2.8).

By a case analysis one can verify that normal order reduction is unique, i.e. for an expression t either no normal order reduction is possible, or there is a unique expression t' (up-to α-equivalence) s.t. $t \xrightarrow{no} t'$.

We sometimes attach more information to the reduction arrow, e.g. $\xrightarrow{no,lbeta}$ denotes a normal order reduction using rule (lbeta). For a binary relation $\rightarrow$, we write $\xrightarrow{+}$ for the transitive closure, and $\xrightarrow{*}$ for the reflexive-transitive closure of $\rightarrow$. E.g., $\xrightarrow{no,*}$ denotes the reflexive-transitive closure of $\xrightarrow{no}$. We write $\xrightarrow{n}$ for exactly n $\rightarrow$-steps and we write $\xrightarrow{n \vee m}$ for either n or m steps. The notation $\xrightarrow{a \vee b}$ is also used for a and b being rule names, meaning the union of the rules a and b. For instance, $\xrightarrow{no,lbeta \vee no,lapp,0 \vee 1}$ means one or none normal order reduction step using rule (lbeta) or rule (lapp).

We define reduction contexts and weak reduction contexts:

Definition 2.7. *A reduction context R is any context, such that its hole will be labeled with sub or top by the labeling algorithm in Fig. 2. A weak reduction context, R^-, is a reduction context, where the hole is not within a letrec-expression.*

Clearly, every reduction context is also a top context, and thus also a surface context.

Definition 2.8. *An expression s is a weak head normal form (WHNF), if s is a value, or s is of the form letrec Env in v, where v is a value, or s is of the form letrec $x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^m$, Env in x_m.*

By inspecting the labeling algorithm and the reduction rules one can verify that every WHNF is irreducible w.r.t. normal order reduction. As usual for lazy functional programming languages, we consider WHNFs as successfully evaluated programs and define the notion or convergence accordingly:

Definition 2.9. *An expression s converges, denoted as $s\downarrow$, iff there exists a WHNF t s.t. $s \xrightarrow{no,*} t$. This may also be denoted as $s \downarrow t$. We write $s\uparrow$ iff $s\downarrow$ does not hold.*

2.2 Program Transformations

A *program transformation P* is a binary relation on expressions. We write $s \xrightarrow{P} t$, if $(s,t) \in P$. For a set of contexts X and a transformation P, the transformation (X, P) is the closure of P w.r.t. the contexts in X, i.e. $C[s] \xrightarrow{X,P} C[t]$ iff $C \in X$ and $s \xrightarrow{P} t$.

Ignoring the lables, the reduction rules in Fig. 3 are also program transformations. In Fig. 4 additional program transformations are defined. The transformation (gc) performs garbage collection by removing unused letrec-environments, and (cpx), (cpax) copy variables, and can be used to shorten chains of indirections. The transformation (cpcx) copies a constructor application into a referenced

position, where the arguments are shared by new `letrec`-bindings. Similarly, (abs) and (abse) perform this sharing without copying the constructor application. The transformation (ucp) means "unique copying" and it inlines a shared expression which is referenced only once.

Definition 2.10. *We define unions for the rules in Fig. 3: (case) is the union of (case-c), (case-in), (case-e); (seq) is the union of (seq-c), (seq-in), (seq-e); (cp) is the union of (cp-in), (cp-e); (llet) is the union of (llet-in), (llet-e); and (lll) is the union of (llet), (lapp), (lcase), and (lseq).*

We use the following unions for the transformations in Fig. 4: (gc) is the union of (gc1) and (gc2); (cpx) is the union of (cpx-in) and (cpx-e); (cpcx) is the union of (cpcx-in) and (cpcx-e); and (ucp) is the union of (ucp1), (ucp2), and (ucp3).

We use the unions of the rules also for normal order reduction and e.g. write $\xrightarrow{no,llet}$ for $\xrightarrow{no,llet-in \lor no,llet-e}$.

2.3 Contextual Equivalence

As program equivalence we use contextual equivalence which equates two expressions if exchanging one program by the other program cannot be observed in any surrounding program contexts. Due to the quantification over all contexts, it is sufficient to observe convergence.

Definition 2.11. *Let s, t be two LR-expressions. We define* contextual equivalence $\sim_c$ *w.r.t. the operational semantics of LR: Let $s \sim_c t$, iff for all contexts $C[\cdot]$: $C[s]\downarrow \iff C[t]\downarrow$.*

Contextual equivalence is a congruence, i.e. it is an equivalence relation which is compatible with contexts. Usually, proving two expressions to be contextually equivalent is hard, since all contexts have to be taken into account. In contrast, disproving an equivalence is often easy, since a single counter-example is sufficient. E.g., the constants `True` and `False` are not contextually equivalent, since the context $\mathtt{case_{Bool}}\ [\cdot]\ (\mathtt{True \text{->} True})\ (\mathtt{False \text{->}} \Omega)$ distinguishes them (where Ω is a divergent expression, for instance $\Omega = (\lambda x.x\ x)\ (\lambda y.y\ y)$).

Definition 2.12. *A program transformation P is* correct*, if it preserves contextual equivalence, i.e. $P \subseteq \sim_c$.*

In [22] we proved that all introduced transformations are correct:

Proposition 2.13 ([22]). *The program transformations (lbeta), (case), (seq), (cp), (lll), (gc), (cpx), (cpax), (cpcx), (abs), (abse), (xch), and (ucp) are correct.*

3 Improvement in the LR-Calculus

While contextual equivalence is a correctness criterion for program transformations, it has no requirements on the transformation being an *optimization* w.r.t. time (or space) complexity of a program. This is where the improvement relation comes into play and restricts contextual equivalence of s and t by the additional requirement that s may be replaced by t (within a program) if the number of computation steps for successfully evaluating the whole program is not increased. We define measures for estimating the time consumption: on the one hand we count all reduction steps, on the other hand we count essential reduction steps where we use a generic approach and leave some freedom in the choice of which steps are counted:

Definition 3.1. *In the following let $\mathfrak{A} := \{(lbeta), (case), (seq)\}$ and $\emptyset \neq A \subseteq \mathfrak{A}$. Let t be a closed expression with $t\downarrow t_0$. Then $\mathtt{rln}_A(t)$ is the number of a-reductions with $a \in A$ in the normal order reduction $t\downarrow t_0$. With $\mathtt{rlnall}(t)$ we denote the number of all reductions in $t\downarrow t_0$*

Similarly, we apply the measures to reduction sequences $t \xrightarrow{*} t'$. The measures are defined as ∞, if $t\uparrow$, since we are only interested in values on converging expressions.

The restriction of counting mainly reductions in $\mathfrak{A}$ can be justified as follows: The (cp)-reductions are not counted, however, since every (no,cp)- is followed by an (no,lbeta)- or an (no,seq)-reduction, or it is the last reduction, the number of (cp)-reductions of an expression t is at most $2 \cdot \mathtt{rln}_{\mathfrak{A}}(t) + 1$.

Also (lll)-reductions are not counted, since these can be more efficiently implemented on abstract machines, often more efficient than in the calculus model, by floating environments to the top in one step instead of doing it step-by-step. A further deviation from real run-time is the size of the abstractions, which are duplicated in a (cp)-reduction. Also the search for a redex (modeled by our labeling algorithm) is not counted by our measures (which is different from [10]). If the computation is long compared to the size of the expression, then the sizes of abstractions can be considered as constant. In particular, in call-by-need computation, the size of abstractions cannot be increased. This alleviates the error made by not counting the size (see also Theorem 5.21).

Using measures with $A \subset \mathfrak{A}$ has two motivations: On the one hand we get more exact information, and on the other hand, we can show an improvement relation also for variations of the measure like $a_1 \cdot \mathtt{rln}_{\{lbeta\}} + a_2 \cdot \mathtt{rln}_{\{case\}} + a_3 \cdot \mathtt{rln}_{\{seq\}}$ with $a_i \geq 0$.

We analyse the relative number of (lbeta)-reductions w.r.t. other reductions in $\mathfrak{A}$. First we define the size of an expression:

Definition 3.2. *For an* LR*-expression s, its size* $\mathtt{size}(s)$ *is defined as*

$$
\begin{aligned}
\mathtt{size}(x) &:= 1, \ \textit{if } x \in \textit{Var} \\
\mathtt{size}(\lambda x.s) &:= 1 + \mathtt{size}(s) \\
\mathtt{size}(s\ t) &:= 1 + \mathtt{size}(s) + \mathtt{size}(t) \\
\mathtt{size}(\mathtt{seq}\ s\ t) &:= 1 + \mathtt{size}(s) + \mathtt{size}(t) \\
\mathtt{size}(\mathtt{letrec}\ x_1 = s_1 \ldots, x_n = s_n\ \mathtt{in}\ s_0) &:= 1 + \sum_{i=0}^{n} \mathtt{size}(s_i) \\
\mathtt{size}(\mathtt{case}_K\ t\ (pat_1\ \mathtt{->}\ s_1) \ldots (pat_{ar(K)}\ \mathtt{->}\ s_{ar(K)})) &:= 1 + \mathtt{size}(t) + \sum_{i=0}^{ar(K)} (\mathtt{size}(pat_i) + \mathtt{size}(s_i)) \\
\mathtt{size}(c_{K,i}\ s_1\ \ldots\ s_{ar(c_{K,i})}) &:= 1 + \sum_{i=1}^{ar(c_{K,i})} \mathtt{size}(s_i)
\end{aligned}
$$

Proposition 3.3. *Let s be a closed expression. Then* $\mathtt{rln}_{\mathfrak{A}}(s) \leq (\mathtt{size}(s) + 1) \cdot (\mathtt{rln}_{\{lbeta\}}(s) + 1)$.

Proof. It suffices to assume that $s{\downarrow}$, since otherwise the claim obviously holds. So suppose $s \xrightarrow{no,*} t$ where t is a WHNF. For every (*cp*)-reduction in this sequence exactly one of the following properties holds: i) The (*cp*)-reduction is the last reduction in the sequence. ii) The (*cp*)-reduction is followed by a (*seq*)-reduction (which removes the copied abstraction). iii) The (*cp*)-reduction is followed by an (*lbeta*)-reduction (which uses the copied abstraction in function position). Let us conflate these reductions, i.e. we write (*cpseq*) for a (*cp*)-reduction followed by (*seq*), (*cplbeta*) for a (*cp*)-reduction followed by (*lbeta*), and (*cpWHNF*) for a (*cp*)-reduction as a last reduction in a normal order reduction to a WHNF. Let $\mathtt{size}_{LR}$ be the size of an expression, where we do not count the $x = y$-bindings, i.e. $\mathtt{size}_{LR}(s) = \mathtt{size}(s) - \textit{indirections}(s)$, where $\textit{indirections}(s)$ is the number of $\mathtt{letrec}$-bindings $x = y$ in s where y is a variable.

Then all the reduction rules of LR with the exception of (*cplbeta*) and (*cpWHNF*) do not increase the $\mathtt{size}_{LR}$, and the reduction rules (*case*), (*seq*), and (*cpseq*) strictly decrease $\mathtt{size}_{LR}$.

Thus to estimate the number of (*case*)-, (*seq*)- and (*cpseq*)-reductions it is sufficient to determine the initial size of the expression and the size increase of (*cplbeta*)-reductions.

In the reduction sequence $s \xrightarrow{no,*} t$, a single (*cplbeta*)-reduction increases the $\mathtt{size}_{LR}$ at most by $\mathtt{size}(s)$. Thus the overall size which can be used by (*seq*), (*cpseq*), and (*case*) is (1+number of (*cplbeta*)) $\cdot$ ($\mathtt{size}(s)$) (the size increase of (*cpWHNF*) cannot be used by (*seq*), (*cpseq*), or (*case*)).

The number of (*cplbeta*)-reductions in $s \xrightarrow{no,*} t$ is at most $\mathtt{rln}_{\{lbeta\}}(s)$. Hence $\mathtt{rln}_{\mathfrak{A}}(s)$ is at most $\mathtt{rln}_{\{lbeta\}}(s)$ plus $\mathtt{size}(s) \cdot (\mathtt{rln}_{\{lbeta\}}(s) + 1)$, which is at most $(\mathtt{size}(s) + 1) \cdot (\mathtt{rln}_{\{lbeta\}}(s) + 1)$. □

Remark 3.4. Note that there are normal-order reduction sequences with an arbitrary number of (*lbeta*)-reductions, but without any (*case*)- or (*seq*)-reductions, hence Proposition 3.3 is not valid for {*case*}, {*seq*} or {*case, seq*} instead of {*lbeta*}.

We present results from [20] w.r.t. reduction lengths of the transformations in Figs. 3 and 4, which are slightly more general, and can be derived from these papers.

Theorem 3.5 ([22]). *Let t be a closed* LR*-expression s.t. $t \downarrow t_0$.*

1. *If $t \xrightarrow{C,a} t'$ and $a \in \{case, seq, lbeta\}$, then for all A:* $\mathtt{rln}_A(t) \geq \mathtt{rln}_A(t')$ *and* $\mathtt{rlnall}(t) \geq \mathtt{rlnall}(t')$.
2. *If $t \xrightarrow{C,cp} t'$, then for all A:* $\mathtt{rln}_A(t) = \mathtt{rln}_A(t')$.
3. *If $t \xrightarrow{S,a} t'$ and $a \in \{case, seq, lbeta\}$, then for all A:* $\mathtt{rln}_A(t) \geq \mathtt{rln}_A(t') \geq \mathtt{rln}_A(t) - 1$ *if $a \in A$, and otherwise, if $a \notin A$, then* $\mathtt{rln}_A(t) = \mathtt{rln}_A(t')$.
4. *If $t \xrightarrow{C,a} t'$ and $a \in \{lll, gc\}$, then for all A:* $\mathtt{rln}_A(t) = \mathtt{rln}_A(t')$ *and* $\mathtt{rlnall}(t) \geq \mathtt{rlnall}(t')$. *For $a = gc1$ the equation* $\mathtt{rlnall}(t) = \mathtt{rlnall}(t')$ *holds.*
5. *If $t \xrightarrow{C,a} t'$ and $a \in \{cpx, cpax, xch, cpcx, abs\}$, then for all A:* $\mathtt{rln}_A(t) = \mathtt{rln}_A(t')$ *and* $\mathtt{rlnall}(t) = \mathtt{rlnall}(t')$.
6. *If $t \xrightarrow{C,ucp} t'$, then for all A:* $\mathtt{rln}_A(t) = \mathtt{rln}_A(t')$ *and* $\mathtt{rlnall}(t) \geq \mathtt{rlnall}(t')$.

3.1 The Improvement Relation

The improvement relation identifies contextual equivalent expressions and requires that the reduction length $\mathtt{rln}_A(\cdot)$ is not increased:

Definition 3.6 (Improvement Relation). *For $s, t \in Expr$, $A \subseteq \mathfrak{A}$, let $s \preceq_A t$ (t is A-improved by s), iff $s \sim_c t$ and for all contexts $C[\cdot]$ s.t. $C[s], C[t]$ are closed:* $\mathtt{rln}_A(C[s]) \leq \mathtt{rln}_A(C[t])$. *We write $t \succeq_A s$ if $s \preceq_A t$ holds. If $s \preceq_A t$ and $s \succeq_A t$, we write $s \approx_A t$.*
A program transformation P is an A-improvement iff $P \subseteq \succeq_A$.

Let $\eta \in \{\leq, =, \geq\}$ be a relation on non-negative integers. For a class of contexts X[3], $A \subseteq \mathfrak{A}$, let $s \bowtie_{\eta, X, A} t$ iff $s \sim_c t$ and for all X-contexts X, s.t. $X[s], X[t]$ are closed: $\mathtt{rln}_A(X[s]) \; \eta \; \mathtt{rln}_A(X[t])$. In particular, instantiating $\bowtie_{\eta, X, A}$ with all contexts gives the already introduced improvement relations, i.e. $\bowtie_{\leq, C, A} = \preceq_A$, $\bowtie_{\geq, C, A} = \succeq_A$, and $\bowtie_{=, C, A} = \approx_A$.

The context lemma for improvement shows that it suffices to take reduction contexts into account for proving improvement. Its proof uses a generalized claim with multicontexts and it is similar to the ones for context lemmas for contextual equivalence in call-by-need lambda calculi (see [22,18]).

Lemma 3.7 (Context Lemma for Improvement). *Let s, t be expressions, $\eta \in \{\leq, =, \geq\}$, and $A \subseteq \mathfrak{A}$. Then $s \bowtie_{\eta, R, A} t$ (or $s \bowtie_{\eta, T, A} t$, or $s \bowtie_{\eta, S, A} t$) iff $s \bowtie_{\eta, C, A} t$.*

Proof. It suffices to consider the case for reduction contexts, since reduction contexts are also T- or S-contexts and thus if the context lemma holds for $\bowtie_{\eta, R, A}$ then it also holds for $\bowtie_{\eta, T, A}$ or $\bowtie_{\eta, S, A}$. One direction of the claim is trivial. For the other direction we prove a more general claim:

> *For all $n \geq 0$ and for all $i = 1, \ldots, n$ let s_i, t_i be expressions with $s_i \sim_c t_i$ and $s_i \bowtie_{\eta, R, A} t_i$. Then for all multicontexts M with n holes such that $M[s_1, \ldots, s_n]$ and $M[t_1, \ldots, t_n]$ are closed:* $\mathtt{rln}_A(M[s_1, \ldots, s_n]) \; \eta \; \mathtt{rln}_A(M[t_1, \ldots, t_n])$.

The proof is by induction on the pair (k, k') where k is the number of normal order reductions of $M[s_1, \ldots, s_n]$ to a WHNF, and k' is the number of holes of M. If M (without holes) is a WHNF,

---

[3] We will instantiate X with all contexts C; all reduction contexts R; all surface contexts S; or all top-contexts T

then the claim holds. If $M[s_1, \ldots, s_n]$ is a WHNF, and no hole is in a reduction context, then also $M[t_1, \ldots, t_n]$ is a WHNF and $\mathtt{rln}_A(M[s_1, \ldots, s_n]) = 0 = \mathtt{rln}_A(M[t_1, \ldots, t_n])$.

If in $M[s_1, \ldots, s_n]$ one s_j is in a reduction context, then one hole, say i, of M is in a reduction context and $M[t_1, \ldots, t_{i-1}, \cdot, t_{i+1}, \ldots, t_n]$ is a reduction context. Applying the induction hypothesis to the multicontext $M[\ldots, \cdot, s_i, \cdot, \ldots]$ (with $n-1$ holes) shows $\mathtt{rln}_A(M[s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n])$ ξ $\mathtt{rln}_A(M[t_1, \ldots, t_{i-1}, s_i, t_{i+1}, \ldots, t_n])$, and the assumption shows $\mathtt{rln}_A(M[t_1, \ldots, t_{i-1}, s_i, t_{i+1}, \ldots, t_n])$ ξ $\mathtt{rln}_A(M[t_1, \ldots, t_{i-1}, t_i, t_{i+1}, \ldots, t_n])$, and hence $\mathtt{rln}_A(M[s_1, \ldots, s_n])$ ξ $\mathtt{rln}_A(M[t_1, \ldots, t_n])$.

If in $M[s_1, \ldots, s_n]$ there is no s_j in a reduction context, then the first normal order reduction is of the form $M[s_1, \ldots, s_n] \xrightarrow{no,a} M'[s'_1, \ldots, s'_{n'}]$, and it may copy or shift some of the s_i where $s'_j = \rho(s_i)$ for some variable permutation ρ. However, the reduction type is the same for the first step of $M[s_1, \ldots, s_n]$ and $M[t_1, \ldots, t_n]$, i.e. $M[t_1, \ldots, t_n] \xrightarrow{no,a} M'[t'_1, \ldots, t'_{n'}]$ with $(s'_j, t'_j) = (\rho(s_i), \rho(t_i))$. We take for granted that the renaming can be carried through. The $\mathtt{rln}_A(.)$-contribution of both $\xrightarrow{no,a}$-reduction steps is either $m = 0$ or $m = 1$, depending on a. We can apply the induction hypothesis to $M'[s'_1, \ldots, s'_{n'}]$ and $M'[t'_1, \ldots, t'_{n'}]$, and thus $\mathtt{rln}_A(M[s_1, \ldots, s_n]) = (m + \mathtt{rln}_A(M'[s'_1, \ldots, s'_{n'}]))$ ξ $(m + \mathtt{rln}_A(M'[t'_1, \ldots, t'_{n'}])) = \mathtt{rln}_A(M[t_1, \ldots, t_n])$.

3.2   Proof Technique

We explain our proof technique for proving that a program transformation is an improvement. The method is similar to the diagram-based technique used to show correctness of program transformations (see e.g. [22]). Let $\xrightarrow{P}$ be a correct program transformation. Due to the context lemma for improvement, it suffices to prove $\xrightarrow{P} \subseteq \bowtie_{\geq, X, A}$ (where X is instantiated with an appropriate set of contexts, i.e. reduction contexts R, top-contexts T, surface contexts S) to conclude $\xrightarrow{P} \subseteq \succeq_A$. To ease notation we will show this claim by proving that for all s, t with $s \xrightarrow{X,P} t$ the inequation $\mathtt{rln}_A(s) \geq \mathtt{rln}_A(t)$ holds, i.e. we use the closure of $\xrightarrow{P}$ w.r.t. X-contexts. If $\mathtt{rln}_A(s) = \infty$, then $s \sim_c t$ implies that $\mathtt{rln}_A(t) = \infty$ must also hold. Thus, the remaining case is that $s\downarrow$. Given the normal order reduction sequence from s to a WHNF, we construct a converging normal order reduction sequence for t and use this construction to prove that $\mathtt{rln}_A(s) \geq \mathtt{rln}_A(t)$ holds. For the construction of the reduction sequence, we use so-called sets of *forking diagrams* for the transformation $\xrightarrow{X,P}$ and often also additional forking diagrams for other transformations. These diagrams are then used in an inductive proof to construct the normal order reduction sequence for t, where often specific induction measures are required (if the diagrams are sufficiently complex).

A forking diagram describes how an overlapping (a fork) of the form $s' \xleftarrow{no,+}$ $s \xrightarrow{X,P} t$ can be closed by a sequence of reductions and transformations of the form $s' \xrightarrow{P',*} t' \xleftarrow{no,+} t$, where $\xrightarrow{P'}$ are program transformations, e.g. $\xrightarrow{X,P}$, but maybe also other transformations. A forking diagram abstracts from the concrete terms and thus it is written as depicted in Fig. 5, where often the reductions and transformations are more concrete (i.e. which rule of a normal order reduction is used). Sometimes also metaterms are used to represent the expressions. The solid arrows are the given reductions (the fork) and the dashed arrows are the existentially quantified reductions. A set of forking diagrams is *complete* for transformation $\xrightarrow{X,P}$, if the set contains an applicable diagram for every fork $s' \xleftarrow{no,+} s \xrightarrow{X,P} t$, where a diagram is applicable if the reductions and transformations concretely exist. A complete set of forking diagrams is obtained by a case analysis which considers all overlappings of a transformation and a normal order reduction. This usually is a tedious task, and thus we do not include the case analyses in the paper, but they can be found in [22].

$$\begin{array}{ccc} \cdot & \xrightarrow{X,P} & \cdot \\ {\scriptstyle no,+}\Big\downarrow & & \Big\downarrow{\scriptstyle no,+} \\ \cdot & \dashrightarrow[P'] & \cdot \end{array}$$

Fig. 5: Forking diagram

If $s \xrightarrow{X,P} t$ and $s\downarrow$, i.e. $s = s_0 \xrightarrow{no} \cdots \xrightarrow{no} s_n$, where s_n is a WHNF, then a complete set of forking diagrams can be used to construct a reduction sequence which witnesses $t\downarrow$ and satisfies
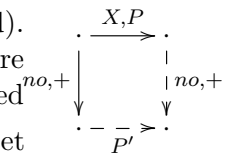
$a \in \{(lbeta), (case), (seq), (lll)\}$ $a \in \{(lbeta), (case), (seq)\}$ $a \in \{(lbeta), (seq)\}$

(1)   (2)   (3)
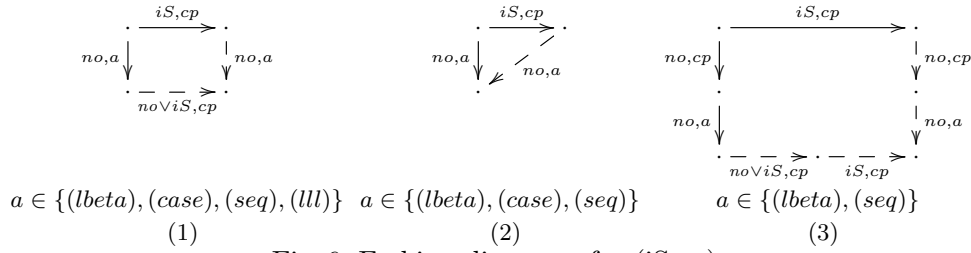
Fig. 6: Forking diagrams for (iS,cp)

$\mathtt{rln}_A(s) \geq \mathtt{rln}_A(t)$: A single forking diagram is applied for every step $s_i \to s_{i+1}$ and one has to show hat the construction terminates (usually using an inductive argument). If other transformations than $\xrightarrow{X,P}$ occur in the forking diagrams, then usually also these diagrams are required for the construction of the reduction sequences.

Note that the same technique can also be used to show $\xrightarrow{P} \subseteq \preceq$ or also $\xrightarrow{P} \subseteq \approx$.

3.3   Properties of (cp) and Other Transformations

We apply this proof technique to prove properties of the (cp)-reduction using the diagrams in [22]. The set of forking diagrams in Fig. 6 covers all cases of overlappings between a normal order reduction and an $\xrightarrow{iS,cp}$-transformation where iS means the closure of (cp) in surface contexts, but excluding (no,cp) reductions. The diagrams are obtained from Lemmas B.8 and B.9 of the appendix of [22]. We use these diagrams to prove the following theorem.

Theorem 3.8. *Let t be closed s.t. $t \downarrow t_0$. If $t \xrightarrow{C, \; cp} t'$ then $\mathtt{rln}_A(t) = \mathtt{rln}_A(t')$.*

Proof. We use the context lemma 3.7 for improvement for the relation $\approx_A$ and for surface contexts, i.e., we show $(cp) \subseteq \bowtie_{=,S,A}$ to derive $\bowtie_{=,C,A} = \approx_A$. Let s be closed and $s \xrightarrow{S,cp} s'$. We already know that $s \sim_c s'$, hence we can assume that $s \downarrow$, which implies $s' \downarrow$. We can also assume that the reduction is not normal order since in this the claim is trivial.

We prove $\mathtt{rln}_A(s) = \mathtt{rln}_A(s')$ by induction on $\mathtt{rln}_A(s)$ and then on the length of a normal order reduction. If the length is 0, then s is a WHNF, and hence s' is a WHNF.

If $s \xrightarrow{no,a} s_1$ for $a \in A$, then $\mathtt{rln}_A(s_1) = \mathtt{rln}_A(s) - 1$. Either diagram (29) or (30) holds. In the former case we can apply the induction hypothesis, and in the latter case the claim obviously holds. If $s \xrightarrow{no,cp} s_1$, then there are two cases: s_1 is a WHNF. In this case it is easy to see that there is a WHNF s_2 with $s' \xrightarrow{no,cp} s_2$, and the claim holds. The other case is that diagram (31) is applicable. Then $s_1 \xrightarrow{no,a} s_2$ and $\mathtt{rln}_A(s_2) = \mathtt{rln}_A(s) - 1$. Hence we can apply the induction hypothesis twice, and obtain the claim. If $s \xrightarrow{no,lll} s_1$, then diagram (29) applies, and by he induction hypothesis, we have $s' \xrightarrow{no,lll} s_1$, and since $\mathtt{rln}_A(s') = \mathtt{rln}_A(s_1)$, we obtain $\mathtt{rln}_A(s) = \mathtt{rln}_A(s')$.

Due to the exact analyses in [22] on the influence of the reduction rules (Fig. 3) and the additional transformations (Fig. 4) concerning the reduction lengths as stated in Theorems 3.5 and 3.8, Proposition 2.13 and Lemma 3.7 imply the following theorem:

Theorem 3.9. *The transformations (case), (seq), (lbeta), (cp), (lll), (gc), (cpx), (cpax), (xch), (abs), and (ucp) are A-improvements. Moreover, for $b \in \{(cp), (lll), (gc), (cpx), (cpax), (xch), (abs), (ucp)\}$ the inclusion $b \subseteq \preceq_A$ and thus the inclusion $b \subseteq \approx_A$ holds.*

4   Common Subexpression Elimination

In this section, we show the improvement property for common subexpression elimination (cse):

(cse) $M[s, \ldots, s] \to \mathtt{letrec}\ x = s\ \mathtt{in}\ M[x, \ldots, x]$
where x is a fresh variable and the multicontext M does not capture a variable in s

Although it seems to be obvious that (cse) is an improvement, its proof is not trivial, due to several reasons. The calculus LR is call-by-need, which means that (parts of) computations can be shared. To the best of our knowledge, we are only aware of a correctness proof of (*cse*) in a call-by-need calculus via a translation into a call-by-name calculus on infinite trees [21], which cannot be used to analyze resource usage under call-by-need. The improvement-property of (cse) was mentioned as a conjecture in [10].

We describe the structure of our proof that $(cse) \subseteq \succeq_A$ holds for all A. As a first step, we consider the general-copy rule (gcp) which allows to inline an arbitrary expression[4]:

$$(gcp) \quad \texttt{letrec } x = s \texttt{ in } C[x] \ \rightarrow \ \texttt{letrec } x = s \texttt{ in } C[s].$$

We will show that the inverse of (*gcp*) is an improvement (and thus $(gcp) \subseteq \preceq_A$). This result together with the result that garbage collection is invariant w.r.t. $\approx_A$ (Theorem 3.9) imply that (cse) is an improvement, since $\texttt{letrec } x = s \texttt{ in } M[x, \ldots, x] \xrightarrow{gcp,*} \texttt{letrec } x = s \texttt{ in } M[s, \ldots, s] \xrightarrow{gc} M[s, \ldots, s]$ and thus $(cse) \subseteq (\succeq \circ \approx)$ which implies $(cse) \subseteq \succeq_A$, since $\succeq \subseteq \approx_A$ and since $\succeq_A$ is a precongruence.

It remains to prove that the inverse of (*gcp*) is an improvement. Note that our previous approaches for proving *correctness* of (*gcp*) using the diagram-based method failed, since we were unable to show termination of the induction in corresponding proofs. However, correctness of (*gcp*) was recently shown in [21] by another technique. Using the correctness result we are able to prove that the transformation (*pcg*) – a slight extension of the inverse of (*gcp*) – is an improvement by using a complete set of forking diagrams for (T, pcg) and then applying the context lemma for improvement.

However, the forking diagrams for (T, pcg) (shown in Fig. 8) contain further transformations. They include some additional transformations from Fig. 4 which are already analyzed w.r.t. their behavior on reduction lengths, but also a new transformation, called (*pcgE*), which unions several variants of (*pcg*) in environments. As already mentioned, this forces us to develop a complete set of forking diagrams also for $(T, pcgE)$. Luckily, these diagrams (shown in Fig. 7) do not require new transformations and thus the complete sets of forking diagrams for (T, pcg) and $(T, pcgE)$ and the results on the additional transformations are sufficient to establish that (*pcg*) is an improvement.

Definition 4.1. *The transformation* (*pcgE*) *is the union of the following rules, where* $Env'\alpha = Env$ *and* α *only renames variables of* $LV(Env')$.

$(\text{pcgE}_1\text{-in})$ $\texttt{letrec } Env, Env_2 \texttt{ in } C[\texttt{letrec } Env', Env_3 \texttt{ in } r]$
  $\rightarrow \texttt{letrec } Env, Env_2 \texttt{ in } C[\texttt{letrec } Env_3\alpha \texttt{ in } r\alpha]$
$(\text{pcgE}_2\text{-in})$ $\texttt{letrec } Env, Env_2 \texttt{ in } C[\texttt{letrec } Env' \texttt{ in } r] \rightarrow \texttt{letrec } Env, Env_2 \texttt{ in } C[r\alpha]$
$(\text{pcgE}_1\text{-e})$ $\texttt{letrec } Env, Env_2, x = C[\texttt{letrec } Env', Env_3 \texttt{ in } r] \texttt{ in } s$
  $\rightarrow \texttt{letrec } Env, Env_2, x = C[\texttt{letrec } Env_3\alpha \texttt{ in } r\alpha] \texttt{ in } s$
$(\text{pcgE}_2\text{-e})$ $\texttt{letrec } Env, Env_2, x = C[\texttt{letrec } Env' \texttt{ in } r] \rightarrow \texttt{letrec } Env, Env_2, x = C[r\alpha] \texttt{ in } s$
(pcgE_3) $\texttt{letrec } Env, Env', Env_3 \texttt{ in } r \rightarrow \texttt{letrec } Env, Env_3\alpha \texttt{ in } r\alpha$

The transformation (*pcg*) *is the union of the rules:*

(pcg-in) $\texttt{letrec } x = s, Env \texttt{ in } C[s] \rightarrow \texttt{letrec } x = s, Env \texttt{ in } C[x]$
(pcg-e) $\texttt{letrec } x = s, Env, y = C[s] \texttt{ in } r \rightarrow \texttt{letrec } x = s, Env, y = C[x] \texttt{ in } r$

Correctness of the transformations (*pcg*) and (*pcgE*) is a consequence of a result shown in [21]:

Proposition 4.2. *The program transformations* (*pcg*) *and* (*pcgE*) *are correct.*

Proof. In [21] the following result for LR was obtained:

> *For an expression s, its infinite tree $IT(s)$ is derived by unfolding all* letrec*-bindings (and removing the* letrec*). If $IT(s) = IT(t)$ for expressions s, t (where $=$ is syntactic equality modulo α-equivalence on infinite trees), then $s \sim_c t$.*

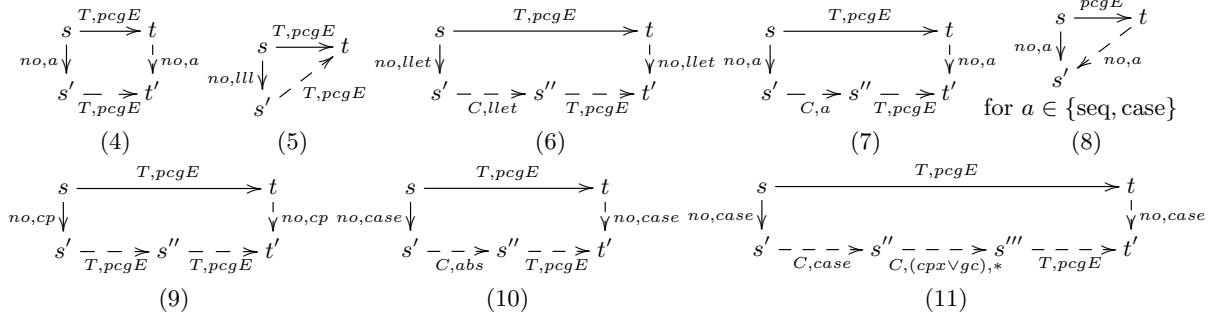---

[4] Note that $x \in FV(s)$ is permitted in (*gcp*).

Fig. 7: Forking diagrams for (pcgE)

Since $s \xrightarrow{pcg \vee pcgE} t$ implies $IT(s) = IT(t)$, correctness of (pcg) and (pcgE) holds.

We first prove that $(pcgE)$ is an improvement in Proposition 4.4, and thereafter we use this result in Lemma 4.5 and Theorem 4.6 to show that (pcg) is an improvement.

Lemma 4.3. *If $s \xrightarrow{T,pcg} s'$ for a WHNF s, then either s' is a WHNF or $s' \xrightarrow{no,cp} s''$ where s'' is a WHNF.*

Proposition 4.4. *If $s \xrightarrow{T,pcgE} t$, then for all $A \subseteq \mathfrak{A}$: $\mathtt{rln}_A(s) \geq \mathtt{rln}_A(t)$ and $\mathtt{rlnall}(s) \geq \mathtt{rlnall}(t)$. Moreover, the $(pcgE)$ is an improvement, i.e. $(pcgE) \subseteq \succeq_A$ for all A.*

Proof. Let $s' \xleftarrow{no,a} s \xrightarrow{T,pcgE} t$. All possible overlappings (forks) can be joined by one of the diagrams in Fig. 7 details are in Appendix C.1. By induction on $\mathtt{rlnall}(s)$ we show that $\mathtt{rlnall}(t) \leq \mathtt{rlnall}(s)$ and $\mathtt{rln}_A(t) \leq \mathtt{rln}_A(s)$. If $\mathtt{rlnall}(s) = 0$ then s is a WHNF, and t must also be a WHNF and $\mathtt{rlnall}(t) = \mathtt{rln}_A(t) = 0$. If $\mathtt{rlnall}(s) > 0$, then let $s \xrightarrow{no,a} s'$.

For diagram (4), we can apply the induction hypothesis to $s' \xrightarrow{T,pcgE} t'$, since $\mathtt{rlnall}(s') = \mathtt{rlnall}(s) - 1$. This shows that $\mathtt{rlnall}(t) \leq \mathtt{rlnall}(s)$ and $\mathtt{rln}_A(t) \leq \mathtt{rln}_A(s)$. For diagram (5), we apply the induction hypothesis to $s' \xrightarrow{T,pcg} t$ which shows $\mathtt{rlnall}(t) \leq \mathtt{rlnall}(s') = \mathtt{rlnall}(s) - 1$ and $\mathtt{rln}_A(t) \leq \mathtt{rln}_A(s') = \mathtt{rln}_A(s)$. For diagram (6), we have $\mathtt{rlnall}(s) > \mathtt{rlnall}(s')$ and $\mathtt{rln}_A(s) = \mathtt{rln}_A(s')$. By Theorem 3.5 (4) $\mathtt{rlnall}(s'') \leq \mathtt{rlnall}(s')$ and $\mathtt{rln}_A(s'') \leq \mathtt{rln}_A(s')$. We can apply the induction hypothesis to $s'' \xrightarrow{T,pcgE} t'$ which shows $\mathtt{rlnall}(t') \leq \mathtt{rlnall}(s'')$ and $\mathtt{rln}_A(t') \leq \mathtt{rln}(s'')$. This implies $\mathtt{rlnall}(t) \leq \mathtt{rlnall}(s)$ and also $\mathtt{rln}_A(t) \leq \mathtt{rln}_A(s)$. For diagram (7), we have $\mathtt{rlnall}(s') < \mathtt{rlnall}(s)$ and $\mathtt{rln}_A(s') \leq \mathtt{rln}_A(s)$. Theorem 3.5 shows that $\mathtt{rlnall}(s'') \leq \mathtt{rlnall}(s')$ and $\mathtt{rln}_A(s'') \leq \mathtt{rln}_A(s')$. We apply the induction hypothesis to $s'' \xrightarrow{T,pcgE} t'$ and have $\mathtt{rlnall}(t') \leq \mathtt{rlnall}(s'')$ and $\mathtt{rln}_A(t') \leq \mathtt{rln}_A(s'')$. This implies both $\mathtt{rlnall}(t) \leq \mathtt{rlnall}(s)$ as well as $\mathtt{rln}_A(t) \leq \mathtt{rln}_A(s)$. For diagram (8), obviously $\mathtt{rlnall}(s) = \mathtt{rlnall}(t)$ and $\mathtt{rln}_A(s) = \mathtt{rln}_A(t)$ hold. For diagram (9), we have $\mathtt{rlnall}(s') < \mathtt{rlnall}(s)$ and $\mathtt{rln}_A(s) = \mathtt{rln}_A(s')$. We apply the induction hypothesis to $s' \xrightarrow{T,pcgE} s''$ and have $\mathtt{rlnall}(s'') \leq \mathtt{rlnall}(s')$ and $\mathtt{rln}_A(s'') \leq \mathtt{rln}_A(s')$. Applying the induction hypothesis to $s'' \xrightarrow{T,pcgE} t'$ yields $\mathtt{rlnall}(t') \leq \mathtt{rlnall}(s'')$ and $\mathtt{rln}_A(t'') \leq \mathtt{rln}_A(s'')$. Since $t \xrightarrow{no,cp} t'$, this shows $\mathtt{rlnall}(t) \leq \mathtt{rlnall}(s)$ and $\mathtt{rln}_A(t) \leq \mathtt{rln}_A(s)$. For diagram (10), we have $\mathtt{rlnall}(s') < \mathtt{rlnall}(s)$ and $\mathtt{rln}_A(s) \leq \mathtt{rln}_A(s')$. By Theorem 3.5 (5) we have $\mathtt{rln}_A(s'') \leq \mathtt{rln}_A(s')$ and $\mathtt{rlnall}(s'') \leq \mathtt{rlnall}(s')$. Applying the induction hypothesis to $s'' \xrightarrow{T,pcgE} t'$ yields $\mathtt{rlnall}(t') \leq \mathtt{rlnall}(s'')$ and $\mathtt{rln}_A(t') \leq \mathtt{rln}_A(s'')$ which shows $\mathtt{rlnall}(t) \leq \mathtt{rlnall}(s)$ and $\mathtt{rln}_A(t) \leq \mathtt{rln}_A(s)$. For diagram (11), we have $\mathtt{rlnall}(s') < \mathtt{rlnall}(s)$ and $\mathtt{rln}_A(s) \leq \mathtt{rln}_A(s')$. By Theorem 3.5 (1), (4), (5) we have $\mathtt{rln}_A(s''') \leq \mathtt{rln}_A(s')$ and $\mathtt{rlnall}(s''') = \mathtt{rlnall}(s')$. Applying the induction hypothesis to $s''' \xrightarrow{T,pcgE} t'$ yields $\mathtt{rlnall}(t') \leq \mathtt{rlnall}(s''')$ and $\mathtt{rln}_A(t') \leq \mathtt{rln}_A(s'')$ which shows $\mathtt{rlnall}(t) \leq \mathtt{rlnall}(s)$ and $\mathtt{rln}_A(t) \leq \mathtt{rln}_A(s)$.

Since (pcgE) is correct (Proposition 4.2), Lemma 3.7 shows $(pcgE) \subseteq \succeq$. $\qed$

Fig. 8: Forking-diagrams for (pcg)

Lemma 4.5. *For closed s with $s \xrightarrow{T,pcg} s'$: $\mathtt{rln}_A(s) \geq \mathtt{rln}_A(s')$ for all A.*

Proof. Let $s{\downarrow}$, and $s \xrightarrow{T,pcg} s'$. Let $\emptyset \neq A \subseteq \mathfrak{A}$ be arbitrary but fixed. We show $\mathtt{rln}_A(s) \geq \mathtt{rln}_A(s')$ by induction on the measure $(\mathtt{rln}_{\mathfrak{A}}(s), \mathtt{rlnall}(s))$, ordered lexicographically. For the base case $\mathtt{rln}_{\mathfrak{A}}(s) = \mathtt{rlnall}(s) = 0$, the expression s is a WHNF and Lemma 4.3 shows that $\mathtt{rln}_A(s') = 0$.

For the induction step, Fig. 8 shows the overlappings between normal order reduction-steps and a $\xrightarrow{T,pcg}$-transformation (details are in Appendix C.2) where the cases that the (T,pcg)-transformation is an inverse (C,cp)- or (C,cpx)-transformation are not covered, since in these cases Theorem 3.5 (5), or Theorem 3.8 show the claim. We consider the remaining cases:

1. If $s \xrightarrow{no,case\vee seq\vee lbeta} t_1$, then diagram (12), (15), (16), (18), or (19) of Fig. 8 holds. The diagrams can be summarized as follows where $a \in \{case, seq, lbeta\}$:

We have $\mathtt{rln}_{\mathfrak{A}}(t_1) = \mathtt{rln}_{\mathfrak{A}}(s) - 1$ and by Theorem 3.5 we have $\mathtt{rln}_{\mathfrak{A}}(t_3) \leq \mathtt{rln}_{\mathfrak{A}}(t_1)$ and $\mathtt{rln}_A(t_3) \leq \mathtt{rln}_A(t_1)$. We apply the induction hypothesis for every step in $t_3 \xrightarrow{T,pcg,*} t_4$ and we derive $\mathtt{rln}_{\mathfrak{A}}(t_4) \leq \mathtt{rln}_{\mathfrak{A}}(t_1) < \mathtt{rln}_{\mathfrak{A}}(s)$ and $\mathtt{rln}_A(t_4) \leq \mathtt{rln}_A(t_1)$. Theorem 3.5 shows that $\mathtt{rln}_{\mathfrak{A}}(t_4) = \mathtt{rln}_{\mathfrak{A}}(t_5)$ and $\mathtt{rln}_A(t_4) = \mathtt{rln}_A(t_5)$, and thus $\mathtt{rln}_{\mathfrak{A}}(s') = \mathtt{rln}_{\mathfrak{A}}(t_5) + 1 \leq \mathtt{rln}_{\mathfrak{A}}(t_1) + 1 = \mathtt{rln}_{\mathfrak{A}}(s)$ and either $\mathtt{rln}_A(s') = \mathtt{rln}_A(t_5) \leq \mathtt{rln}_A(t_1) = \mathtt{rln}_A(s)$ (if $a \notin A$), or $\mathtt{rln}_A(s') = \mathtt{rln}_A(t_5) + 1 \leq \mathtt{rln}_A(t_1) + 1 = \mathtt{rln}_A(s)$ (if $a \in A$).

2. Let $s \xrightarrow{no,cp} t_1$. If t_1 is a WHNF, then by Lemma 4.3 $s' \xrightarrow{no,cp,0\vee 1} t'_1$ where t'_1 is a WHNF and $\mathtt{rln}_{\mathfrak{A}}(s') \leq \mathtt{rln}_{\mathfrak{A}}(s)$ and also $\mathtt{rln}_A(s') \leq \mathtt{rln}_A(s)$ holds. If t_1 is not a WHNF, then $t_1 \xrightarrow{no,a} t_2$ where $a \in \{(lbeta), (seq)\}$ and diagram (13) or (14) of Fig. 8 holds. For diagram (13) we have:

Then $\mathtt{rln}_{\mathfrak{A}}(t_2) < \mathtt{rln}_{\mathfrak{A}}(s)$ and we apply the induction hypothesis to $t_2 \xrightarrow{T,pcg} t_3$ which shows $\mathtt{rln}_{\mathfrak{A}}(t_3) \leq \mathtt{rln}_{\mathfrak{A}}(t_2) < \mathtt{rln}_{\mathfrak{A}}(s)$. We then apply the induction hypothesis to $t_3 \xrightarrow{T,pcg,0\vee 1} t_4$ which shows $\mathtt{rln}_{\mathfrak{A}}(t_4) \leq \mathtt{rln}_{\mathfrak{A}}(t_2) < \mathtt{rln}_{\mathfrak{A}}(s)$, $\mathtt{rln}_{\mathfrak{A}}(s') \leq \mathtt{rln}_{\mathfrak{A}}(s)$, and either $\mathtt{rln}_A(s') = \mathtt{rln}_A(t_4) \leq \mathtt{rln}_A(t_2) = \mathtt{rln}_A(s)$ (if $a \notin A$) or $\mathtt{rln}_A(s') = \mathtt{rln}_A(t_4) + 1 \leq \mathtt{rln}_A(t_2) + 1 = \mathtt{rln}_A(s)$ (if $a \in A$).

Similarly, in diagram (14) the situation is: $t_2 \xleftarrow{no,a} t_1 \xleftarrow{no,cp} s \xrightarrow{pcg} s' \xrightarrow{no,cp} t_1' \xrightarrow{no,a} t_5$, and $t_2 \xrightarrow{C,cp} t_3 \xrightarrow{C,a} t_4 \xrightarrow{T,pcg} t_5$. Then $\mathtt{rln}_{\mathfrak{A}}(t_2) < \mathtt{rln}_{\mathfrak{A}}(s)$, $\mathtt{rln}_A(t_2) \leq \mathtt{rln}_A(s)$ and by Theorem 3.5 (1) $\mathtt{rln}_{\mathfrak{A}}(t_4) \leq \mathtt{rln}_{\mathfrak{A}}(t_2)$, $\mathtt{rln}_A(t_4) \leq \mathtt{rln}_A(t_2)$ and we apply the induction hypothesis to $t_4 \xrightarrow{T,pcg} t_5$ and have $\mathtt{rln}_{\mathfrak{A}}(t_5) \leq \mathtt{rln}_{\mathfrak{A}}(t_4) < \mathtt{rln}_{\mathfrak{A}}(s)$ and $\mathtt{rln}_{\mathfrak{A}}(s') \leq \mathtt{rln}_{\mathfrak{A}}(s)$ and also $\mathtt{rln}_A(s') \leq \mathtt{rln}_A(s)$.

3. If $s \xrightarrow{no,lll} t_1$, then diagram (12), (15), or (17) of Fig. 8 holds, which can be summarized as follows:

$$
\begin{array}{ccc}
s & \xrightarrow{\quad T,pcg \quad} & s' \\
{\scriptstyle no,lll} \downarrow & & \downarrow {\scriptstyle no,lll} \\
t_1 \underset{C,lll,*\vee T,pcgE}{\dashrightarrow} & t_2 \underset{pcg}{\dashrightarrow} & t_3
\end{array}
$$

Then $\mathtt{rln}_{\mathfrak{A}}(t_1) = \mathtt{rln}_{\mathfrak{A}}(s)$, $\mathtt{rln}_A(t_1) = \mathtt{rln}_A(s)$, and $\mathtt{rlnall}(t_1) = \mathtt{rlnall}(s) - 1$. Theorem 3.5 (4), and Proposition 4.4 show that $\mathtt{rln}_{\mathfrak{A}}(t_2) \leq \mathtt{rln}_{\mathfrak{A}}(t_1)$, $\mathtt{rln}_A(t_2) \leq \mathtt{rln}_A(t_1)$, and $\mathtt{rlnall}(t_2) \leq \mathtt{rlnall}(t_1)$. Thus we can apply the induction hypothesis to $t_2 \xrightarrow{T,pcg} t_3$ which yields $\mathtt{rln}_A(t_3) \leq \mathtt{rln}_A(t_2)$ (also for $A = \mathfrak{A}$). Since $s' \xrightarrow{no,lll} t_3$, we have $\mathtt{rln}_A(s') = \mathtt{rln}_A(t_3) \leq \mathtt{rln}_A(t_2) = \mathtt{rln}_A(t_1) = \mathtt{rln}_A(s)$ which shows the claim. □

Theorem 4.6. *The program transformations (pcg), (pcgE), and (cse) are improvements.*

Proof. For (*pcgE*) the claim is proved in Proposition 4.4 and for (*pcg*) this follows from Lemma 4.5, correctness of (*pcg*) (Proposition 4.2) and the context lemma for improvement (Lemma 3.7).

As already demonstrated, the transformation (*cse*) can be represented as a sequence $\xleftarrow{gc} \cdot \xrightarrow{pcg,*}$ and since $(gc) \subseteq \approx_A$ for all A by Theorem 3.9, this shows that (*cse*) is an improvement for all A.

5   Abstract Machine Semantics and Analysis of Resource Measures

The goal of this section is to investigate the relationship between the measure $\mathtt{rln}(\cdot)$ and the counting measures used in [10,5] for their improvement relations. The following results are obtained in this section:

- In Theorem 5.15 we show that $\mathtt{rln}_A(\cdot)$ coincides with the number of essential transition steps $\mathtt{mln}_A(\cdot)$ for all A of the abstract machine of [10].
- In Theorem 5.19 we compare the number of all transitions steps (the measure used by [10]) with our measure and the measure used by [5].
- For normal forms under (lll)-, (gc)- and (cpax)-reductions, the main measures for the machine transitions are almost the same. (see Theorem 5.24).
- We show that the complexity of first-order functions is the same under various measures of reductions.

To compare and relate the resource consumption of two program calculi, we define the notion of an *asymptotically resource-preserving translation*. Therefore, we use the *O*-notation as follows. For functions $f, g : E \to \mathbb{N}$, we write $f \in O(g)$, if there is a constant $c > 0$, s.t. for all $e \in E$: $f(e) \leq c \cdot g(e)$.

Definition 5.1. *Let $\mathcal{K}_1 = (E_1, \sim_1, \mathtt{size}_1, \mu_1)$, $\mathcal{K}_2 = (E_2, \sim_2, \mathtt{size}_2, \mu_2)$ be two calculi with sets of expressions E_i, contextual equivalences $\sim_i$, size-measures $\mathtt{size}_i$ for expressions, and measures μ_i for reduction length of expressions. A translation $\phi_1 : \mathcal{K}_1 \to \mathcal{K}_2$ is size-preserving, iff $\mathtt{size}_1(e) \in O(\mathtt{size}_2(\phi(e)))$ and ϕ is fully abstract; i.e., for all $e, e' \in E_1$: $e \sim_1 e' \iff \phi_1(e) \sim_2 \phi_1(e')$. A translation $\phi : \mathcal{K}_1 \to \mathcal{K}_2$ is* asymptotically resource-preserving, *iff ϕ is size-preserving and there exists an $n \in \mathbb{N}$ with $\mu_1(e_1) \in O(\mathtt{size}_2(\phi_1(e_1))^n \cdot (\mu_2(\phi_1(e_1)) + 1))$.*

In Theorem 5.21 we prove several results on asymptotic resource-preserving translations between the calculus LR and the abstract machine of [10] w.r.t. different measures for reduction lengths.

$$
\begin{array}{ll}
\text{(Lookup)} & \langle \Gamma, x = s \mid x \mid S \rangle \rightharpoonup \langle \Gamma \mid s \mid \#\mathsf{upd}(x) : S \rangle \\
\text{(Update)} & \langle \Gamma \mid v \mid \#\mathsf{upd}(x) : S \rangle \rightharpoonup \langle \Gamma, x = v \mid v \mid S \rangle \quad \text{where } v \text{ is a value } (v = \lambda x.s \text{ or } v = c\,\overrightarrow{y}) \\
\text{(Unwind1)} & \langle \Gamma \mid (s\ x) \mid S \rangle \rightharpoonup \langle \Gamma \mid s \mid \#\mathsf{app}(x) : S \rangle \\
\text{(Unwind2)} & \langle \Gamma \mid (\mathsf{seq}\ s\ x) \mid S \rangle \rightharpoonup \langle \Gamma \mid s \mid \#\mathsf{seq}(x) : S \rangle \\
\text{(Unwind3)} & \langle \Gamma \mid \mathsf{case}_K\ s\ alts \mid S \rangle \rightharpoonup \langle \Gamma \mid s \mid \#\mathsf{case}(alts) : S \rangle \\
\text{(Subst)} & \langle \Gamma \mid \lambda x.s \mid \#\mathsf{app}(y) : S \rangle \rightharpoonup \langle \Gamma \mid s[y/x] \mid S \rangle \\
\text{(Seq)} & \langle \Gamma \mid v \mid \#\mathsf{seq}(y) : S \rangle \rightharpoonup \langle \Gamma \mid y \mid S \rangle \quad \text{where } v \text{ is a value } (v = \lambda x.s \text{ or } v = c\,\overrightarrow{y}) \\
\text{(Branch)} & \langle \Gamma \mid c_{i,K}\ \overrightarrow{x} \mid \#\mathsf{case}(\ldots\ (c_{i,K}\ \overrightarrow{y}\ \text{->}\ t)\ldots) : S \rangle \rightharpoonup \langle \Gamma \mid t[\overrightarrow{x}/\overrightarrow{y}] \mid S \rangle \\
\text{(Letrec)} & \langle \Gamma \mid \mathtt{letrec}\ Env\ \mathtt{in}\ s \mid S \rangle \rightharpoonup \langle \Gamma, Env \mid s \mid S \rangle
\end{array}
$$

Fig. 9: Machine transitions

5.1 Abstract Machine Semantics

We recall the abstract machine used by [10] slightly extended to handle seq-expression which were not present in the work of [10].

Definition 5.2. *The syntax of* machine expressions *is the same as the syntax for* LR-*expressions except that argument positions are restricted to variables, i.e. in applications* $(s\ t)$, seq-*expressions* $(\mathsf{seq}\ s\ t)$, *and constructor applications* $(c\ t_1 \ldots t_{ar(c)})$ *the expressions* t, t_i *must be variables.*

We define a translation from LR-expressions into machine expressions:

Definition 5.3. *The translation* ψ *from* LR-*expressions into machine expressions is*

$$
\begin{array}{ll}
\psi(x) & := x, \quad \text{if } x \in \mathit{Var} \\
\psi(s\ t) & := \mathtt{letrec}\ x = \psi(t)\ \mathtt{in}\ (\psi(s)\ x) \\
\psi(\mathsf{seq}\ s\ t) & := \mathtt{letrec}\ x = \psi(t)\ \mathtt{in}\ (\mathsf{seq}\ \psi(s)\ x) \\
\psi(c\ s_1\ \ldots\ s_n) & := \mathtt{letrec}\ x_1 = \psi(s_1), \ldots, x_n = \psi(s_n)\ \mathtt{in}\ (c\ x_1\ \ldots\ x_n) \\
\psi(M[s_1, \ldots, s_n]) & := M[\psi(s_1), \ldots, \psi(s_n)],
\end{array}
$$
for all multicontexts M *of the forms* $\mathtt{letrec}\ x_1 = [\cdot], \ldots, x_n = [\cdot]\ \mathtt{in}\ [\cdot]$, $\lambda x.[\cdot]$, *or* $\mathsf{case}_K\ [\cdot]\ (pat_1 \text{->} [\cdot]) \ldots (pat_n \text{->} [\cdot])$.

Lemma 5.4. *For* $s \in \mathit{Expr}$, $\mathtt{size}(s) \leq \mathtt{size}(\psi(s)) \leq \max(3, 2 + maxArityOfConstructors(s)) \cdot \mathtt{size}(s)$ *where* $maxArityOfConstructors(s)$ *is the maximal arity of constructors that occur in* s.

Proof. Since $\mathtt{size}(\psi(s\ t)) = \mathtt{size}(\mathtt{letrec}\ x = \psi(t)\ \mathtt{in}\ (\psi(s)\ x)) = 3 + \mathtt{size}(\psi(s)) + \mathtt{size}(\psi(t))$, the factor 3 is required. $\mathtt{size}(\psi(c\ s_1 \ldots s_n)) = \mathtt{size}(\mathtt{letrec}\ x_1 = \psi(t_1), \ldots, x_n = \psi(t_n)\ \mathtt{in}\ (c\ x_1 \ldots x_n)) = 2 + n + \sum_i \mathtt{size}(\psi(s_i))$, which has to be compared with $\mathtt{size}(c\ s_1 \ldots s_n)$, which is $1 + \sum \mathtt{size}(s_i)$, hence the factor $2 + n$ is required. So we can take the maximum for an estimation.

Since $\psi(t) \xrightarrow{C, ucp, *} t$, Theorem 3.5 (6) implies:

Lemma 5.5. *For all closed* $t \in \mathit{Expr}$ *and all* A: $\mathtt{rln}_A(t) = \mathtt{rln}_A(\psi(t))$.

Definition 5.6. *A state* Q *of the machine is a tuple* $\langle \Gamma \mid s \mid S \rangle$, *where* Γ *is an environment of bindings (like a* $\mathtt{letrec}$-*environment),* s *is a machine expression, and* S *is a stack, with entries* $\#\mathsf{upd}(x)$, $\#\mathsf{app}(x)$, $\#\mathsf{seq}(x)$, $\#\mathsf{case}(alts)$ *where* x *is a variable and* $alts$ *is a set of* case-*alternatives. We use list notation for the stack* S. *The transition rules of the machine are shown in Fig. 9. With (*Unwind*) we denote the union of (*Unwind1*), (*Unwind2*), and (*Unwind3*). The machine starts with* $\langle \emptyset \mid s \mid [] \rangle$ *for an expression* s *and an accepting state is of the form* $\langle \Gamma \mid v \mid [] \rangle$ *where* v *is a value (i.e. an abstraction or a constructor application). A machine state* $\langle \Gamma \mid s \mid S \rangle$ *is* reachable *iff there exists an expression* t *s.t.* $\langle \emptyset \mid t \mid [] \rangle \xrightarrow{*} \langle \Gamma \mid s \mid S \rangle$.

We define a mapping ϕ from reachable machine states to machine expressions:

$$\phi(\langle \Gamma \mid s \mid \#\mathsf{upd}(x) : S \rangle) := \phi(\langle \Gamma, x = s \mid x \mid S \rangle) \qquad \phi(\langle \Gamma \mid s \mid \#\mathsf{app}(x) : S \rangle) \quad := \phi(\langle \Gamma \mid (s\ x) \mid S \rangle)$$
$$\phi(\langle \Gamma \mid s \mid \#\mathsf{seq}(x) : S \rangle) := \phi(\langle \Gamma \mid (\mathtt{seq}\ s\ x) \mid S \rangle) \qquad \phi(\langle \Gamma \mid s \mid \#\mathsf{case}(alts) : S \rangle) := \phi(\langle \Gamma \mid (\mathtt{case}\ s\ alts) \mid S \rangle)$$
$$\phi(\langle \Gamma \mid s \mid [] \rangle) \qquad\qquad\quad := \mathtt{letrec}\ \Gamma\ \mathtt{in}\ s$$

Note that $\phi(\langle \Gamma \mid v \mid [] \rangle) = \mathtt{letrec}\ \Gamma\ \mathtt{in}\ v$ and thus accepting states are mapped to WHNFs. Let $\psi(lbeta) = \text{Subst}$, $\psi(case) = \text{Branch}$, and $\psi(seq) = \text{Seq}$, and for $\emptyset \neq A \subseteq \{lbeta, case, seq\}$ let $\psi(A) := \{\psi(a) \mid a \in A\}$, in particular $\psi(\mathfrak{A}) = \{\text{Subst}, \text{Branch}, \text{Seq}\}$.

Definition 5.7. *Let s be a closed machine expression such that $\langle \emptyset \mid s \mid [] \rangle \xrightarrow{n} Q$ where Q is an accepting state. Then $\mathtt{mlnall}(s) = n$, and $\mathtt{mln}_A(s)$ counts the number of the (Subst)-, (Branch)-, and (Seq)-steps which are in $\psi(A)$. The number $\mathtt{mlnlook}(s)$ counts all (Lookup)-transitions. If no such sequence exists for s, then $\mathtt{mlnall}(s) = \mathtt{mln}_{\mathfrak{A}}(s) = \mathtt{mln}_A(s) = \mathtt{mlnlook}(s) = \infty$. We use $\mathtt{mln}(\cdot)$ with the same meaning also for reachable states Q of the machine.*

Note that the improvement theory in [10] is based on the measure $\mathtt{mlnall}(\cdot)$, whereas the measure in [5] is $\mathtt{mlnlook}(\cdot)$ (see the comparison Theorems 5.21 and 5.24). Note also that the complexity of reduction of expressions is adequately measured with $\mathtt{mlnall}$:

Proposition 5.8. *The number of steps k on a RAM executing the reduction of a machine expression s on the abstract machine is smaller than $a \cdot \mathtt{size}(s) \cdot \mathtt{mlnall}(s)$ where a is a constant independent of s. This is under the assumption that storing and accessing the store requires constant time.*

Proof. The transitions (Lookup), (Unwind), (Seq), and (Letrec) require constant time, and the transitions (Update), (Subst), and (Branch) require time $O(\mathtt{size}(s))$, since the size of abstractions and case-alternatives in the transition sequence starting with $\langle \emptyset \mid s \mid [] \rangle$ is at most $\mathtt{size}(s)$.

5.2 Relating the Essential Reduction Steps

In this section we show that for a machine expression s the number of essential transition steps coincides with the number of essential normal order reductions in LR, i.e. we show that $\mathtt{rln}_A(s) = \mathtt{mln}_A(s)$. With Lemma 5.5 this also implies that for every LR-expression s the equation $\mathtt{rln}_A(s) = \mathtt{mln}_A(\phi(s))$ holds.

Lemma 5.9. *For a reachable state Q with $Q \rightharpoonup Q'$, one of the following cases holds for $\phi(Q)$ and $\phi(Q')$:*

Proposition 5.10. *For a closed machine expression s and $\emptyset \neq A \subseteq \mathfrak{A}$, $\mathtt{mln}_A(s) = n$ implies $\mathtt{rln}_A(s) = n$.*

Proof. Let $\mathtt{mln}_A(s) = n$. We consider the sequence of machine transitions from $\langle \emptyset \mid s \mid [] \rangle$ to an accepting state and construct a sequence of (no,lbeta)-, (no,case-c)-, (no,seq)-, (T,cp)-, (T,cpcx)-, (T,cpax)-, (T,lll)-, and (T-gc)-transformations from s to a WHNF.

So let $Q_0 = \langle \emptyset \mid s \mid [] \rangle \xrightarrow{k} Q_k$ where Q_k is an accepting state. We use induction on k: If $k = 0$ then $s = \phi(Q_0)$ is a WHNF. If $k > 0$ then we apply Lemma 5.9 to $Q_0 \rightharpoonup Q_1$ and then apply the induction

(mo,cpv-e) $\texttt{letrec } Env, x = v^{\mathsf{sub}}, y = C[x^{\mathsf{sub}\vee\mathsf{nontarg}}] \texttt{ in } s \rightarrow \texttt{letrec } Env, x = v, y = C[v] \texttt{ in } s$
  where v is a value

(mo,cpv-in) $\texttt{letrec } Env, x = v^{\mathsf{sub}} \texttt{ in } C[x^{\mathsf{sub}}] \rightarrow \texttt{letrec } Env, x = v \texttt{ in } C[v]$ where v is a value

(mo,β-var) $C[(\lambda x.s)^{\mathsf{sub}} \ y] \rightarrow C[s[y/x]]$

(mo,casecx) $C[\texttt{case}_K \ (c_{K,i} \overrightarrow{x})^{\mathsf{sub}} \dots (c_{K,i} \overrightarrow{y} \texttt{ -> } s) \dots] \rightarrow C[s[\overrightarrow{x}/\overrightarrow{y}]]$

(mo,seq-c) $C[\texttt{seq } v^{\mathsf{sub}} \ x] \rightarrow C[x]$

(mo,glletm) $R^-[(\texttt{letrec } Env \texttt{ in } s)^{\mathsf{sub}}] \rightarrow \texttt{letrec } Env \texttt{ in } R^-[s]$

(mo,gllet-in) $\texttt{letrec } Env_1 \texttt{ in } C[(\texttt{letrec } Env_2 \texttt{ in } s)^{\mathsf{sub}}] \rightarrow \texttt{letrec } Env_1, Env_2 \texttt{ in } C[s]$

(mo,gllet-e) $\texttt{letrec } y = C[(\texttt{letrec } Env_1 \texttt{ in } s)^{\mathsf{sub}}], Env_2 \texttt{ in } t \rightarrow \texttt{letrec } y = C[s], Env_1, Env_2 \texttt{ in } t$
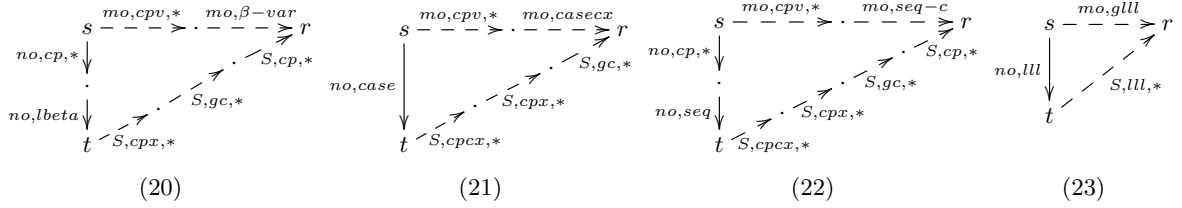
Fig. 10: Machine order reduction rules

Fig. 11: Diagrams for transferring normal order reductions into machine order reductions

hypothesis to $Q_1 \xrightarrow{k-1} Q_k$. This construction gives a sequence of transformations from $s = \phi(Q_0)$ to a WHNF, where the sum of (no,lbeta)-, (no,case-c)-, and (no,seq)-steps is n.

Now iteratively apply Theorems 3.5 and 3.8 from right to left to every transformation which is not a normal order reduction. Since all these steps leave the measure $\texttt{rln}_A(\cdot)$ unchanged, and the normal order step increases the measure by 1, this shows $\texttt{rln}_A(s) = n$.

We define a variant of the normal order reduction for machine expressions – called *machine order reduction*: It uses the reduction rules shown in Fig. 10 and the machine order redex is found by the labeling algorithm in Definition 2.4. Let (mo,cpv) be the union of (mo,cpv-e) and (mo,cpv-in), and (mo,glll) be the union of (mo,glletm), (mo,gllet-in), and (mo,gllet-e). A *machine order WHNF* (MWHNF) is a value or an expression of the form $\texttt{letrec } Env \texttt{ in } v$ where v is a value. Let $\overline{\psi}(lbeta) = (mo,\beta\text{-var})$, $\overline{\psi}(case) = (mo, casecx)$, and $\overline{\psi}(seq) = (mo, seqc)$. For a closed expression s and $\emptyset \neq A \subseteq \mathfrak{A}$, let $\texttt{rln}_{mo,A}(s)$ be the number of (mo, a)-reductions with $(mo, a) \in \overline{\psi}(A)$ in a machine order reduction sequence from s to an MWHNF, and $\texttt{rln}_{mo,A}(s) = \infty$ otherwise.

Lemma 5.11. *If the machine expression s is a WHNF, then either s is also an MWHNF, or $s \xrightarrow{mo,cpv,*} s'$ where s' is an MWHNF.*

Lemma 5.12. *Let $s \xrightarrow{no,a} t$ where a is not a (cp), or $s \xrightarrow{no,cp,*} s' \xrightarrow{no,seq\vee lbeta} t$. The diagrams in Fig. 11 show how at least one machine order reduction can be performed for s, s.t. $s \xrightarrow{mo,+} r$ and how t and r are joinable by program transformations.*

Proposition 5.13. *For all $\emptyset \neq A \subseteq \mathfrak{A}$, closed machine expressions s: $\texttt{rln}_A(s) = n \implies \texttt{rln}_{mo,A}(s) = n$.*

Proof. Let $s \xrightarrow{no,k} t_k$ where t_k is a WHNF. We use induction on $(\texttt{rln}_\mathfrak{A}(s), \texttt{rlnall}(s))$ to show the claim. If $\texttt{rlnall}(s) = \texttt{rln}_A(s) = 0$, then s is a WHNF, and $s \xrightarrow{mo,cpv,*} s'$ where s' is an MWHNF and thus $\texttt{rln}_{mo,A}(s) = 0$. Now assume $\texttt{rlnall}(s) > 0$. If $s \xrightarrow{no,cp} s'$, where s' is a WHNF, then $s \xrightarrow{mo,cpv,*} s''$, where s'' is a WHNF, and so $\texttt{rln}_A(s) = \texttt{rln}_{mo,A}(s'') = 0$. In the other cases we apply a diagram from Lemma 5.12 to a prefix of $s \xrightarrow{no,k} t_k$. For diagram (23) we have $\texttt{rln}_\mathfrak{A}(t) = \texttt{rln}_\mathfrak{A}(s)$, $\texttt{rlnall}(t) < \texttt{rlnall}(s)$, and $\texttt{rln}_A(t) = \texttt{rln}_A(s)$ By Theorem 3.5 (4) $\texttt{rln}_A(r) = \texttt{rln}_A(s)$ (also for $A = \mathfrak{A}$) and $\texttt{rlnall}(r) < \texttt{rlnall}(s)$. We apply the induction hypothesis to r and get $\texttt{rln}_{mo,A}(r) = \texttt{rln}_A(s)$ and thus $\texttt{rln}_{mo,A}(s) = \texttt{rln}_A(s)$.

If diagram (21), (22), or (20) is applied, then $\mathtt{rln}_\mathfrak{A}(t) = \mathtt{rln}_\mathfrak{A}(s) - 1$ and Theorem 3.5 shows that $\mathtt{rln}_A(r) = \mathtt{rln}_A(t)$. Let $\xrightarrow{no,a}$ be the given normal order reduction which is not a (no, cp). Applying the induction hypothesis to r shows $\mathtt{rln}_{mo,A}(r) = \mathtt{rln}_A(s) - 1$ (if $a \in A$) or $\mathtt{rln}_{mo,A}(r) = \mathtt{rln}_A(s)$ (if $a \notin A$). Since $s \xrightarrow{mo,*} r$ where exactly one (mo,casecx)-, (mo,seq-c)-, or (mo,β-var)-reduction is in the sequence, where $\overline{\psi}(a)$ is this reduction, we have $\mathtt{rln}_{mo,A}(s) = \mathtt{rln}_A(s)$. $\qquad\square$

Proposition 5.14. *For all $\emptyset \neq A \subseteq \mathfrak{A}$ and closed machine expressions s: $\mathtt{rln}_A(s) = n \implies \mathtt{mln}_A(s) = n$.*

Proof. From $\mathtt{rln}_A(s) = n$ we get $\mathtt{rln}_{mo,A}(s) = n$ by Proposition 5.13. Let $s \xrightarrow{mo,k} s'$ where s' is an MWHNF. By induction on k, we show that for every reachable machine state Q_0 with $\phi(Q_0) = s$ there exists an accepting state Q_m s.t. $Q_0 \xrightarrow{*} Q_m$ and $\mathtt{mln}(Q_0) = n$. If $k = 0$, then $Q_0 \xrightarrow{\text{Letrec},0\vee1} Q'$ where Q' is accepting. For $k > 0$, let $s \xrightarrow{mo} s_0 \xrightarrow{mo,k-1} s'$. The following diagrams (where (UL) is (Unwind) $\vee$ (Lookup)) show the relationship between $s \xrightarrow{mo} s_0$ and the machine transition for Q_0:

$$
\begin{array}{cccccccccc}
s \xrightarrow{mo,cpv} s_0 & \quad & s \xrightarrow{mo,\beta-var} s_0 & \quad & s \xrightarrow{mo,seqc} s_0 & \quad & s \xrightarrow{mo,casecx} s_0 & \quad & s \xrightarrow{mo,glll} s_0 \\
\phi\uparrow \quad\quad \uparrow\phi & & \phi\uparrow \quad\quad \uparrow\phi & & \phi\uparrow \quad\quad \uparrow\phi & & \phi\uparrow \quad\quad \uparrow\phi & & \phi\uparrow \quad\quad \uparrow\phi \\
Q_0 \underset{UL,*}{\dashrightarrow} \underset{\text{Update}}{\dashrightarrow} Q_1 & & Q_0 \underset{UL,*}{\dashrightarrow} \underset{\text{Subst}}{\dashrightarrow} Q_1 & & Q_0 \underset{UL,*}{\dashrightarrow} \underset{\text{Seq}}{\dashrightarrow} Q_1 & & Q_0 \underset{UL,*}{\dashrightarrow} \underset{\text{Branch}}{\dashrightarrow} Q_1 & & Q_0 \underset{UL,*}{\dashrightarrow} \underset{\text{Letrec}}{\dashrightarrow} Q_1
\end{array}
$$

The diagrams show that after applying the induction hypothesis to s_0 and Q_1 we have $\mathtt{rln}_{mo,A}(s) = \mathtt{mln}_A(Q_0)$. Finally, since $\phi(Q_0) = s$ for $Q_0 = \langle \emptyset \mid s \mid [] \rangle$, we have $\mathtt{mln}_A(s) = \mathtt{rln}_{mo,A}(s)$. $\qquad\square$

By Propositions 5.10 and 5.14 and Lemma 5.5 we have:

Theorem 5.15. *For any closed $s \in \textit{Expr}$ and $\emptyset \neq A \subseteq \mathfrak{A}$: $\mathtt{rln}_A(s) = \mathtt{mln}_A(\psi(s))$.*

Corollary 5.16. *The translation ψ seen as a translation from LR to the abstract machine of [10] in the variant presented in this paper is fully-abstract.*

5.3 Relating Essential and All Transition Steps

In this section, we compare our different length measures and relate counting the essential transition steps and counting all transition steps. In the following, we write (ULLU) for the union of (Unwind), (Letrec), (Lookup), (Update) and (SBS) for the union of (Subst), (Branch), (Seq).

Theorem 5.17. *For all closed machine expressions s with $s\downarrow$: $\mathtt{mlnall}(s) \leq 3 \cdot (\mathtt{size}(s) + 2) \cdot (\mathtt{mln}_\mathfrak{A}(s) + 1)$.*

Proof. Let $\mathtt{mln}_\mathfrak{A}(s) = n$ and $Q_0 = \langle \emptyset \mid s \mid [] \rangle \xrightarrow{m} Q_m$, where Q_m is an accepting state and the sequence contains n (SBS)-transitions. The number of (Update)-transitions is equal to the number of (Lookup)-transitions. The number of (Unwind)-transitions is equal to the number of (SBS)-transitions. It remains to count the (Letrec)- and the (Lookup)-transitions. First observe that letrec-expressions and -bindings which are generated by an (SBS)-transition are a copy of a subexpression which exists in s (where variables may be permuted). Since a (Letrec)-transition removes the letrec, there are at most $(n+1) \cdot \mathtt{size}(s)$ (Letrec)-transitions. The same argument applies to the *first* (Lookup)-transitions of a binding $x = \ldots$. The number of other (Lookup)- transitions (which are not the first for a binding $x = \ldots$) is bounded by $n + 1$, since for such a (Lookup) the binding must be $x = v$, where v is a value, which implies, that no other (Lookup) transition can follow before another (SBS)-transition is performed. Thus, in total there are at most $(n + 1) \cdot (\mathtt{size}(s) + 1)$ (Lookup)-transitions.

Concluding, in the sequence there are at most $(n + 1) \cdot \mathtt{size}(s)$ (Letrec)-transitions, at most $(n+1) \cdot (\mathtt{size}(s) + 1)$ (Lookup)-transitions, at most $(n+1) \cdot (\mathtt{size}(s) + 1)$ (Update)-transitions, and exactly n (Unwind)-transitions. By adding the n (SBS)-transitions this shows $\mathtt{mlnall}(s) \leq 3 \cdot (n+1) \cdot (\mathtt{size}(s) + 2)$. $\mathtt{mlnall}(s) \leq 3 \cdot (n + 1) \cdot (\mathtt{size}(s) + 2)$.

We analyze whether the measure $\mathtt{mlnlook}(\cdot)$ is appropriate as claimed in [10] and used in [5].

Proposition 5.18. *For all closed machine expressions s with $s{\downarrow}$:* $\mathtt{mlnall}(s) \leq (2 \cdot \mathtt{size}(s) \cdot (\mathtt{mlnlook}(s) + 1))$.

Proof. Consider a valid transition subsequence without a (Lookup)-transition. For every intermediate machine state $m_i = \langle \varGamma \mid s_i \mid S_i \rangle, i = 1, \ldots, n$ consider the expression $u_i = \phi(\langle\ \emptyset \mid s_i \mid S_i \rangle)$. Then $\mathtt{size}(u_i)$ is never increased by the intermediate steps, but strictly decreased by (Subst), (Branch), (Seq), (Update), and (Letrec). The maximal size of u_i is not greater than $\mathtt{size}(s)$, (as already argued) hence $\mathtt{mln}(s)$ + (number of (Update)s) + (number of (Letrec)s) is not greater than $\mathtt{size}(s) \cdot (\mathtt{mlnlook}(s) + 1)$. Since the overall number of (Unwind)s is exactly $\mathtt{mln}(s)$, we obtain $\mathtt{mlnall}(s) \leq (2 \cdot \mathtt{size}(s) \cdot (\mathtt{mlnlook}(s) + 1))$.

Theorem 5.19. *Let s be a closed machine expression. Then* $\mathtt{mlnlook}(s) \leq \mathtt{mlnall}(s) \leq (2 \cdot \mathtt{size}(s) \cdot (\mathtt{mlnlook}(s) + 1))$, *and* $\mathtt{mln}_{\mathfrak{A}}(s) \leq \mathtt{mlnall}(s) \leq 3 \cdot (\mathtt{size}(s) + 4) \cdot (\mathtt{mln}_{\mathfrak{A}}(s) + 1)$.

Note that Theorem 5.19 justifies our claim that common subexpression elimination (also called β-expand) is an improvement in [10] and also in [5]. However, our proofs only show that this is the case if improvement is defined w.r.t. $\mathtt{mln}_A(.)$ in their calculus[5]. Note that also the size is not increased (up to the initial inverse (gc)) by common subexpression elimination.

As direct consequence of Proposition 3.3, Theorem 5.15, and Lemma 5.4 is:

Corollary 5.20. *Let s be an expression. Then for $s' = \psi(s)$:* $\mathtt{rln}_{\mathfrak{A}}(s') \leq (\mathtt{size}(s') + 1) \cdot (\mathtt{rln}_{\{lbeta\}}(s') + 1)$ *as well as* $\mathtt{mln}_{\mathfrak{A}}(s') \leq (\mathtt{size}(s') + 1) \cdot (\mathtt{mln}_{\{lbeta\}}(s') + 1)$. *With the size estimation in Lemma 5.4 this also shows:* $\mathtt{rln}_{\mathfrak{A}}(s) \leq (4 + maxArityOfConstructors(s)) \cdot \mathtt{size}(s) \cdot (\mathtt{rln}_{\{lbeta\}}(s) + 1)$ *and* $\mathtt{mln}_{\mathfrak{A}}(s) \leq (4 + maxArityOfConstructors(s)) \cdot \mathtt{size}(s) \cdot (\mathtt{mln}_{\{lbeta\}}(s) + 1)$.

The results in this section imply:

Theorem 5.21. *The following calculi allow asymptotically resource-preserving translations into each other: (i)* LR *with* $\mathtt{rln}_{\mathfrak{A}}$; *(ii) Moran-Sands calculus with* $\mathtt{mlnall}$; *(iii) Moran-Sands calculus with* $\mathtt{mln}_{\mathfrak{A}}$; *(iv) the Moran-Sands calculus with* $\mathtt{mlnlook}$; *(v) Moran-Sands calculus with* $\mathtt{mln}_{\{lbeta\}}$;*and (vi)* LR *with* $\mathtt{rln}_{\{lbeta\}}$;

Note that in LR switching from $\mathtt{rln}(.)$ to $\mathtt{rlnall}(.)$ is not resource-preserving, since there are LR-expressions s s.t. $\mathtt{rlnall}(s) \in O(\mathtt{size}(s)^n(\mathtt{rln}_{\mathfrak{A}}(s) + 1))$ is false for all n (details are in Appendix B). However, this is not a counter argument against the LR-calculus, but only an argument against an implementation that really mimics the (lll)-reductions.

5.4   Comparison of Measures for Simplified Expressions

We show that after a simplification, the measures $\mathtt{mln}$, $\mathtt{mlnall}$, $\mathtt{mlnlook}$ are almost equivalent.

Definition 5.22. *A closed expression s is a* (lcpgc)-normal form, *if there are no (lll)-, (cpax)- nor (gc)-reductions (the (lcpgc)-reductions) possible for s. For every closed expressions s, its* (lcpgc)-normal form $nf_{lcpgc}(s)$ *can be effectively computed by applying the reductions.*

Lemma 5.23. *For every closed expression s its unique $nf_{lcpgc}(s)$ exists. It can be computed in polynomial time by exhaustively applying (lcpgc)-reductions. If s is a machine expression, then also $nf_{lcpgc}(s)$ is a machine-expression. The expression s is improved by $nf_{lcpgc}(s)$.*

A closed expression in (lcx)-normal form has the following syntactic structure: The letrec-*subexpressions can only be the following: (i) s itself; (ii) the body of abstractions, (iii) alternatives of case-expressions, (iv), the second expression of a seq, (v) the immediate subexpressions of a constructor application; (v) bindings $x = y$ can only be of the form $x = x$.*

Proof. Every reduction sequence using only (lcpgc)-reductions is terminating and confluent [22]. The other properties follow easily by checking the reduction rules.

There is a better estimation in Theorem 5.17 for (lcpgc)-normal-forms s:

Theorem 5.24. *Let s be a closed (machine-)expression with $s\downarrow$ and that is in (lcpgc)-normal form. Then*

1. $\mathtt{mlnall}(s) \leq 5 \cdot (\mathtt{mln}_{\mathfrak{A}}(s) + 1)$.
2. $\mathtt{mln}_{\mathfrak{A}}(s) \leq \mathtt{mlnlook}(s)$.
3. $\mathtt{mlnlook}(s) \leq \mathtt{mln}_{\mathfrak{A}}(s) + 1$.
4. $\mathtt{mlnall}(s) \leq 5 \cdot (\mathtt{mlnlook}(s) + 1)$.

Proof. Let $s = s_0$ be in (lcpgc)-normal form. Let $Q_0\langle \emptyset \mid s_0 \mid [] \rangle \xrightarrow{m} Q_m$, where Q_m is an accepting state, the sequence has n (SBS)-transitions, and let $Q_i = \langle H_i, s_i, S_i \rangle$. Note that $\phi(Q_i)$ is in (lcpgc)-normalform.

If $Q_i \xrightarrow{\text{Letrec}} Q_{i+1}$, then since s is in (lcpgc)-normal form, the subsequent transition applied to Q_{i+1} cannot be a (Letrec)-transition. The transitions that may generate a state $\langle H_i, s_i, S_i \rangle$, where s_i is a $\mathtt{letrec}$-expression are only (Subst) and (Branch), since for Q_j: $\phi(Q_j)$ is in (lcpgc)-normal form. Thus the maximal number of (Letrec)-transitions in the sequence $Q_0 \xrightarrow{m} Q_m$ is at most $(n+1)$.

The number of (Update)-transitions is at most $(n+1)$, since after an (Update)-transition either a final state is reached or an (SBS)-transition must follow: this holds since all $\psi(Q_j)$ are in (lcpgc)-normal form and thus there are no $\mathtt{letrec}$-bindings $x = y$ which implies that after an (Update)-transition the subsequent transition cannot be (Update). Bindings $x = x$ are irrelevant, since their evaluation would imply $\mathtt{mln}_{\mathfrak{A}}(s) = \infty$. Since the number of (Unwind)-transitions is exactly n (since every (SBS)-transition requires a preceeding (Unwind)-transition), and since the number of (Lookup)-transitions is equal to the number of (Update)-transitions, this shows $\mathtt{mlnall}(s) \leq 4 \cdot (n+1) + \mathtt{mln}_{\mathfrak{A}}(s) \leq 5 \cdot (\mathtt{mln}_{\mathfrak{A}}(s) + 1)$.

The other inequations follow from the already provided arguments.

The example $\mathtt{letrec}\ x = \lambda y.\mathtt{seq}\ \mathtt{True}\ (\mathtt{seq}\ \mathtt{True}\ (\ldots (\mathtt{seq}\ \mathtt{True}\ y)\ldots))\ \mathtt{in}\ x\ (x\ \ldots (x\ \mathtt{True})\ldots)$ shows that the relation between $\mathtt{mln}_{\mathfrak{A}}$ and $\mathtt{mln}_{\{lbeta\}}$ is not linear. The number of (*lbeta*)-reductions is n and the number of seq-reductions is $n \cdot m$, where n, m can be chosen independently. In this series of examples the inequation $\mathtt{mln}_{\mathfrak{A}}(s) \leq c \cdot \mathtt{mln}_{\{lbeta\}}(s)$ can be refuted for every constant c.

5.5   Invariance of Complexity of Functions

In this section we want to analyze the relation between complexity of functions, improvement, and different measures. We will apply the analysis to machine expressions and use the mln-measures.

We will use *linear real-valued functions* $f : \mathbb{R} \to \mathbb{R}$ of the form $f\ x = a_1 \cdot x + a_2$, where $a_1, a_2 \in \mathbb{R}$ and $a_1, a_2 \geq 0$; The composition of two linear real-valued functions is again a linear real-valued function.

Definition 5.25. *Let the weak improvement relation $\preceq_{weak}$ be defined by $s \preceq_{weak} t$ iff $s \sim_c t$ and there is a linear real-valued function k, s.t. for all contexts C: $\mathtt{mln}_{\mathfrak{A}}(C[s]) \leq k(\mathtt{mln}_{\mathfrak{A}}(C[t]))$.*

Note that this relation is a precongruence, and that $s \preceq t$ implies $s \preceq_{weak} t$.

The claim that we show in the following is that the complexity of functions on data is not made worse under weak improvements. Roughly, the complexity of a function expression f is defined as a positive real function g, s.t. for all arguments a, $\mathtt{mln}_{\mathfrak{A}}(f\ a) \leq g(\mathtt{size}(a))$.

We will first analyze unary functions that operate on data, i.e. on arguments that do not contain function expressions or abstractions.

Definition 5.26. *A* data expression *in* LR *is a closed expression that consists only of constructors.*

Thus a data expression may be a constant like $\mathtt{True}$, a pair of constants, or a finite list of data expressions.

Definition 5.27. *Let us define an upper* complexity-bound *of a closed expression s as a real-valued function g, s.t. there exists a linear real-valued function k and for all data expressions d:* $\mathtt{mln}_\mathfrak{A}(s\ d) \leq k(g(\mathtt{size}(d)))$.

Proposition 5.28. *If $s \preceq_{weak} t$, then every complexity-bound of t is also a complexity-bound of s.*

Proof. Let g be a complexity-bound of t. Then there exist linear functions k_1, k_2, such that for all data expressions d: $\mathtt{mln}_\mathfrak{A}(s\ d) \leq k_1(\mathtt{mln}_\mathfrak{A}(t\ d))$, and $\mathtt{mln}_\mathfrak{A}(t\ d) \leq k_2(g(\mathtt{size}(d)))$. Then for all data expressions d: $\mathtt{mln}_\mathfrak{A}(s\ d) \leq k_1(\mathtt{mln}_\mathfrak{A}(t\ d)) \leq k_1(k_2(g(\mathtt{size}(d))))$, and $(k_1 \circ k_2)$ is also a linear function. Hence g is also a complexity-bound of s.

Lemma 5.29. *Let s be a machine expression and $\mu, \mu' \in \{\mathtt{mln}_\mathfrak{A}(\cdot), \mathtt{mlnall}(\cdot), \mathtt{mlnlook}(\cdot), \mathtt{mln}_{\{lbeta\}}(\cdot)\}$ and d be a (machine-) data expression i.e. $d = \phi(d')$ for a data expression d'. Then there are linear real-valued functions h_1, h_2, such that $\mu(s\ d) \leq h_2(\mathtt{size}(s)) \cdot h_1(\mu'(s\ d))$.*

Proof. This can be derived mainly from Section 5.3. But an additional argument is required: Applying the formula there would lead to a $h(\mathtt{size}(s\ d))$ factor instead of $h(\mathtt{size}(s))$. However, the data expressions do not contribute to this size component, since the contribution comes from copying abstraction. But there are no abstractions in d, hence the size of any abstraction in $(s\ d)$ is not greater than the size of s. The nontrivial cases can be derived from Theorem 5.17 and Proposition 5.18, and for $\mathtt{mln}_{\{lbeta\}}$, from Proposition 3.3.

Theorem 5.30. *Upper complexity-bounds of* LR-*functions on data are the same for all the measures* $\{\mathtt{mln}_\mathfrak{A}(\cdot), \mathtt{mlnall}(\cdot), \mathtt{mlnlook}(\cdot), \mathtt{mln}_{\{lbeta\}}(\cdot)\}$.

Proof. We apply Lemma 5.29 for an expression s and for $\mu, \mu' \in \{\mathtt{mln}_\mathfrak{A}(\cdot), \mathtt{mlnall}(\cdot), \mathtt{mlnlook}(\cdot), \mathtt{mln}_{\{lbeta\}}(\cdot)\}$. There are linear real-valued functions h_1, h_2, such that for all data expressions d: $\mu(s\ d) \leq h_2(\mathtt{size}(s)) \cdot h_1(\mu'(s\ d))$. If g is a complexity upper bound of s w.r.t. μ', then there is a linear real-valued function k_1 with $\mu'(s\ d) \leq k_1(g(\mathtt{size}(d)))$. Combining this, we obtain the inequation: $\mu(s\ d) \leq h_2(\mathtt{size}(s)) \cdot h_1(k_1(g(\mathtt{size}(d)))$. Since s was chosen fixed, $h_2(\mathtt{size}(s))$ is a constant, hence there is a linear real-valued function k_2 with $\mu(s\ d) \leq k_2(\mathtt{size}(d))$. This argument is valid for all pairs μ, μ', hence the theorem holds.

Extending the analysis to higher-order functions where the arguments d are general, i.e. may also contain abstractions, leads to weaker estimations. However, a natural condition on the expression s and argument d, implies that also in this case, the same bounds are valid. We extend Definition 5.27:

Definition 5.31. *Let s be an expression and let P be a predicate on closed expressions. An upper (higher-order)* complexity-bound *of a (closed) expression s under condition P is a real-valued function g, s.t. there is a linear real-valued function k_1 and for* all closed expressions d that satisfy P: $\mathtt{mln}_\mathfrak{A}(s\ d) \leq k_1(g(\mathtt{size}(d)))$.

Proposition 5.28 still holds for the higher-order complexity bounds and all P, which can be proved using the same estimations. However, Theorem 5.30 is weakened: by switching the measure, the upper higher-order complexity bounds may increase from g to $\lambda x.x \cdot g(x)$, i.e. for polynomial complexities, the degree may increase by 1 at most. For example, consider switching from $\mathtt{mln}_\mathfrak{A}$ to $\mathtt{mlnall}$. Let g be the higher-order complexity bound. The inequation becomes $\mathtt{mlnall}((s\ d)) \leq 3 \cdot (\mathtt{size}(s\ d) + 2) \cdot (\mathtt{mln}_\mathfrak{A}((s\ d)) + 1)) = 3 \cdot (3 + \mathtt{size}(s) + \mathtt{size}(d)) \cdot (\mathtt{mln}_\mathfrak{A}((s\ d)) + 1)) \leq 3 \cdot (3 + \mathtt{size}(s) + \mathtt{size}(d)) \cdot (k_1(g(\mathtt{size}(d))))$. We see that the asymptotical upper bound is $a_1 + a_2 \cdot \mathtt{size}(d) \cdot g(\mathtt{size}(d))$. Thus for higher-order functions, the complexity may be increased by one order if the complexity is polynomial.

Using the estimations in Theorem 5.24, we can show that the complexity also of higher-order functions is not changed under switching between $\mathtt{mln}$, $\mathtt{mlnall}$, and $\mathtt{mlnlook}$, under some natural conditions on the syntactic argument structure:

Lemma 5.32. *Let s be a machine expression in lcpgc-normal form (see Definition 5.22), and $\mu, \mu' \in \{\mathtt{mln}_\mathfrak{A}(\cdot), \mathtt{mlnall}(\cdot), \mathtt{mlnlook}(\cdot)\}$, and d be an lcpgc-normal form. Then there is a linear real-valued function h, s.t. $\mu(s\ d) \leq h(\mu'(s\ d))$.*

$$\tau \in \mathit{Typ} := a \mid (\tau_1 \to \tau_2) \mid K\ \tau_1 \ldots \tau_{ar(K)}$$
$$\rho \in \mathit{PTyp} := \tau \mid \boldsymbol{\lambda} a.\rho$$
$$u \in \mathit{PExpr}_F := \Lambda a_1.\ldots.\Lambda a_k.\lambda x :: \tau.s,\ k \geq 0$$
$$r,s,t \in \mathit{Expr} := u \mid x :: \rho \mid (s\ \tau) \mid (s\ t) \mid (c_{K,i} :: \tau\ s_1\ \ldots\ s_{ar(c_{K,i})}) \mid (\texttt{seq}\ s\ t)$$
$$\mid (\texttt{letrec}\ x_1 :: \rho_1 = s_1, \ldots, x_n :: \rho_n = s_n\ \texttt{in}\ t)$$
$$\mid (\texttt{case}_K\ s\ (c_{K,1} :: \tau_1\ x_{1,1} :: \tau_{1,1} \ldots x_{1,ar(c_{K,1})} :: \tau_{1,ar(c_{K,1})} \texttt{->} t_1)\ \ \ldots$$
$$(c_{K,|D_K|} :: \tau_{|D_K|}\ x_{|D_K|,1} :: \tau_{|D_K|,1} \ldots x_{|D_K|,ar(c_{K,|D_K|})} :: \tau_{|D_K|,ar(c_{K,i})} \texttt{->} t_{|D_K|}))$$

Fig. 12: Types $\tau \in \mathit{Typ}$, polymorphic types $\rho \in \mathit{PTyp}$, polymorphic abstractions $u \in \mathit{PExpr}_F$, and expressions $r,s,t \in \mathit{Expr}$ of the language LRP where $x, x_i \in \mathit{Var}$ are term variables and $a, a_i \in \mathit{TVar}$ are type variables.

$$\frac{s :: \tau_2}{(\lambda x :: \tau_1.s) :: \tau_1 \to \tau_2} \quad \frac{s :: \rho}{\Lambda a.s :: \boldsymbol{\lambda} a.\rho} \quad \frac{s :: \boldsymbol{\lambda} a.\rho}{(s\ \tau) :: \rho[\tau/a]} \quad \frac{s :: \tau_1 \to \tau_2 \quad t :: \tau_1}{(s\ t) :: \tau_2} \quad \frac{s :: \tau \quad t :: \tau'}{(\texttt{seq}\ s\ t) :: \tau'}$$

$$\frac{s :: \tau_1 \quad pat_i :: \tau_1 \quad t_i :: \tau_2}{(\texttt{case}_K\ s\ (pat_1 \texttt{->} t_1) \ldots (pat_{|D_K|} \texttt{->} t_{|D_K|})) :: \tau_2} \quad \frac{s_1 :: \rho_1 \quad \ldots \quad s_n :: \rho_n \quad t :: \rho}{(\texttt{letrec}\ x_1 :: \rho_1 = s_1, \ldots, x_n :: \rho_n = s_n\ \texttt{in}\ t) :: \rho}$$

$$\frac{s_1 :: \tau_1, \ldots, s_{ar(c)} :: \tau_{ar(c)} \quad \tau = \tau_1 \to \ldots \to \tau_{ar(c)} \to \tau_{ar(c)+1}}{type(c) = \boldsymbol{\lambda} a_1, \ldots, a_m.\tau'' \quad \text{there are } \tau'_1, \ldots, \tau'_m \text{ with } \tau''[\tau'_1/a_1, \ldots, \tau'_m/a_m] = \tau}{(c :: \tau\ s_1 \ldots s_{ar(c)}) :: \tau_{ar(c)+1}}$$

Fig. 13: Typing Rules for LRP

Proof. Since $(s\ d)$ is also in lcpgc-normal form, this follows from Proposition 5.24.

Theorem 5.33. *Let s be a closed machine expression in lcpgc-normal form, and let P a predicate for arguments, and assume that there exists a complexity-bound according to Definition 5.31. Then the complexity-bounds of s are the same for the measures $\mathtt{mln}_{\mathfrak{A}}(\cdot), \mathtt{mlnall}(\cdot), \mathtt{mlnlook}(\cdot)$.*

Proof. Let $\mu, \mu' \in \{\mathtt{mln}_{\mathfrak{A}}(\cdot), \mathtt{mlnall}(\cdot), \mathtt{mlnlook}(\cdot)\}$. Lemma 5.32 shows that there is a linear real-valued function h s.t. $\mu(s\ d) \leq h(\mu'(s\ d))$. Let g be a complexity upper bound of s w.r.t. μ' and P. Then there is a linear real-valued function k with $\mu'(s\ d) \leq k(g(\mathtt{size}(d)))$. Combining both inequations yields $\mu(s\ d) \leq h(k(g(\mathtt{size}(d))))$. Since $h \circ k$ is a linear real-valued function, this shows that g is an upper complexity bound of s w.r.t. μ and P. Since μ, μ' where chosen freely, the theorem holds.

6   Adding Polymorphic Typing

In this section we consider the polymorphically typed variant LRP of the calculus LR, and look for improvement in the typed setting. Considering the typed case is motivated by the fact, that there are equivalences and improvements which hold in LRP, but do not hold in LR: For the test in the definition of contextual equivalence and in the improvement relation a smaller set of contexts is taken into account, since only those contexts need to be considered which leave the expressions well-typed. Since LRP is a core language of (pure) Haskell [9] our results are applicable there. As we show, our results and techniques on improvements can straightforwardly be transferred from LR to LRP.

The extensions of LRP w.r.t. LR are type annotations at variables and constructors, and extra language components, e.g. types as arguments, including a type reduction. This will be in system-F-style and restricted to let-polymorphism [3,13,12,25]. See also [19] for an analysis of typed LR of simulations as a tool for correctness. The syntax of the calculus LRP is defined in Fig. 12, where every data constructor $c \in D_K$ has a polymorphic type $type(c)$ of the form $\boldsymbol{\lambda} a_1, \ldots a_k.\tau_1 \to \ldots \tau_{ar(c)} \to K(a_1, \ldots, a_k)$. We assume the distinct variable convention for term and type variables. The typing rules are in Fig. 13. All expressions of a polymorphic type $\boldsymbol{\lambda} a.\rho$ are of the form $x :: \rho$, $\Lambda.e$, $(e\ \tau)$, and $(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ e)$, other forms are not possible. A *polymorphic abstraction* is an expression of the form $\Lambda a_1, \ldots, a_k.\lambda x.e$, and *values* are abstraction, polymorphic abstractions, and constructor applications.
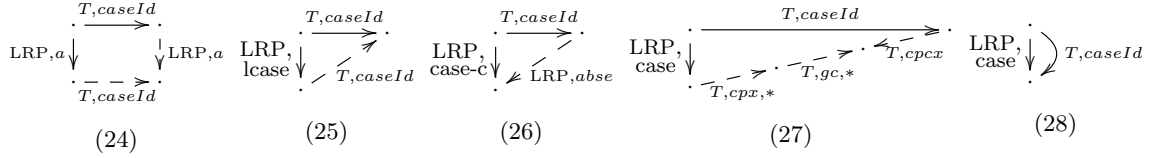
Fig. 14: Diagrams for (caseId)

Definition 6.1. *The reduction rules of the calculus* LRP *are:*

$$(Tbeta) \quad ((\Lambda a.u)^{\mathsf{sub}} \ \tau) \to u[\tau/a]$$

and all other rules from LR *(Fig. 3), extended by types and the extended syntax such that: every variable is labeled with a type, and fresh variables in rules are labeled with a type, which is derived in the rules (case-in) and (case-e) from the types of the t_i such that the types in the binding $x_i = t_i$ are equal, and in rule (cp) also polymorphic abstractions can be copied.*

The labeling algorithm is the same as for LR (see Fig. 2) where type applications are treated like usual applications. If the labeling algorithm terminates without `Fail`, then either a normal order redex is found, which is a superterm of the `sub`-marked subexpression, or the evaluation is already finished (a WHNF). Reduction contexts, weak reduction contexts, surface and top contexts are as for LR, extended by typing.

Definition 6.2 (Normal Order Reduction in LRP**).** *Let t be an expression. Then a single normal order reduction step $\xrightarrow{\text{LRP}}$ is defined by first applying the labeling algorithm to t, and if the labeling algorithm terminates successfully, then one of the rules in Definition 6.1 has to be applied, if possible, where the labels* sub, vis *must match the labels in the expression t.*

Definition 6.3. *A weak head normal form (WHNF) in* LRP *is a value, or an expression of the form* (letrec *Env* in v)*, where v is a value, or of the form* (letrec $x_1 = (c \ \overrightarrow{t})$, $\{x_i = x_{i-1}\}_{i=2}^m$, *Env* in x_m)*.*

An LRP*-expression s converges, denoted as $s\downarrow$, iff there exists a WHNF t such that $s \xrightarrow{\text{LRP},*} t$. Let s, t be two* LRP*-expressions of the same type ρ. Then s and t are* contextually equivalent *(denoted by $s \sim_c t$), iff for all contexts $C[\cdot :: \rho]$: $C[s]\downarrow \iff C[t]\downarrow$.*

Contextual equivalence satisfies the type substitution properties of logical relations (see e.g. [13]), i.e.: If $s \sim_c t$ with $s, t :: \rho$, then $s[\tau'/a] \sim_c t[\tau'/a]$ for $s, t :: \rho[\tau/a]$, and if $s, t :: \boldsymbol{\lambda} a.\rho$ with $s \sim_c t$, then $(s \ \tau) \sim_c (t \ \tau)$.

Definition 6.4. *The type erasure function $\epsilon :$ LRP $\to$ LR maps* LRP*-expressions to* LR*-expressions by removing the types, the type information and the Λ-construct. In particular: $\varepsilon(s \ \tau) = \varepsilon(s)$, $\varepsilon(\Lambda a.s) = \varepsilon(s)$, $\varepsilon(x :: \rho) = x$, and $\varepsilon(c :: \rho) = c$.*

Clearly, $\xrightarrow{\text{LRP}}$-reductions are mapped by ε to LR-normal-order reductions where exactly the (Tbeta)-reductions are omitted. The translation ε is not fully abstract, but adequate:

Proposition 6.5. *The translation ε is adequate (i.e. $\varepsilon(e_1) \sim_c \varepsilon(e_2) \implies e_1 \sim_c e_2$) and resource-preserving.*

The measure for estimating the time consumption of computation also in LRP is $\text{rln}_A(t)$ for $\emptyset \neq A \subseteq \mathfrak{A} = \{lbeta, case, seq\}$. We do not count (TBeta)-reductions.

Definition 6.6. *Let s, t be two* LRP*-expressions of the same type ρ. We define the improvement relation $\preceq_A$ for* LRP*: Let $s \preceq_A t$ iff $s \sim_c t$ and for all contexts $C[\cdot :: \rho]$: if $C[s], C[t]$ are closed, then $\text{rln}_A(C[s]) \leq \text{rln}_A(C[t])$. If $s \preceq_A t$ and $t \preceq_A s$, we write $s \approx_A t$.*

The following facts are valid and can easily be verified:

1. for closed LRP-expressions s, the equation $\mathtt{rln}_A(s) = \mathtt{rln}_A(\varepsilon(s))$ holds,
2. the reduction rules and extra transformations in their typed forms can also be used in LRP. They are correct program transformations and improvements,
3. common subexpression elimination applied to well-typed expressions is an improvement in LRP.

For $\eta \in \{\leq, =, \geq\}$ and a class of contexts X we define: For s, t of type ρ the relation $s \bowtie_{\eta, X, A} t$ (in LRP) holds iff for all X-contexts $X[\cdot : \rho]$: if $X[s], X[t]$ are closed, then $\mathtt{rln}_A(X[s]) \; \eta \; \mathtt{rln}_A(X[t])$. The context lemma for improvement also holds for LRP with almost the same proof.

Lemma 6.7 (Context Lemma for Improvement). *Let s, t be LRP-expressions of type ρ. Then $s \bowtie_{\eta, R, A} t$ (or $s \bowtie_{\eta, S, A} t$ or $s \bowtie_{\eta, T, A} t$) implies $s \bowtie_{\eta, C, A} t$.*

We end this section by showing that the following transformation (caseId) is an improvement in LRP:

$$\text{(caseId)} \qquad (\mathtt{case}_K \; s \; (pat_1 \text{ -> } pat_1) \dots (pat_{|D_K|} \text{ -> } pat_{|D_K|})) \to s$$

The rule (caseId) is the heart also of other type-dependent transformations, and it is only correct under typing, i.e. in LRP, but not in LR, which can be seen by trying the case $s = \lambda x.t$.

Lemma 6.8. *Let $s \xrightarrow{T, caseId} t$. If s is a WHNF, then t is a WHNF. If t is a WHNF, then $s \xrightarrow{LRP, lll, *} \xrightarrow{LRP, case, 0 \vee 1} \xrightarrow{LRP, lll, *} s'$ where s' is a WHNF.*

Lemma 6.9. *If $s\downarrow \wedge s \xrightarrow{T, caseId} t$, then $t\downarrow$ and $\mathtt{rln}_A(s) \geq \mathtt{rln}_A(t)$.*

Proof. Let $s \xrightarrow{T, caseId} t$ and $s \xrightarrow{LRP, k} s'$ where s' is a WHNF. We use induction on k. For $k = 0$ Lemma 6.8 shows the claim. For the induction step, let $s \xrightarrow{LRP} s_1$. The diagrams in Fig. 14 describe all cases how the fork $s_1 \xleftarrow{LRP} s \xrightarrow{T, caseId}$ can be closed. For diagram (24) we apply the induction hypothesis to $s_1 \xrightarrow{T, caseId} t_1$ which shows $t_1\downarrow$, $\mathtt{rln}_A(s_1) \geq \mathtt{rln}_A(t_1)$ and thus also $t\downarrow$ and $\mathtt{rln}_A(s) \geq \mathtt{rln}_A(t)$. For diagram (25) the induction hypothesis shows the claim. For diagram (26) we have $t\downarrow$, since (abse) is correct. Moreover, $t \xrightarrow{T, abse} s'$ is equivalent to $s' \xrightarrow{T, ucp \vee gc, *} t$ and Theorem 3.5 (4) and (6) show $\mathtt{rln}_A(s') = \mathtt{rln}_A(t)$. Thus also $\mathtt{rln}_A(s) \geq \mathtt{rln}_A(t)$. For diagram (27) we have $t\downarrow$, since (cpcx),(gc), and (cpx) are correct. Theorem 3.5 shows that $\mathtt{rln}_A(s) \geq \mathtt{rln}_A(s') = \mathtt{rln}_A(t)$, since (cpcx),(cpx) and (gc) do not change the measure $\mathtt{rln}_A(\cdot)$. For diagram(28) the claim obviously holds.

Theorem 6.10. *(caseId) is an improvement.*

Proof. Lemma 6.8 and the diagrams in Fig. 14 can be used to show (by induction on the sequence for t) that if $s \xrightarrow{T, caseId} t$ and $t\downarrow$, then $s\downarrow$, since the used existentially quantified transformations are correct and diagram 25 can only be applied finitely often. Then the context lemma for $\sim_c$ (which states that convergence preservation and reflection in reduction contexts suffices to show $\sim_c$, see e.g. [18]) and Lemma 6.9 show that (caseId) is correct. Finally, the context lemma for improvement (Lemma 6.7) and Lemma 6.9 show that (caseId) is an improvement.

7   Conclusion

We have proved that in the call-by-need functional core language LR, common subexpression elimination is an improvement, which appears to be a novel and useful result, and proves a conjecture in [10]. Since counting in [10] is based on an abstract machine, and our counting on a subset of the reduction rules, we analyzed the differences and proved that these are not substantial.

Future work could investigate other resources like space in call-by-need calculi (see [4]).

References

1. Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Funct. Programming*, 7(3):265–301, 1997.
2. Richard Bird. *Thinking functionally with Haskell*. Cambridge University Press, Cambridge, UK, 2014.
3. J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, 1994.
4. Jörgen Gustavsson and David Sands. Possibilities and limitations of call-by-need space improvement. In Benjamin C. Pierce, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001.*, pages 265–276, 2001.
5. Jennifer Hackett and Graham Hutton. Worker/wrapper/makes it/faster. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 95–107. ACM, 2014.
6. John Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989.
7. Patricia Johann and Janis Voigtländer. The impact of seq on free theorems-based program transformations. *Fundamenta Informaticae*, 69(1–2):63–102, 2006.
8. Arne Kutzner and Manfred Schmidt-Schauß. A nondeterministic call-by-need lambda calculus. In *International Conference on Functional Programming 1998*, pages 324–335. ACM Press, 1998.
9. Simon Marlow, editor. *Haskell 2010 – Language Report*. 2010. `www.haskell.org`.
10. 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.
11. Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4+5):393–434, July 2002.
12. Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
13. Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Math. Structures Comput. Sci.*, 10:321–359, 2000.
14. David Sabel and Manfred Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Math. Structures Comput. Sci.*, 18(03):501–553, 2008.
15. David Sabel and Manfred Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In Peter Schneider-Kamp and Michael Hanus, editors, *PPDP 2011*, pages 101–112, New York, NY, USA, 2011. ACM.
16. David Sands. *improvement theory and its applications*, pages 275–306. Cambridge university press, 1998.
17. M. Schmidt-Schauß, E. Machkasova, and D. Sabel. Extending Abramsky's lazy lambda calculus: (non)-conservativity of embeddings. In Femke van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications (RTA 2013)*, volume 21 of *LIPIcs*, pages 239–254, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
18. Manfred Schmidt-Schauß and David Sabel. On generic context lemmas for higher-order calculi with sharing. *Theoret. Comput. Sci.*, 411(11-13):1521 – 1541, 2010.
19. Manfred Schmidt-Schauß and David Sabel. Contextual equivalences in call-by-need and call-by-name polymorphically typed calculi (preliminary report). In *First International Workshop on Rewriting Techniques for Program Transformations and Evaluation, WPTE 2014, July 13, 2014, Vienna, Austria*, volume 40 of *OASICS*, pages 63–74. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014.
20. Manfred Schmidt-Schauß and David Sabel. Improvements in a functional core language with call-by-need operational semantics. Frank report 55, Institut für Informatik, Goethe-Universität Frankfurt am Main, March 2015. `http://www.ki.informatik.uni-frankfurt.de/papers/frank/`.
21. Manfred Schmidt-Schauß, David Sabel, and Elena Machkasova. Simulation in the call-by-need lambda-calculus with letrec, case, constructors, and seq. *Log. Methods Comput. Sci.*, 11(1), 2015.
22. Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.
23. Neil Sculthorpe, Andrew Farmer, and Andy Gill. The HERMIT in the tree: Mechanizing program transformations in the GHC core language. In *Proceedings of the 24th Symposium on Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 86–103, 2013.
24. P. Sestoft. Deriving a lazy abstract machine. *J. Funct. Programming*, 7(3):231–264, 1997.
25. Dimitrios Vytiniotis and Simon Peyton Jones. Evidence Normalization in System FC (Invited Talk). In Femke van Raamsdonk, editor, *24th RTA 2013*, volume 21 of *LIPIcs*, pages 20–38, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

A   Proof of Theorem 3.8

The following diagrams cover all cases of overlappings between normal order reduction and an (iS, cp)-transformation where iS means that the closure of (cp) in surface contexts, but excluding (no, cp) reductions. The diagrams are obtained from Lemmas B.8 and B.9 of the appendix of [22][6].

$$
\begin{array}{ccc}
\cdot & \xrightarrow{iS,cp} & \cdot \\
{\scriptstyle no,a}\downarrow & & \downarrow{\scriptstyle no,a} \\
\cdot & \xrightarrow[no\lor iS,cp]{} & \cdot
\end{array}
\qquad\qquad
\begin{array}{ccc}
\cdot & \xrightarrow{iS,cp} & \cdot \\
{\scriptstyle no,a}\downarrow & \swarrow{\scriptstyle no,a} & \\
\cdot & &
\end{array}
$$

$$a \in \{(lbeta), (case), (seq), (lll)\} \quad a \in \{(lbeta), (case), (seq)\}$$

$$(29) \qquad\qquad\qquad (30)$$

$$
\begin{array}{ccc}
\cdot & \xrightarrow{iS,cp} & \cdot \\
{\scriptstyle no,cp}\downarrow & & \downarrow{\scriptstyle no,cp} \\
\cdot & & \cdot \\
{\scriptstyle no,a}\downarrow & & \downarrow{\scriptstyle no,a} \\
\cdot & \xrightarrow[no\lor iS,cp]{} \cdot \xrightarrow[iS,cp]{} & \cdot
\end{array}
$$

$$a \in \{(lbeta), (seq)\}$$

$$(31)$$

We will use these diagrams to prove Theorem 3.8 which is repeated here:

Theorem A.1. *Let t be a closed* LR-*expression with $t \downarrow t_0$, $\emptyset \neq A \subseteq \mathfrak{A}$.*
If $t \xrightarrow{C,\ cp} t'$ then $\mathtt{rln}_A(t) = \mathtt{rln}_A(t')$.

Proof. We use the context lemma 3.7 for improvement for the relation $\approx$, i.e., we show $(cp) \subseteq \bowtie_{=,S}$ to derive $\bowtie_{=,C} = \approx$. Let s be closed and $s \xrightarrow{S,cp} s'$. We already know that $s \sim_c s'$, hence we can assume that $s \downarrow$, which implies $s' \downarrow$. We can also assume that the reduction is not normal order since in this the claim is trivial.

We prove $\mathtt{rln}_A(s) = \mathtt{rln}_A(s')$ by induction on $\mathtt{rln}_{\mathfrak{A}}(s)$ and then on the length of a normal order reduction. If the length is 0, then s is a WHNF, and hence s' is a WHNF.

If $s \xrightarrow{a} s_1$ for $a \in \{(lbeta), (case), (seq)\}$, then $\mathtt{rln}_{\mathfrak{A}}(s_1) = \mathtt{rln}_{\mathfrak{A}}(s) - 1$ and $\mathtt{rln}_A(s_1) = \mathtt{rln}_A(s)$ (if $a \notin A$) and $\mathtt{rln}_A(s_1) = \mathtt{rln}_A(s) - 1$ (if $a \in A$) . Either diagram (29) or (30) holds. In the former case we can apply the induction hypothesis, and in the latter case the claim obviously holds.

If $s \xrightarrow{no,cp} s_1$, then there are two cases: s_1 is a WHNF. In this case it is easy to see that there is a WHNF s_2 with $s' \xrightarrow{no,cp} s_2$, and the claim holds. The other case is that diagram (31). Then $s_1 \xrightarrow{no,a} s_2$ and $\mathtt{rln}_{\mathfrak{A}}(s_2) = \mathtt{rln}_{\mathfrak{A}}(s) - 1$. Hence we can apply the induction hypothesis twice, and obtain the claim.

If $s \xrightarrow{no,lll} s_1$, then diagram (29) applies, and we can apply the induction hypothesis, we have $s' \xrightarrow{no,lll} s_1$, and since $\mathtt{rln}_A(s') = \mathtt{rln}_A(s_1)$, we obtain $\mathtt{rln}_A(s) = \mathtt{rln}_A(s')$.

B   On the number of $\mathtt{rlnall}$-reductions in LR

We show by a counter example that the identity-translation from LR with $\mathtt{rlnall}$ into LR with $\mathtt{rln}_{\mathfrak{A}}$ is not resource-preserving.

We first prove a lemma which shows that the number of (lapp)-reductions can be quadratic in the number of applications, while the number of (lbeta)-, (case)-, and (seq)-reductions is linear:

Let us write id_i as an abbreviation for the expression $\lambda x_i.x_i$.

Lemma B.1. *For an environment Env and a number $n \geq 1$, let $s = \mathtt{letrec}\ Env\ \mathtt{in}\ (id_1\ \ldots\ id_n)$.*
Then the equality $\mathtt{rlnall}(s) = \dfrac{n \cdot (n+3) - 4}{2}$ and $\mathtt{rln}_{\mathfrak{A}}(s) = n - 1$ holds.

---

[6] we do not distinguish between (cpd)- and (cpt)-transformations as in [22] and simply write (cp)

Proof. By induction on n: For $n = 1$, the expression is a WHNF, and thus $\mathtt{rln}_{\mathfrak{A}}(s) = \mathtt{rlnall}(s) = 0$. For $n = 2$ the normal order reduction is as follows:

$$\mathtt{letrec}\ Env\ \mathtt{in}\ (id_1\ id_2)$$
$$\xrightarrow{no,lbeta}\ \mathtt{letrec}\ Env\ \mathtt{in}\ \mathtt{letrec}\ x_1 = id_2\ \mathtt{in}\ x_1$$
$$\xrightarrow{no,llet}\ \mathtt{letrec}\ Env, x_1 = id_2\ \mathtt{in}\ x_1$$
$$\xrightarrow{no,cp}\ \mathtt{letrec}\ Env, x_1 = id_2\ \mathtt{in}\ id_2$$

and thus $\mathtt{rln}_{\mathfrak{A}}(s) = 1$ and $\mathtt{rlnall}(s) = 3$.

For the induction step, let $n \geq 3$. We consider the normal order reduction of s:

$$\mathtt{letrec}\ Env\ \mathtt{in}\ id_1\ id_2 \ldots id_n$$
$$\xrightarrow{no,lbeta}\ \mathtt{letrec}\ Env\ \mathtt{in}\ ((\mathtt{letrec}\ x_1 = id_2\ \mathtt{in}\ x_1)\ id_3\ \ldots id_n)$$
$$\xrightarrow{no,lapp,n-2}\ \mathtt{letrec}\ Env\ \mathtt{in}\ (\mathtt{letrec}\ x_1 = id_2\ \mathtt{in}\ (x_1\ id_3\ \ldots id_n))$$
$$\xrightarrow{no,llet}\ \mathtt{letrec}\ Env, x_1 = id_2\ \mathtt{in}\ (x_1\ id_3\ \ldots id_n)$$
$$\xrightarrow{no,cp}\ \mathtt{letrec}\ Env, x_1 = id_2\ \mathtt{in}\ (id_2\ id_3\ \ldots id_n)$$

By the induction hypothesis, we have

$$\mathtt{rlnall}(s) = (n+1) + \frac{(n-1) \cdot (n+2) - 4}{2} = \frac{n \cdot (n+3) - 4}{2}$$

and $\mathtt{rln}_{\mathfrak{A}}(s) = n - 2 + 1 = n - 1$.

However, the previous lemma is not sufficient to disprove resource-preservation, since the size of the input-expression is $c * n$. Thus, in the remainder of the section we show, that we can generate the input expression s (from Lemma B.1) with $n = c * 2^m$ from an expression of size $d * m$ (where $c, d > 0$ are constants).

Let us assume that Peano-numbers are available with constructors S and Z and let us write $(S^n\ Z)$ for the n-th peano number.

First consider the expression

$$s_{2^m} := \mathtt{letrec}\ n = (S^{2^m}\ Z), f = F\ \mathtt{in}\ f\ n$$

where $F := \lambda x.(\mathtt{case}\ x\ \mathtt{of}\ ((S\ y)\text{->}f\ y\ id)\ (Z\text{->}id))$

Then $s_{2^m} \xrightarrow{no,*} \mathtt{letrec}\ Env\ \mathtt{in}\ (id_1\ \ldots\ id_{2^m+1})$ where the number of (no,case)- and (no,lbeta)-reductions is $2 * (2^m + 1)$. However, for constructing the counter-example the representation of the Peano number is insufficient, since the size of s_{2^m} is $O(2^m)$ which is too large. Hence, we use a shared representation of the Peano representation of 2^m which replaces the binding for n in the expression s_{2^m}, and thus let

$$t_{2^m} := \mathtt{letrec}\quad x_{2^0} = \lambda h.S\ h, \{x_{2^i} = \lambda h.x_{2^{i-1}}(x_{2^{i-1}}h)\}_{i=1}^m,$$
$$h_0 = Z, h_{2^m} = x_{2^m}\ h_0,$$
$$f = F$$
$$\mathtt{in}\ f\ h_{2^m}$$

One can verify that h_{2^m} indeed represents 2^m as a Peano number, and that evaluating h_{2^m} results in a binding $h_{2^m} = S\ h_{2^m-1}$, and iteratively evaluating $h_{2^m-1}, h_{2^m-2}, \ldots h_1$ (which f does) results in an expression

$$\mathtt{letrec}\ h_0 = Z, \{h_i = S\ h_{i-1}\}_{i=1}^{2^m},\ Env\ \mathtt{in}\ (id_1\ \ldots\ id_{2^m+1}).$$

Clearly, for counting (lbeta)- and (case)-reductions, the reductions for evaluating the 2^m calls to f are still $2 * (2^m + 1)$, but there are additional (lbeta)-reductions for decompressing the number which we will count in the following.

During generation and evaluation of the bindings for h_i, they are of one of the following forms (ignoring some intermediate forms, which are removed by (lll)-reductions):

1. $h_i = S\ h_{i-1}$, or
2. $h_i = x_{2^j}\ h_k$, where $k + 2^j = i$, or
3. $h_i = (x_{2^j}\ (x_{2^j}\ h_k))$, where $k + 2 * 2^j = i$.

We analyze the connection (in the reduction) between these three possible forms, where we will also add some (gc)- and (cpx)-transformations, which does not break our counting, since both transformations do not change the $\mathtt{rln}_{\mathfrak{A}}(\cdot)$ measure.

1. For case (1) the binding h_i is successfully evaluated and no more (lbeta)-reductions are necessary for this binding.
2. For case (2) we consider two subcases.
   (a) Assume that $j > 0$. Then the evaluation is

$$
\begin{aligned}
& h_i = (x_{2^j}\ h_k) \\
\xrightarrow{cp}\quad & h_i = (\lambda h.x_{2^{j-1}}\ (x_{2^{j-1}}\ h))\ h_k \\
\xrightarrow{lbeta,llet}\quad & h_i = x_{2^{j-1}}\ (x_{2^{j-1}}\ h), h = h_k \\
\xrightarrow{cpx,gc}\quad & h_i = x_{2^{j-1}}\ (x_{2^{j-1}}\ h_k).
\end{aligned}
$$

   Thus with one (lbeta) step, we derive a binding of type (3)
   (b) Assume that $j = 0$. Then the evaluation is

$$
\begin{aligned}
& h_i = (x_{2^0}\ h_{i-1}) \\
\xrightarrow{cp}\quad & h_i = (\lambda h.S\ h)\ h_{i-1} \\
\xrightarrow{lbeta,llet}\quad & h_i = S\ h, h = h_{i-1} \\
\xrightarrow{cpx,gc}\quad & h_i = S\ h_{i-1}.
\end{aligned}
$$

   Thus with one (lbeta) step, we derive a binding of type (1).
3. For case (3), the reduction is

$$
\begin{aligned}
& h_i = (x_{2^j}\ (x_{2^j}\ h_k)) \\
\xrightarrow{cp}\quad & h_i = (\lambda h_{2^j+k}.(x_{2^{j-1}}\ (x_{2^{j-1}}\ h_{2^j+k}))\ (x_{2^j}\ h_k)) \\
\xrightarrow{lbeta,llet}\quad & h_i = (x_{2^{j-1}}\ (x_{2^{j-1}}\ h_{2^j+k})), h_{2^j+k} = (x_{2^j}\ h_k).
\end{aligned}
$$

   Thus with one (lbeta) step we a derive a binding of type (3) and additionally generate a binding of type (2).

The following ideas help to prove that the generation of h_i is unique. First note that generating bindings of type (1) terminates, since the indices get smaller in every step. For uniqueness, which means that every h_i is generated only once, the following invariant can be used in an induction proof: Let H be the environment consisting of all the bindings of the three forms. Let $g(b)$ for bindings b of type (1) be 1, for $b = \{h_i = x_{2^j}\ h_k\}$ of type (2) let $g(b) = 2^j$, and for $b = \{h_i = (x_{2^j}\ (x_{2^j}\ h_k))\}$ of type (3), let $g(b) = 2^{j+1}$. Let $g(H) = \sum_{b \in H} g(b)$. Then the rules for type (2) and (3) remove one binding and add 1 or 2, but leave the sum invariant. Hence, by induction, and since the start we have $g(H) = 2^m$, exactly 2^m bindings are created. It is also easy to see that every number will be generated.

Now we calculate the sum of the (lbeta)-reductions: Case (1) does not require (lbeta)-reductions, case (3) can only occur $2^m - 1$ times (since there are no more generated h_i-bindings), case (2a) can also only occur $2^m - 1$ times (since it results in case (3)), and case (2b) can occur 2^m times (i.e. once for each binding h_i). This results in $3 * 2^m - 2$ (lbeta)-reductions for decompressing the Peano-number. Summing up the essential reductions for decompressing the Peano-number, for unfolding the definition of f, and for evaluating $\mathtt{letrec}\ Env\ \mathtt{in}\ (id_1\ \ldots\ id_{2^m+1})$ (Lemma B.1) yields $\mathtt{rln}_{\mathfrak{A}}(t_{2^m}) = 3 * 2^m - 2 + 2 * (2^m + 1) + ((2^m + 1) - 1) = 6 * 2^m$.

However, by Lemma B.1 $\mathtt{rlnall}(t_{2^m}) \geq c*2^{2*m} = c*(2^m)^2$ for some constant c, and $\mathtt{size}(t_{2^m}) = d* m$ for some constant d. Since for all positive integers k, $(2^m)^2$ is asymptotically larger than m^k*2^m, the translation from LR with measure $\mathtt{rln}_{\mathfrak{A}}(\cdot)$ into LR with measure $\mathtt{rlnall}(\cdot)$ is not resource-preserving.

C Diagrams for pcgE and pcg

C.1 Diagrams for pcgE

(a) Typical cases for diagram (5)

$$\begin{array}{l}\texttt{letrec } x = (\texttt{letrec } E \texttt{ in } r), E_1 \\ \qquad x' = (\texttt{letrec } E' \texttt{ in } r'), E_1' \ E_2 \\ \texttt{in } R^-[x]\end{array} \xrightarrow{\quad T,pcgE \quad} \begin{array}{l}\texttt{letrec } \ x = (\texttt{letrec } E \texttt{ in } r), E_1, E_2\alpha \\ \texttt{in } R^-[x]\alpha\end{array}$$
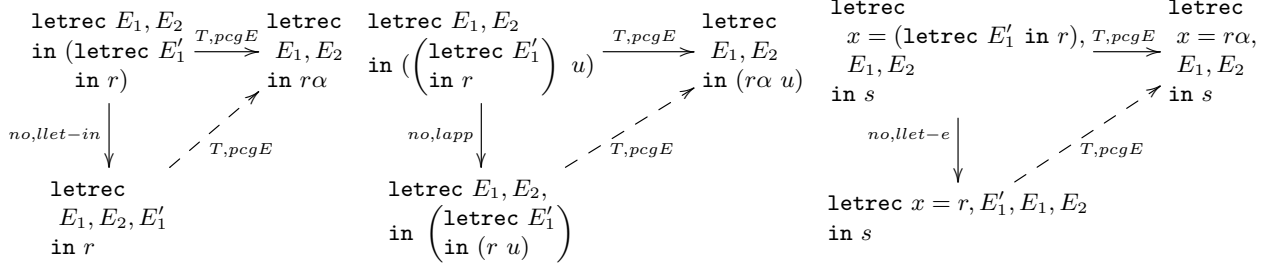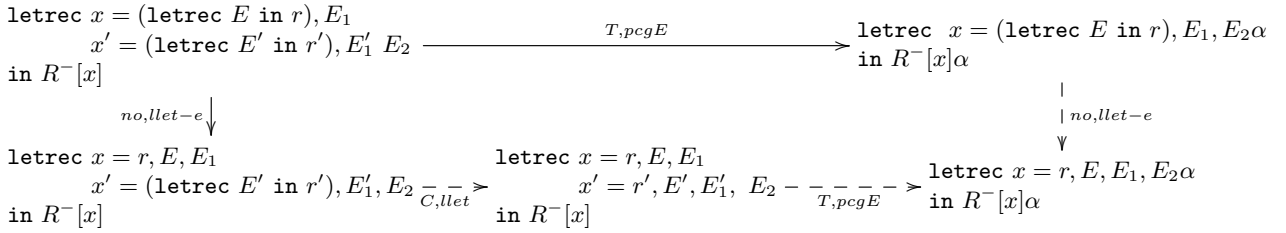
(b) Typical case for diagram (6)

(c) Typical case for diagram (7)

$$R[\texttt{seq } (c \ (\texttt{letrec } E_1, E_1' \texttt{ in } r)) \ s] \xrightarrow{T,pcgE} R[\texttt{seq } (c \ (\texttt{letrec } E_1 \texttt{ in } r\alpha)) \ s]$$
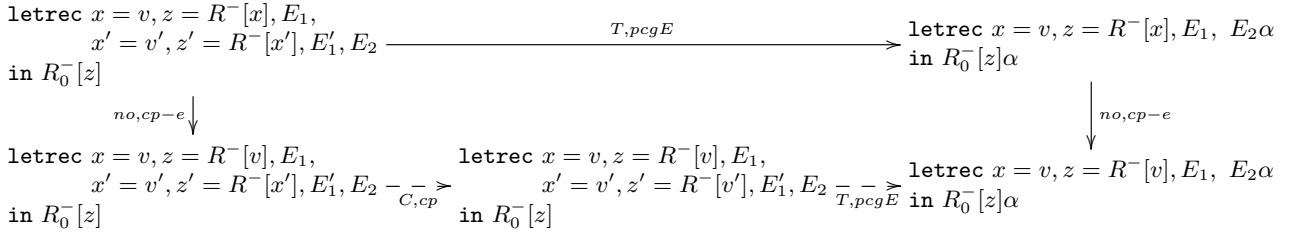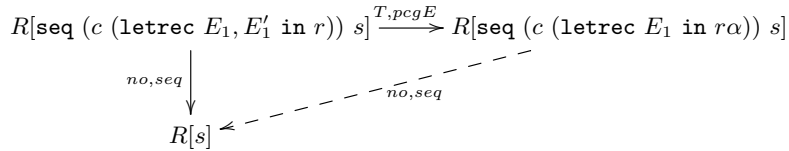
(d) Typical case for diagram (8)

Fig. 15: Typical cases for the diagrams for (pcgE)

Inspecting all overlappings of a normal order reduction step and a (pcgE)-transformation shows that all overlappings between a normal order reduction step and a (T,pcgE)-transformation can be closed by one of the diagrams shown in Figs. 7.

Diagram (4) describes the case of a non-critical overlap where the steps can be commuted. Diagram (5) covers the case, where a `letrec`-expression which is part of an (no,lll)-redex is removed by (pcgE), i.e. three typical cases are shown in Fig. 15a

Diagram(6) covers the cases where a binding environment is removed which includes a `letrec`-expression which is a duplicate of a letrec-expression that is part of a (no,llet)-redex is removed (pcgE). A typical case is is in Fig. 15b.

Diagram (7) covers the case where the normal order reduction modifies parts of a `letrec`-environment which is a duplicate used by the (pcgE)-transformation. A typical case is in Fig. 15c.

$$\texttt{letrec } x = (y\ u),\quad\qquad\qquad\qquad\quad \xrightarrow{\ \ T,pcgE\ \ }\quad \texttt{letrec } x = (y\ u), y = \lambda z.s, E$$
$$y = \lambda z.(\texttt{letrec } E \texttt{ in } s), E \qquad\qquad\qquad\qquad\qquad\quad \texttt{in } x$$
$$\texttt{in } x$$

no,cp $\Big\downarrow$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ no,cp $\Big\downarrow$

$$\texttt{letrec } x = (\lambda z.(\texttt{letrec } E \texttt{ in } s)\ u) \qquad\qquad\quad \texttt{letrec } x = (\lambda z.s\ u),$$
$$y = \lambda z.(\texttt{letrec } E \texttt{ in } s), E\ \ \dashrightarrow_{\ T,pcgE,T,pcgE}\ \ \ y = \lambda z.s, E$$
$$\texttt{in } x \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{in } x$$

(a) Typical case for diagram (9)

$$\texttt{letrec } x = c\ s_1\ s_2,$$
$$x' = c\ s_1\ s_2 \qquad\qquad\qquad\qquad \xrightarrow{\ \ \ T,pcgE\ \ \ } \qquad\qquad \texttt{letrec } x = c\ s_1\ s_2,$$
$$\texttt{in case } x \texttt{ of } c\ y_1\ y_2 \texttt{->} t \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{in case } x \texttt{ of } c\ y_1\ y_2 \texttt{->} t$$

$no,case$ $\Big\downarrow$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $no,case$ $\Big\downarrow$

$$\texttt{letrec } x = c\ w_1\ w_2, w_1 = s_1, w_2 = s_2, \qquad \texttt{letrec } x = c\ w_1\ w_2, w_1 = s_1, w_2 = s_2,$$
$$x' = c\ s_1\ s_2 \qquad\qquad \dashrightarrow_{\ T,abs}\qquad x' = c\ w'_1\ w'_2, w'_1 = s_1, w'_2 = s_2 \dashrightarrow_{\ T,cpgE} \quad \texttt{letrec } x = c\ w_1\ w_2, w_1 = s_1, w_2 = s_2,$$
$$\texttt{in letrec } y_1 = w_1, y_2 = w_2 \texttt{ in } t \qquad \texttt{in letrec } y_1 = w_1, y_2 = w_2 \texttt{ in } t \qquad\qquad \texttt{in letrec } y_1 = w_1, y_2 = w_2 \texttt{ in } t$$

(b) Typical case for diagram (10)

$$\begin{array}{l}\texttt{letrec}\\ \quad x = \texttt{case } y \texttt{ of}\\ \qquad c\ z_1\ z_2 \to t,\\ \quad x' = \texttt{case } y \texttt{ of}\\ \qquad c\ z'_1\ z'_2 \to t',\\ \quad y = c\ s_1\ s_2\\ \texttt{in } x\end{array} \qquad \xrightarrow{\ \ T,pcgE\ \ } \qquad \begin{array}{l}\texttt{letrec}\\ \quad x = \texttt{case } y \texttt{ of}\\ \qquad c\ z_1\ z_2 \to t,\\ \quad y = c\ s_1\ s_2\\ \texttt{in } x\end{array}$$

$no,case$ $\Big\downarrow$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $no,case$ $\Big\downarrow$

$$\begin{array}{l}\texttt{letrec}\\ \quad x = \texttt{letrec}\\ \qquad z_1 = w_1,\\ \qquad z_2 = w_2\\ \quad\ \texttt{in } t,\\ \quad x' = \texttt{case } y \texttt{ of}\\ \qquad c\ z'_1\ z'_2 \to t',\\ \quad y = c\ w_1\ w_2,\\ \quad w_1 = s_1,\\ \quad w_2 = s_2\\ \texttt{in } x\end{array} \dashrightarrow_{C,case} \begin{array}{l}\texttt{letrec}\\ \quad x = \texttt{letrec}\\ \qquad z_1 = w_1,\\ \qquad z_2 = w_2\\ \quad\ \texttt{in } t,\\ \quad x' = \texttt{letrec}\\ \qquad z'_1 = w'_1,\\ \qquad z'_2 = w'_2\\ \quad\ \texttt{in } t',\\ \quad y = c\ w'_1\ w'_2,\\ \quad w'_1 = w_1,\\ \quad w'_2 = w_2,\\ \quad w_1 = s_1,\\ \quad w_2 = s_2\\ \texttt{in } x\end{array} \dashrightarrow_{C,cpx,*,C,gc,*} \begin{array}{l}\texttt{letrec}\\ \quad x = \texttt{letrec}\\ \qquad z_1 = w_1,\\ \qquad z_2 = w_2\\ \quad\ \texttt{in } t,\\ \quad x' = \texttt{letrec}\\ \qquad z'_1 = w_1,\\ \qquad z'_2 = w_2\\ \quad\ \texttt{in } t',\\ \quad y = c\ w_1\ w_2,\\ \quad w_1 = s_1,\\ \quad w_2 = s_2\\ \texttt{in } x\end{array} \dashrightarrow_{T,cpgE} \begin{array}{l}\texttt{letrec}\\ \quad x = \texttt{letrec}\\ \qquad z_1 = w_1,\\ \qquad z_2 = w_2\\ \quad\ \texttt{in } t,\\ \quad y = c\ w_1\ w_2,\\ \quad w_1 = s_1,\\ \quad w_2 = s_2\\ \texttt{in } x\end{array}$$

(c) Typical case for diagram (11)

Fig. 16: Typical cases for the diagrams for (pcgE), cont'd.

Diagram (8) covers the case where the (pcgE)-redex is removed by the normal order reduction, an example is given in Fig. 15d. Diagram (9) covers the case where the (pcgE)-redex is copied by the normal order reduction, an example is given in Fig. 16a.

The seventh diagram (10) covers the case where the environment shared by (pcgE) contains a constructor application used by (no,case) reduction. An example is given in Fig. 16b.

The seventh diagram (11) covers the case where the redex of a (no,case) is shared by (pcgE). An example is given in Fig. 16c.

C.2   Diagrams for pcg

$$\texttt{letrec } z = r'', x = \lambda x'.r'' \texttt{ in } (x\ u) \xrightarrow{\quad T,pcg \quad} \texttt{letrec } z = r'', x = \lambda x'.z \texttt{ in } (x\ u)$$

$$\downarrow_{no,cp} \qquad\qquad\qquad \downarrow_{no,cp}$$

$$\begin{array}{l}\texttt{letrec } z = r'', x = \lambda x'.r'' \\ \texttt{in } ((\lambda x'.r'')\ u)\end{array} \qquad\qquad \begin{array}{l}\texttt{letrec } z = r'', x = \lambda x'.z \\ \texttt{in } ((\lambda x'.z)\ u)\end{array}$$

$$\downarrow_{no,lbeta} \qquad\qquad\qquad \downarrow_{no,lbeta}$$

$$\begin{array}{l}\texttt{letrec } z = r'', x = \lambda x'.r'' \\ \texttt{in } (\texttt{letrec } x' = u \texttt{ in } r'')\end{array} \dashrightarrow_{T,pcg} \begin{array}{l}\texttt{letrec } z = r'', x = \lambda x'.z \\ \texttt{in } (\texttt{letrec } x' = u \texttt{ in } r'')\end{array} \dashrightarrow_{T,pcg} \begin{array}{l}\texttt{letrec } z = r'', x = \lambda x'.z \\ \texttt{in } (\texttt{letrec } x' = u \texttt{ in } z)\end{array}$$

(a) Typical case for diagram (13)

$$\texttt{letrec } z = r'', x = \lambda x'.r'' \texttt{ in } (\texttt{seq } x\ u) \xrightarrow{\quad T,pcg \quad} \texttt{letrec } z = r'', x = \lambda x'.z \texttt{ in } (\texttt{seq } x\ u)$$

$$\downarrow_{no,cp} \qquad\qquad\qquad \downarrow_{no,cp}$$

$$\begin{array}{l}\texttt{letrec } z = r'', x = \lambda x'.r'' \\ \texttt{in } (\texttt{seq } (\lambda x'.r'')\ u)\end{array} \qquad\qquad \begin{array}{l}\texttt{letrec } z = r'', x = \lambda x'.z \\ \texttt{in } (\texttt{seq } (\lambda x'.z)\ u)\end{array}$$

$$\downarrow_{no,seq} \qquad\qquad\qquad \downarrow_{no,seq}$$

$$\begin{array}{l}\texttt{letrec } z = r'', x = \lambda x'.r'' \\ \texttt{in } u\end{array} \dashrightarrow_{T,pcg} \begin{array}{l}\texttt{letrec } z = r'', x = \lambda x'.z \\ \texttt{in } u\end{array} =\!=\!=\!=\!=\!= \begin{array}{l}\texttt{letrec } z = r'', x = \lambda x'.z \\ \texttt{in } u\end{array}$$

(b) Typical case for diagram (13) with seq

$$\begin{array}{l}\texttt{letrec } z = x\ u, x = \lambda x'.r \\ \texttt{in } (x\ u)\end{array} \xrightarrow{\quad T,pcg \quad} \texttt{letrec } z = x\ u, x = \lambda x'.r \texttt{ in } z$$

$$\downarrow_{no,cp} \qquad\qquad\qquad \downarrow_{no,cp}$$

$$\begin{array}{l}\texttt{letrec } z = x\ u, x = \lambda x'.r \\ \texttt{in } ((\lambda x'.r)\ u)\end{array} \qquad\qquad \begin{array}{l}\texttt{letrec } z = ((\lambda x'.r)\ u), x = \lambda x'.r \\ \texttt{in } z\end{array}$$

$$\downarrow_{no,lbeta} \qquad\qquad\qquad \downarrow_{no,lbeta}$$

$$\begin{array}{l}\texttt{letrec } z = x\ u, x = \lambda x'.r \\ \texttt{in } (\texttt{letrec } x' = u \texttt{ in } r)\end{array} \underset{C,cp;C,lbeta}{\dashrightarrow} \begin{array}{c}\texttt{letrec } z = (\texttt{letrec } x' = u \texttt{ in } r), \\ x = \lambda x'.r \\ \texttt{in } (\texttt{letrec } x' = u \texttt{ in } r)\end{array} \underset{T,pcg}{\dashrightarrow} \begin{array}{c}\texttt{letrec } z = (\texttt{letrec } x' = u \texttt{ in } r), \\ x = \lambda x'.r \\ \texttt{in } z\end{array}$$

(c) Typical case for diagram (14)

$$\begin{array}{l}\texttt{letrec } z = \texttt{seq } x\ u, x = \lambda x'.r \\ \texttt{in } (\texttt{seq } x\ u)\end{array} \xrightarrow{\quad T,pcg \quad} \texttt{letrec } z = \texttt{seq } x\ u, x = \lambda x'.r \texttt{ in } z$$

$$\downarrow_{no,cp} \qquad\qquad\qquad \downarrow_{no,cp}$$

$$\begin{array}{l}\texttt{letrec } z = \texttt{seq } x\ u, x = \lambda x'.r \\ \texttt{in } (\texttt{seq } (\lambda x'.r)\ u)\end{array} \qquad\qquad \begin{array}{l}\texttt{letrec } z = (\texttt{seq } (\lambda x'.r)\ u), x = \lambda x'.r \\ \texttt{in } z\end{array}$$

$$\downarrow_{no,seq} \qquad\qquad\qquad \downarrow_{no,seq}$$

$$\begin{array}{l}\texttt{letrec } z = \texttt{seq } x\ u, x = \lambda x'.r \\ \texttt{in } u\end{array} \underset{C,cp;C,seq}{\dashrightarrow} \begin{array}{c}\texttt{letrec } z = u, \\ x = \lambda x'.r \\ \texttt{in } u\end{array} \underset{T,pcg}{\dashrightarrow} \begin{array}{c}\texttt{letrec } z = u, \\ x = \lambda x'.r \\ \texttt{in } z\end{array}$$

(d) Typical case for diagram (14) with (seq)

We inspect the overlappings between normal-order reduction steps and top-context-applications of (pcg). Some easy cases (which need not be treated by a diagram) are the following:

$$\texttt{letrec } x = (r\ u) \texttt{ in } C[(r\ u)] \xrightarrow{\quad pcg \quad} \texttt{letrec } x = (r\ u) \texttt{ in } C[x]$$

$\Big\downarrow no,lbeta \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Big\downarrow no,lbeta$

$$\begin{array}{l} \texttt{letrec } x = (r\ u) \\ \texttt{ in } C[(\texttt{letrec } y = u \texttt{ in } r')] \end{array} \xrightarrow{C,lbeta} \begin{array}{l} \texttt{letrec } x = \\ (\texttt{letrec } y = u \texttt{ in } r') \\ \texttt{in } C[(\texttt{letrec } y = u \texttt{ in } r')] \end{array} \xrightarrow{T,pcg} \begin{array}{l} \texttt{letrec } x = (\texttt{letrec } y = u \texttt{ in } r') \\ \texttt{in } C[x] \end{array}$$

$$\begin{array}{l}\texttt{letrec } x = r \\ \texttt{in } C[r] \end{array} \xrightarrow{\quad T,pcg \quad} \begin{array}{l}\texttt{letrec } x = r \\ \texttt{in } C[x] \end{array} \qquad \begin{array}{l}\texttt{letrec } x = r \\ \texttt{in } M[x,r] \end{array} \xrightarrow{\quad T,pcg \quad} \begin{array}{l}\texttt{letrec } x = r \\ \texttt{in } M[x,x] \end{array}$$

$\Big\downarrow no,a \qquad\qquad\qquad\qquad\qquad \Big\downarrow no,a \qquad \Big\downarrow no,a \qquad\qquad\qquad\qquad\qquad\qquad \Big\downarrow no,a$

$$\begin{array}{l}\texttt{letrec } x = r \\ \texttt{in } C[r'] \end{array} \xrightarrow{C,a,*} \begin{array}{l}\texttt{letrec } x = r' \\ \texttt{in } C[r'] \end{array} \xrightarrow{T,pcg} \begin{array}{l}\texttt{letrec } x = r' \\ \texttt{in } C[x] \end{array} \quad \begin{array}{l}\texttt{letrec } x = r' \\ \texttt{in } M[x,r] \end{array} \xrightarrow{C,a,*} \begin{array}{l}\texttt{letrec } x = r' \\ \texttt{in } M[x,r'] \end{array} \xrightarrow{T,pcg} \begin{array}{l}\texttt{letrec } x = r' \\ \texttt{in } M[x,x] \end{array}$$

(e) Typical cases for diagram (15)

$$R[\texttt{seq } (c\ C[r])\ s] \xrightarrow{T,pcg} R[\texttt{seq } (c\ C[x])\ s]$$

$\Big\downarrow no,seq \qquad\qquad \nearrow \quad no,seq$

$$R[s]$$

(f) Typical case is for diagram (16)

$$\begin{array}{l}\texttt{letrec} \\ \quad x = (\texttt{letrec } E_1 \texttt{ in } s_2) \\ \texttt{in } M[x,(\texttt{letrec } E_1 \texttt{ in } s_2)] \end{array} \xrightarrow{\quad T,pcg \quad} \begin{array}{l}\texttt{letrec} \\ \quad x = (\texttt{letrec } E_1 \texttt{ in } s_2) \\ \texttt{in } M[x,x] \end{array}$$

$\Big\downarrow no,llet \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Big\downarrow no,llet$

$$\begin{array}{l}\texttt{letrec} \\ \quad E_1, x = s_2 \\ \texttt{in } M[x,(\texttt{letrec } E_1 \texttt{ in } s_2)] \end{array} \xrightarrow{T,pcgE} \begin{array}{l}\texttt{letrec} \\ \quad E_1, x = s_2 \\ \texttt{in } M[x,s_2] \end{array} \xrightarrow{T,pcg} \begin{array}{l}\texttt{letrec} \\ \quad E_1, x = s_2 \\ \texttt{in } M[x,x] \end{array}$$

(g) Typical case for diagram (17)

$$\begin{array}{l}\texttt{letrec } x = (c\ s_1\ s_2) \\ \texttt{in } C[\texttt{case } (c\ s_1\ s_2) \texttt{ of} \\ \quad (c\ z_1\ z_2) \texttt{-> } r[z_1,z_2]] \end{array} \xrightarrow{\qquad\qquad T,pcg \qquad\qquad} \begin{array}{l}\texttt{letrec } x = (c\ s_1\ s_2) \\ \texttt{in } C[\texttt{case } x \texttt{ of} \\ \quad (c\ z_1\ z_2) \texttt{-> } r[z_1,z_2]] \end{array}$$

$\Big\downarrow no,case \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Big\downarrow no,case$

$$\begin{array}{l}\texttt{letrec } x = c\ s_1\ s_2 \texttt{ in} \\ \quad C[\texttt{letrec} \\ \quad\quad y_1 = s_1, y_2 = s_2 \\ \quad\quad \texttt{in } r[y_1,y_2]] \end{array} \xrightarrow{C,lll,*} \begin{array}{l}\texttt{letrec } x = c\ s_1\ s_2, \\ \quad y_1 = s_1, y_2 = s_2 \\ \texttt{in } C[r[y_1,y_2]] \end{array} \xrightarrow{T,pcg,pcg} \begin{array}{l}\texttt{letrec } x = c\ y_1\ y_2, \\ \quad y_1 = s_1, y_2 = s_2 \\ \texttt{in } C[r[y_1,y_2]] \end{array} \xleftarrow{C,cpx,cpx,gc} \begin{array}{l}\texttt{letrec } x = c\ y_1\ y_2, \\ \quad y_1 = s_1, y_2 = s_2 \\ \texttt{in } C[\texttt{letrec } z_1 = y_1, z_2 = y_2 \\ \quad \texttt{in } r[z_1,z_2]] \end{array}$$

(h) Typical case for diagram (18)

$$\begin{array}{l}\texttt{letrec } x = \texttt{case } y \texttt{ of} \\ \quad (c\ z_1\ z_2) \texttt{-> } r[z_1,z_2], \\ \quad y = (c\ s_1\ s_2) \\ \texttt{in } R^-[\texttt{case } y \texttt{ of} \\ \quad (c\ z_1\ z_2) \texttt{-> } r[z_1,z_2]] \end{array} \xrightarrow{\qquad\qquad T,pcg \qquad\qquad} \begin{array}{l}\texttt{letrec } x = \texttt{case } y \texttt{ of} \\ \quad (c\ z_1\ z_2) \texttt{-> } r[z_1,z_2], \\ \quad y = (c\ s_1\ s_2) \\ \texttt{in } R^-[x] \end{array}$$

$\Big\downarrow no,case \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Big\downarrow no,case$

$$\begin{array}{l}\texttt{letrec } x = \texttt{case } y \texttt{ of} \\ \quad (c\ z_1\ z_2) \texttt{-> } r[z_1,z_2], \\ \quad y = (c\ u_1\ u_2), \\ \quad u_1 = s_1, u_2 = s_2 \\ \texttt{in } R^-[\texttt{letrec } z_1 = u_1, z_2 = u_2 \\ \texttt{in } r[z_1,z_2]] \end{array} \xrightarrow{S,case} \begin{array}{l}\texttt{letrec } x = (\texttt{letrec} \\ \quad z_1 = u_1', z_2 = u_2' \\ \quad \texttt{in } r[z_1,z_2]) \\ \quad y = (c\ u_1'\ u_2'), \\ \quad u_1' = u_1, u_2' = u_2, \\ \quad u_1 = s_1, u_2 = s_2 \\ \texttt{in } R^-[\texttt{letrec} \\ \quad z_1 = u_1, z_2 = u_2 \\ \texttt{in } r[z_1,z_2]] \end{array} \xrightarrow{T,cpx,*,gc,*} \begin{array}{l}\texttt{letrec } x = (\texttt{letrec} \\ \quad z_1 = u_1, z_2 = u_2 \\ \quad \texttt{in } r[z_1,z_2]) \\ \quad y = (c\ u_1\ u_2), \\ \quad u_1 = s_1, u_2 = s_2 \\ \texttt{in } R^-[\texttt{letrec} \\ \quad z_1 = u_1, z_2 = u_2 \\ \texttt{in } r[z_1,z_2]] \end{array} \xrightarrow{T,pcg} \begin{array}{l}\texttt{letrec } x = \\ \quad (\texttt{letrec } z_1 = u_1, z_2 = u_2 \\ \quad\quad \texttt{in } r[z_1,z_2]), \\ \quad u_1 = s_1, u_2 = s_2, \\ \quad x = (c\ u_1\ u_2) \\ \texttt{in } R^-[x] \end{array}$$

(i) Typical case for diagram (19)

Fig. 17: Typical cases for the diagrams for (pcg), cont'd.

- The (T,pcg)-transformation is also an inverse (C,cp) or an inverse (C,cpx) transformation. Examples are $\texttt{letrec } x = \lambda y.y \texttt{ in } (\lambda y'.y') (\lambda z.z) \xrightarrow{T,pcg} \texttt{letrec } x = \lambda y.y \texttt{ in } x (\lambda z.z)$ and $\texttt{letrec } x = \lambda y.y, x = y, z = y \texttt{ in } r \xrightarrow{T,pcg} \texttt{letrec } x = \lambda y.y, x = y, z = x \texttt{ in } r$.
- The normal order reduction step is a (no,cp)-reduction and leads to a WHNF, e.g. $\texttt{letrec } x = \lambda y.y, z = \lambda y.y \texttt{ in } x \xrightarrow{no,cp} \texttt{letrec } x = \lambda y.y, z = \lambda y.y \texttt{ in } \lambda y.y$

For the remaining cases at least one of the diagrams shown in Figs. 8 is applicable.

We explain the diagrams in Fig. 8 and gives exemplary instances of the diagrams:

Diagram (12) describes the case where the reductions can be commuted. Diagrams (13) and (14) cover the cases where a (no,cp)-reduction is followed by a (no,lbeta)-reduction, and the shared expression is inside the copied expression. Typical cases for the second and third diagram are in Figs. 17a and 17c.

Diagram (15) describes the cases where the normal order reduction modifies the subexpression which occurs twice and is shared by the (cpg)-transformation. Three prototypical expressions and overlappings for diagram (15) are show in Fig. 17e

Diagram(16) covers the case, that the duplicated expression is inside the first argument of $\texttt{seq}$ or in an unused alternative of a $\texttt{case}$-expression. A typical case is given in Fig. 17f.

Diagram (17) covers the case that the duplicated subexpression is a $\texttt{letrec}$-expression which is deconstructed by an $(no,llet)$-reduction. A typical case is given in Fig. 17g. Diagram (18) covers the case that the scrutinee of a $\texttt{case}$-expression is one of the duplicated expressions. A typical case is given in Fig. 17h.

D   Further Estimations

Lemma D.1. *Let s be a closed expression with $s\downarrow$. Then $\texttt{rlnall}(s)$ is in $\mathcal{O}(\texttt{rln}_{\mathfrak{A}}(s)^2 * \texttt{size}(s))$*

Proof. Suppose $s \xrightarrow{no,*} s_n$ where s_n is a WHNF, and let s_i be the intermediate expressions.
For every (cp)-reduction in this sequence exactly one of the following statements holds:

- The (cp)-reduction is the last reduction in the sequence.
- The (cp)-reduction is followed by a (seq)-reduction (which removes the copied abstraction)
- The (cp)-reduction is followed by an (lbeta)-reduction (which uses the copied abstraction in function position)

Let us conflate these reductions, i.e. we write (cpseq) for a (cp)-reduction followed by (seq), (cplbeta) for a (cp)-reduction followed by (lbeta), and (cpWHNF) for a (cp)-reduction as a last reduction in a normal order reduction to a WHNF.

Let $\texttt{size}_{LR}$ be the size of an expression, where we do not count the $x = y$-bindings, i.e. $\texttt{size}_{LR}(s) = \texttt{size}(s) - indirections(s)$, where $indirections(s)$ is the number of $\texttt{letrec}$-bindings $x = y$ in s where y is a variable.

Then all the reduction rules of LR with the exception of $(cplbeta)$ and $(cpWHNF)$ do not increase the $\texttt{size}_{LR}$, and the reduction rules $(case)$, (seq), and $(cpseq)$ strictly decrease $\texttt{size}_{LR}$. The (llet)-rules strictly decrease $\texttt{size}_{LR}$.
Let $sumdepth_{LR}$, be the sum of the depth of all letrec-symbols. The (lapp)-, (lcase)-, and (lseq)-reductions do not change $\texttt{size}_{LR}$, but they decrease $sumdepth_{LR}$, The only rules that increases this size measure is: (cplbeta) by $\texttt{size}_{LR}(s_i) * \texttt{size}_{LR}(s)$.

We make a rough estimation: $\texttt{size}_{LR}(s_n) \le (\texttt{rln}_{\mathfrak{A}}(s) + 1) * (\texttt{size}(s) + 1)$. Thus $sumdepth_{LR}(s_i) \le (\texttt{rln}_{\mathfrak{A}}(s) + 1)^2 * (\texttt{size}(s) + 1)$.

Now we are ready to compute an upper bound for $\texttt{rlnall}(s)$ using $\texttt{rln}_{\mathfrak{A}}$: there are at most $(\texttt{rln}_{\mathfrak{A}}(s) + 1)$ (cp)-reductions, at most $(\texttt{rln}_{\mathfrak{A}}(s) + 1) * (\texttt{size}(s) + 1)$ (llet)-reductions, and at most $(\texttt{rln}_{\mathfrak{A}}(s) + 1)^2 * (\texttt{size}(s) + 1)$ (lapp)-, (lcase)-, and (lseq)-reductions. Hence $\texttt{rlnall}(s) \le \texttt{rln}_{\mathfrak{A}}(s) + (\texttt{rln}_{\mathfrak{A}}(s) + 1) + (\texttt{rln}_{\mathfrak{A}}(s) + 1) * (\texttt{size}(s) + 1) + (\texttt{rln}_{\mathfrak{A}}(s) + 1)^2 * (\texttt{size}(s) + 1)$ Hence $\texttt{rlnall}(s)$ is in $\mathcal{O}(\texttt{rln}_{\mathfrak{A}}(s)^2 * \texttt{size}(s))$ $\qquad\square$

E Properties of the Polymorphic Type System

Some properties are described and proved and the expressive power is analyzed.

First some remarks

- The convention that bound type variables are different, is a necessary requirement that the type system works correct. Consider the expression

$$\Lambda a.\lambda x{::}a.\texttt{letrec } z{::}\boldsymbol{\lambda}b.b \to b = (\Lambda a.\lambda x :: a.\texttt{case True of } (\texttt{True -> } x); (\texttt{True -> } y)) \texttt{ in } z$$

This expression passes the type check, since variables x, y have the same type. However, a renaming of the type variables will produce

$$\Lambda a.\lambda x{::}a.\texttt{letrec } z{::}\boldsymbol{\lambda}b.b \to b = (\Lambda a'.\lambda x :: a'.\texttt{case True of } (\texttt{True -> } x); (\texttt{True -> } y)) \texttt{ in } z$$

which does not pass the type check, since now x, y have different types. Clearly both expressions are not correct. Thus, for the type check it is necessary that bound type variables are all different.
- The syntactic type system is monomorphic, which means that every subexpression has exactly one type (up to variable renaming).
- There is a bot-expression of every polymorphic type: The expressions is $\texttt{letrec } x{::}\rho = x \texttt{ in } x$. This expressions is permitted due to our syntax, its type is also ρ, a and it does not normal-order reduce. Note that this expression has a polymorphic type without a Λ. Let us make this more concrete. Consider $(\texttt{letrec } x{::}\boldsymbol{\lambda}a.a = x \texttt{ in } x) \texttt{ Bool}$. Then this expression permits a normal order reduction step resulting in $(\texttt{letrec } x{::}\boldsymbol{\lambda}a.a = x \texttt{ in } (x \texttt{ Bool}))$; and then there is no further normal order reduction. There is no possibility to reduce it to $(\texttt{letrec } x{::}\texttt{Bool} = x \texttt{ in } x$, since there is no Λ. However, the program translation $(\texttt{letrec } x{::}\boldsymbol{\lambda}a.\rho = C[(x \texttt{ Bool})] \texttt{ in } C'[(x \texttt{ Bool})] \rightarrow (\texttt{letrec } x{::}\rho[\texttt{Bool}/a] = C[x] \texttt{ in } C'[x]$ appears to be correct.

Proposition E.1. *Normal-order reduction in* LRP *does not produce type errors. More rigorously: if* $s :: \rho$, *and* $s \xrightarrow{\text{LRP}} s'$, *then* $s' :: \rho'$. *This also holds for the main reduction rules in any context, and for the extra reduction rules.*

Proof. The proof for the first part consists of checking for all reduction rules that the type remains unchanged (up to renaming of type variables). We make this explicit only for a selection of the reduction rules.

- The (Tbeta) rule is $((\Lambda a.u) \ \tau) \to u[\tau/a]$. If the type of u is ρ, then the type of $((\Lambda a.u) \ \tau)$ is $\rho[\tau/a]$. The type computation of $u[\tau/a]$ will also compute the type $\rho[\tau/a]$, provided the distinct variable convention holds for the expression.
- The (llet-e)-rule is $\texttt{letrec } Env_1, x = (\texttt{letrec } Env_2 \texttt{ in } t)^{\mathsf{sub}} \texttt{ in } r \to \texttt{letrec } Env_1, Env_2, x = t \texttt{ in } r$. Since there is type scoping in the binding $(\texttt{letrec } Env_2 \texttt{ in } t)$, the computed types for both expressions are identical.

Checking all the extra transformation rules is also an easy scan. Only two issues have to be noted: Freshly introduced variable must have a type such that the new expression remains well-typed; and the expressions before the transformation must satisfy the distinct variable convention for expression variables and for type variables.

The translation $\varepsilon : \text{LRP} \to \text{LR}$ removes all types and type information. We also use it for reduction sequences, where only the reduction (Tbeta) is eliminated.

We show relation between the two notions of normal-order reductions:

Proposition E.2. *Let* s, s' *be* LRP-*expression.*

1. *If* $s \xrightarrow{\text{LRP},a} s'$, *then either* $\varepsilon(s) = \varepsilon(s')$ *if* $a = (Tbeta)$ *or* $s \xrightarrow{\text{LR},a} s'$ *for all other reductions* a.

2. If $\varepsilon(s) \xrightarrow{\text{LR},a} \varepsilon(s')$, *then* $s \xrightarrow{\text{LRP},\textit{Tbeta},*} \xrightarrow{\text{LRP},a} \xrightarrow{\text{LRP},\textit{Tbeta},*} s'$.

Proof. Part (1) is immediate, since the mapping ε leads to the same reduction if $a \neq (\textit{Tbeta})$, and in the case $a = (\textit{Tbeta})$, the LR-expressions are the same. Note that the restriction to polymorphic abstractions is required for this argument, since then the copied polymorphic abstractions in (cp) are mapped to abstractions.

For Part (2) the syntactic restrictions have to be checked. Since abstractions are monomorphic, it is not possible to construct an expressions $((\lambda x.s)\ \tau)$, and since $\varLambda a.s$ requires s to be an abstraction or a polymorphic abstraction bindings $x = \lambda x.s$ in LR can only stem from $x = r$, where in case there is no normal-order (Tbeta)-reduction possible for r, the expressions r is a polymorphic abstraction. Hence it can also be copied. We have to check the expressions of the form $(r\ \tau)$, if these cannot be further normal-order reduced. r can only be a polymorphic expression of a ρ-type: If it is a $\varLambda a.r'$, or a letrec then a reduction is possible. If it is a variable, then a cp-reduction has to be done before it can be reduced. It can also not be a seq-expression, a case-expression, an application where the right expression is not a type, nor a constructor expression. Hence we have checked all cases. Thus the claimed form of the LRP-reduction is valid.

Proposition E.3. *Let s be an* LRP*-expression. Then* $s \downarrow_{\text{LRP}} \iff \varepsilon(s) \downarrow_{\text{LR}}$

Proof. This follows from the previous proposition and the fact that the WHNFs correspond.