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

Research group for Artificial Intelligence and Software Technology
Institut für Informatik,
Fachbereich Informatik und Mathematik,
Johann Wolfgang Goethe-Universität,
Postfach 11 19 32, D-60054 Frankfurt, Germany

March 28, 2015

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 [8] support declarative programming. They allow a definition of the intended results leaving the exact sequence of operations unspecified and provide a high-level of abstraction [5].

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.

Moreover, call-by-name models the computation of results and can be used to reason about program semantics, but it fails to model resource consumption in real implementations. In contrast call-by-need program calculi provide a good model of both: the correctness of the computation as well as the amount of required work. 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 is a lot of research in the area of analyzing and proving the correctness of program transformations (e.g. [10, 6, 20]). However, there seems to be few research on whether the (correct) program transformations are also optimizations – i.e. while preserving the meaning of the programs they also decrease the runtime or the space behavior of the programs. Having such results is e.g. useful in automated tools for program transformation like Hermit [21].

A theory of optimizations or improvements in extended lambda calculi is treated in [9] for a call-by-need higher order language, and a call-by-value variant in [15]. In [9] the resource model counts the steps of an abstract machine for call-by-need evaluation which is a variant of Sestoft's abstract machine [22]. The work of Moran and Sands [9] provides a foundation for program improvements which leads to several results exhibiting program transformations that are improvements and also provides

Term variables: $x, x_i \in Var$ where Var is the set of term variables

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

(a) Expressions of the language LR

$(s\ t)^{\mathsf{sub}\vee\mathsf{top}}$	$\to (s^{\mathsf{sub}}\ t)^{\mathsf{vis}}$
$(\texttt{letrec}\ Env\ \texttt{in}\ s)^{\mathsf{top}}$	$\to (\texttt{letrec}\ Env\ \texttt{in}\ s^{\mathsf{sub}})^{\mathsf{vis}}$
$(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C[x^{\mathsf{sub}}])$	$\to (\texttt{letrec}\ x = s^{\mathsf{sub}}, Env\ \texttt{in}\ C[x^{\mathsf{vis}}])$
$(\texttt{letrec}\ x = s, y = C[x^{\mathsf{sub}}], Env\ \texttt{in}\ t)$	$\to (\texttt{letrec}\ x = s^{\mathsf{sub}}, y = C[x^{\mathsf{vis}}], Env\ \texttt{in}\ t)$, where C is not trivial
$(\texttt{letrec}\ x = s, y = x^{\mathsf{sub}}, Env\ \texttt{in}\ t)$	$\to (\texttt{letrec}\ x = s^{\mathsf{sub}}, y = x^{\mathsf{nontarg}}, Env\ \texttt{in}\ t)$
$(\texttt{seq}\ s\ t)^{\mathsf{sub}\vee\mathsf{top}}$	$\to (\texttt{seq}\ s^{\mathsf{sub}}\ t)^{\mathsf{vis}}$
$(\texttt{case}\ s\ \texttt{of}\ alts)^{\mathsf{sub}\vee\mathsf{top}}$	$\to (\texttt{case}\ s^{\mathsf{sub}}\ \texttt{of}\ alts)^{\mathsf{vis}}$
$\texttt{letrec}\ x = s^{\mathsf{vis}\vee\mathsf{nontarg}}, y = C[x^{\mathsf{sub}}], Env\ \texttt{in}\ t$	$\to \texttt{Fail}$
$(\texttt{letrec}\ x = C[x^{\mathsf{sub}}], Env\ \texttt{in}\ s)$	$\to \texttt{Fail}$

(b) Computing reduction positions using labels in LR, where $a \vee b$ means label a or label b. The algorithm does not overwrite non-displayed labels.

Fig. 1: Syntax and Labeling for the Calculus LR

techniques for showing program transformations being improvements. In [9] 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 [3] 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 [9], but not proved to be an improvement (but it is conjectured).

Recently, Hackett and Hutton [4] rediscovered the improvement theory of [9] to argue that optimizations are indeed improvements, with a particular focus on worker/wrapper transformations (see e.g. [1] for more examples). The work of [4] uses the same call-by-need abstract machine as [9] 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 [20], an extended higher-order lambda calculus with call-by-need evaluation which models the core language of Haskell. 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. [6, 16]).

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 built upon a detailed analysis of reduction lengths (performed in [20] 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 [7, 20, 13, 14] to show correctness of program transformations; and correctness of inlining (or common subexpression elimination) via infinite expressions (unfolding the letrecs) established in [19]. We prefer analyzing reductions in LR, due to the success of the diagram method in LR.

Since our improvement relation is different from [9] (it uses a different measure and operational semantics), we compare our measures with those in [9, 4]. The result is that our improvement theory can be transferred to the abstract machine of [9] 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, typing arguments are required for showing that interesting program transformations are improvements (see e.g. [4]). Due to cyclic bindings, using monomorphic typing and monomorphising a polymorphic calculus is insufficient. Hence we adapt ideas from system-F polymorphism [2, 11], in particular from an intermediate language in a Haskell compiler [8, 23], and develop an improvement theory for the calculus LRP – a polymorphically typed variant of LR with `let`-polymorphism. The type erasure of reduction sequences exactly leads to the untyped reduction sequences in LR, so that our analyses complement each other. 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 introduce improvements for the polymorphically typed variant LRP of the calculus LR and prove that a type dependent transformation is an improvement. We conclude in Sect. 7.

2 The Call-by-Need Lambda Calculus LR

We recall the calculus LR [20], which is an untyped call-by-need lambda calculus which extends the lambda calculus by recursive `letrec`, data constructors, case-expressions, and Haskell's `seq`-operator. We also recall several results from previous investigations: from [20] we reuse a counting theorem for reduction lengths and correctness of several program transformations. From [19] we reuse correctness of copying arbitrary expressions.

We employ the syntax of the calculus LR [20]. 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 $ar(c_{K,i}) \geq 0$. The syntax of expressions $r, s, t \in Expr$ of LR is defined in Fig. 1a. 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* $(c_{K,i}\ s_1 \ldots s_{ar(c_{K,i})})$ are only allowed to occur fully saturated. We sometimes omit the index of the constructor or use vector notation and thus write e.g. $(c\ \overrightarrow{s})$ instead of $(c_{K,i}\ s_1 \ldots s_{ar(c_{K,i})})$.

In a `letrec`-*expression* $\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t$ 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 the expression t is called the `in`-*expression*. Sometimes the environment is abbreviated by *Env* (e.g. we write (`letrec` *Env* `in` s), if the exact syntax of the bindings is not relevant). In an environment $Env = \{x_1 = t_1, \ldots, x_n = t_n\}$, we define $LV(Env) = \{x_1, \ldots, x_n\}$ and we sometimes write $\{x_i = t_i\}_{i=1}^n$ as abbreviation for such an environment. We also use *Env* for parts of the environment like e.g. in $\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ s$. 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 $\{x_i = x_{i-1}\}_{i=j}^m$.

In a `seq`-*expression* $(\texttt{seq}\ s\ t)$ the expression s must be successfully evaluated before the expression t is evaluated, and thus `seq` can be used for strict evaluation of expressions. The syntax includes `case`-*expressions* $(\texttt{case}_K\ s\ \texttt{of}\ ((c_{K,1}\ x_1 \ldots x_{ar(c_{K,1})})\ \texttt{->}\ t_1)\ \ldots\ ((c_{K,|D_K|}\ x_1 \ldots x_{ar(c_{K,|D_K|})})\ \texttt{->}\ t_{|D_K|}))$ where there is a $\texttt{case}_K$ for every type constructor K. A `case`-expression has 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 $(\texttt{case}_K\ s\ \texttt{of}\ alts)$.

Definition 2.1. *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*

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

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

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

(llet-in) $(\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ r)^{\mathsf{sub}}) \to (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ r)$

(llet-e) $(\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s_x)^{\mathsf{sub}}\ \texttt{in}\ r) \to (\texttt{letrec}\ Env_1, Env_2, x = s_x\ \texttt{in}\ r)$

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 2: Reduction rules

contexts where the hole is not in an alternative of a case, and weak top contexts *are top contexts where the hole is not in a* `letrec`*.*

A multicontext M *is an expression with several (or also no) 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 the calculus are defined in Fig. 2, where the role of the labels sub, top, vis, nontarg will be explained below in Definition 2.2. 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 `letrec`-environments. The rules (lapp), (lcase), and (lseq) float-out a `letrec` from the first argument of an application, a `case`-expression, or a `seq`-expression. The rules (seq-c), (seq-in), and (seq-e) evaluate a `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 `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, which is a call-by-name strategy adapted to sharing. It applies the reduction rules at specific positions. Instead of defining the call-by-need evaluation in terms of a syntactic definition of reduction contexts (using a context free grammar), we provide an algorithm to find the position of a redex and also to describe the syntactic form of so-called reduction contexts[1].

Definition 2.2 (Labeling Algorithm). *The labeling algorithm detects the position to which a reduction rule will be applied according to normal order. It uses the labels:* top,sub,vis,nontarg *where* top

[1] However, a syntactic (and rather complex) definition of reduction contexts by a context free grammar and the corresponding description of normal order reduction for the calculus LR can be found in [20].

(gc1)	$(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n, Env\ \texttt{in}\ t) \to (\texttt{letrec}\ Env\ \texttt{in}\ t)$ if for all $i : x_i$ does not occur in Env nor in t
(gc2)	$(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t) \to t$ if for all $i : x_i$ does not occur in t
(cpx-in)	$(\texttt{letrec}\ x = y, Env\ \texttt{in}\ C[x]) \to (\texttt{letrec}\ x = y, Env\ \texttt{in}\ C[y])$ where y is a variable and $x \neq y$
(cpx-e)	$(\texttt{letrec}\ x = y, z = C[x], Env\ \texttt{in}\ t) \to (\texttt{letrec}\ x = y, z = C[y], Env\ \texttt{in}\ t)$ where y is a variable and $x \neq y$
(cpax)	$(\texttt{letrec}\ x = y, Env\ \texttt{in}\ s) \to (\texttt{letrec}\ x = y, Env[y/x]\ \texttt{in}\ s[y/x])$ where y is a variable, $x \neq y$ and $y \in FV(s, Env)$
(cpcx-in)	$(\texttt{letrec}\ x = c\ \overrightarrow{t}, Env\ \texttt{in}\ C[x]) \to (\texttt{letrec}\ x = c\ \overrightarrow{y}, y_1 = t_1, \ldots, y_n = t_{ar(c)}, Env\ \texttt{in}\ C[c\ \overrightarrow{y}])$
(cpcx-e)	$(\texttt{letrec}\ x = c\ \overrightarrow{t}, z = C[x], Env\ \texttt{in}\ t) \to (\texttt{letrec}\ x = c\ \overrightarrow{y}, y_1 = t_1, \ldots, y_{ar(c)} = t_{ar(c)}, z = C[c\ \overrightarrow{y}], Env\ \texttt{in}\ t)$
(abs)	$(\texttt{letrec}\ x = c\ \overrightarrow{t}, Env\ \texttt{in}\ s) \to \texttt{letrec}\ x = c\ \overrightarrow{x}, x_1 = t_1, \ldots, x_{ar(c)} = t_{ar(c)}, Env\ \texttt{in}\ s$ where $ar(c) \geq 1$
(abse)	$(c\ \overrightarrow{t}) \to (\texttt{letrec}\ x_1 = t_1, \ldots, x_{ar(c)} = t_{ar(c)}\ \texttt{in}\ c\ \overrightarrow{x})$ 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. 3: Extra transformation rules

means reduction of the top term, sub *means reduction of a subterm,* vis *marks already visited subexpressions, and* nontarg *marks already 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. 1b exhaustively.*

Note that the labeling algorithm does not descend into sub-labeled `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 2.3 (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 Figure 2 has to be applied, if possible, where the labels* sub, vis *must match the labels in the expression t (t may have more labels).*

It can be verified (by a case analysis) 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' (upto 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 $\to$ we write $\xrightarrow{+}$ for the transitive closure, and $\xrightarrow{*}$ for the reflexive-transitive closure of $\to$. E.g., $\xrightarrow{no,*}$ denotes the reflexive-transitive closure of $\xrightarrow{no}$. We write $\xrightarrow{n}$ for exactly n $\to$-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 2.4. *A reduction context R is any context, such that its hole will be labeled with* sub *or* top *by the labeling algorithm in Fig. 1b. A* weak reduction context*, R^-, is a reduction context, where the hole is not within a* `letrec`*-expression.*

Definition 2.5. *A* weak head normal form (WHNF) *is a value v, or an expression $(\texttt{letrec}\ Env\ \texttt{in}\ v)$, where v is a value, or an expression $(\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ x_m)$.*

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 $s \downarrow t$*

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 $s \xrightarrow{P} t$.

The reduction rules in Fig. 2 are also program transformations where we ignore the labels.

Definition 2.6. *We define unions for the rules in Fig. 2: (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).*

In Fig. 3 additional program transformations are defined. We use the following unions: (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 reduction rules also for normal order reduction and thus e.g. write $\xrightarrow{no,llet}$ for $\xrightarrow{no,llet-in \vee no,llet-e}$.

We briefly explain the additional transformations: (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.

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.

Definition 2.7. *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 $C[\cdot]$: $C[s]{\downarrow} \iff C[t]{\downarrow}$.*

Note that contextual equivalence is a congruence, i.e. it is an equivalence relation which is compatible with contexts.

Definition 2.8. *A program transformation P is* correct*, if it preserves contextual equivalence, i.e. $P \subseteq \sim_c$.*

In [20] we proved that all introduced transformations are correct:

Proposition 2.9 ([20]). *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,t by the further 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 two measures for estimating the time consumption, counting the essential and all reduction steps:

Definition 3.1. *Let t be a closed expression with $t \downarrow t_0$.*

1. $\texttt{rln}(t)$ *is the number of lbeta, case, seq-reductions in $t \downarrow t_0$.*
2. $\texttt{rlnall}(t)$ *is the number of all reductions in $t \downarrow t_0$.*

It is consistent to define the measures as ∞*, if* $t\uparrow$.

The main measure throughout this paper is $\mathtt{rln}(\cdot)$ which 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 $2 \cdot \mathtt{rln}(t) + 1$.

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 [9]). 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 5.17).

From [20] we repeat several invariance properties w.r.t. reduction lengths of the transformations in Figs. 2 and 3:

Theorem 3.2 ([20]). *Let* t *be a closed* LR*-expression with* $t \downarrow t_0$.

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

Definition 3.3 (Improvement Relation). *For* $s, t \in Expr$ *let* $s \preceq t$ *(*t *is improved by* s*), iff* $s \sim_c t$ *and for all contexts* $C[\cdot]$ *s.t.* $C[s]$, $C[t]$ *are closed:* $\mathtt{rln}(C[s]) \leq \mathtt{rln}(C[t])$*. We write* $t \succeq s$ *if* $s \preceq t$ *holds. If* $s \preceq t$ *and* $s \succeq t$*, we write* $s \approx t$.

A program transformation P *is* an improvement *iff* $P \subseteq \succeq$.

Let $\eta \in \{\leq, =, \geq\}$ be a relation on non-negative integers, and for a class of contexts X (we will instantiate X with: all contexts C; all reduction contexts R; all surface contexts S; or all top-contexts T) let $s \bowtie_{\eta,X} t$ iff for all X-contexts X, s.t. $X[s], X[t]$ are closed: $\mathtt{rln}(X[s])\ \eta\ \mathtt{rln}(X[t])$. In particular, $\bowtie_{\leq,C} = \preceq$, $\bowtie_{\geq,C} = \succeq$, and $\bowtie_{=,C} = \approx$.

The context lemma for improvement shows that it suffices to take reduction contexts into account for proving improvement. Its proof is similar to the ones for context lemmas for contextual equivalence in call-by-need lambda calculi (see [20, 17]).

Lemma 3.4 (Context Lemma for improvement). *Let* s, t *be expressions with* $s \sim_c t$*,* $\eta \in \{\leq, =, \geq\}$*. Then* $s \bowtie_{\eta,R} t$ *iff* $s \bowtie_{\eta,C} t$.

Proof. One direction is trivial. For the other direction we prove a more general claim using multicontexts:

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* $s_i \bowtie_{\eta,R} t_i$*. Then for all multicontexts* M *with* n *holes s.t.* $M[s_1, \ldots, s_n]$ *and* $M[t_1, \ldots, t_n]$ *are closed:* $\mathtt{rln}(M[s_1, \ldots, s_n])\ \eta\ \mathtt{rln}(M[t_1, \ldots, t_n])$.

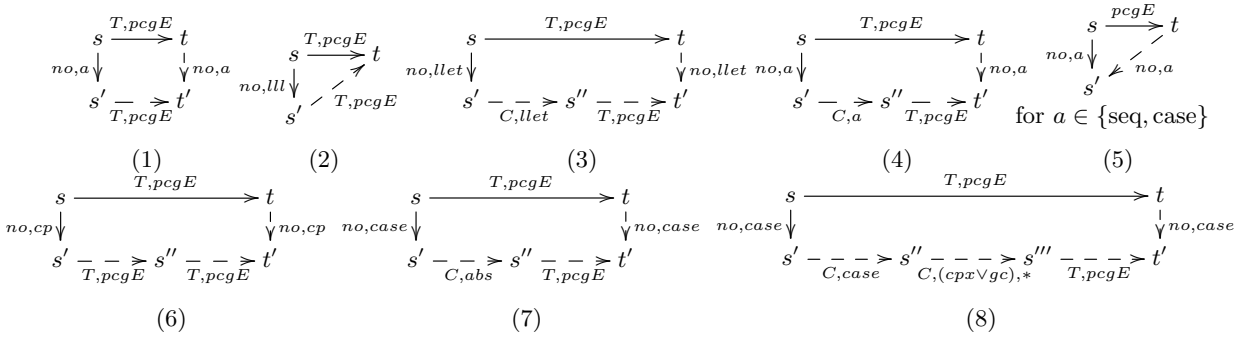


Fig. 4: Forking diagrams for (pcgE)

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, 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 $\mathtt{rln}(M[s_1, \ldots, s_n]) = 0 = \mathtt{rln}(M[t_1, \ldots, t_n])$.

If in $M[s_1, \ldots, s_n]$ one s_i 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. By the induction hypothesis, using the multi-context $M[\ldots, \cdot, s_i, \cdot, \ldots]$, we have $\mathtt{rln}(M[s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n])$ η $\mathtt{rln}(M[t_1, \ldots, t_{i-1}, s_i, t_{i+1}, \ldots, t_n])$, and from the assumption we have $\mathtt{rln}(M[t_1, \ldots, t_{i-1}, s_i, t_{i+1}, \ldots, t_n])$ η $\mathtt{rln}(M[t_1, \ldots, t_{i-1}, t_i, t_{i+1}, \ldots, t_n])$, and hence $\mathtt{rln}(M[s_1, \ldots, s_n])$ η $\mathtt{rln}(M[t_1, \ldots, t_n])$.

If in $M[s_1, \ldots, s_n]$ there is no s_i in a reduction context, then $M[s_1, \ldots, s_n] \xrightarrow{no,a} M'[s'_1, \ldots, s'_{n'}]$, 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 $\mathtt{rln}(.)$-count on both sides is $m = 0$ or $m = 1$, depending on a. Thus we can apply the induction hypothesis to $M'[s'_1, \ldots, s'_{n'}]$ and $M[t'_1, \ldots, t'_{n'}]$, and so we have $\mathtt{rln}(M[s_1, \ldots, s_n]) = m + \mathtt{rln}(M'[s'_1, \ldots, s'_{n'}])$ η $m + \mathtt{rln}(M'[t'_1, \ldots, t'_{n'}]) = \mathtt{rln}(M[t_1, \ldots, t_n])$.

Since reduction contexts are also T- or S-contexts, we have:

Corollary 3.5. *Let s, t be expressions with $s \sim_c t$, and $\eta \in \{\leq, =, \geq\}$. Then $s \bowtie_{\eta,T} t$ (or $s \bowtie_{\eta,S} t$) implies that $s \bowtie_{\eta,C} t$.*

Now we can prove properties of the (cp)-reduction using the diagrams in [20] (see also Appendix A)

Theorem 3.6. *Let t be a closed* LR*-expression with $t \downarrow t_0$. If $t \xrightarrow{C,cp} t'$ then $\mathtt{rln}(t) = \mathtt{rln}(t')$.*

Proof. This follows using correctness of (cp) and the diagrams in [20, Lemmas B.8, B.9] where forking diagrams for $t_1 \xleftarrow{no,a} s \xrightarrow{S,cp}$ are computed. Then counting the reductions contributing to $\mathtt{rln}(.)$ and using Corollary 3.5 shows the claim.

Due to the exact analyses in [20] on the influence of the reduction rules (Fig. 2) and the additional transformations (Fig. 3) concerning the reduction lengths as stated in Theorems 3.2 and 3.6, Proposition 2.9 and Corollary 3.5. imply the following theorem:

Theorem 3.7. *The transformations (case), (seq), (lbeta), (cp), (lll), (gc), (cpx), (cpax), (xch), (abs), and (ucp) are improvements.*

Moreover, for $a \in \{$(cp), (lll), (gc), (cpx), (cpax), (xch), (abs), (ucp)$\}$ the inclusion $a \subseteq \preceq$ and thus the inclusion $a \subseteq \approx$ holds.

4 Common Subexpression Elimination

Common subexpression elimination (cse) can be expressed as:

$$\begin{array}{ll}\text{(cse)} & M[s,\ldots,s] \to \texttt{letrec}\ x = s\ \texttt{in}\ M[x,\ldots,x] \\ & \quad \text{where } x \text{ is a fresh variable and the multicontext } M \\ & \quad \text{does not capture a variable in } s\end{array}$$

We will show that (cse) is an improvement. Although this appears to be obvious, it is not trivial, due to several reasons. The calculus LR is call-by-need, which means that computations and also parts of computations can be shared. Correctness of (cse) could only be proved via a call-by-name calculus on infinite trees, which cannot be used to analyse resource usage under call-by-need, and it was mentioned as a conjecture in [9] (for a related core language).

To show $(cse) \subseteq \succeq$ we consider the general-copy rule

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

We first show that the inverse of (gcp) is an improvement. Since (gc) $\subseteq \approx$ and $\texttt{letrec}\ x = s\ \texttt{in}\ M[x,\ldots,x] \xrightarrow{gcp,*} \texttt{letrec}\ x = s\ \texttt{in}\ M[s,\ldots,s] \xrightarrow{gc} M[s,\ldots,s]$, this implies that (cse) is an improvement. For the proof that (pcg) (the inverse of (gcp)) is an improvement we require several variants of the rule. Note that $x \in FV(s)$ is permitted (gcp) and (pcg).

Definition 4.1. *The transformation (pcg) is the union of the rules:*

$$\begin{array}{ll}\textit{(pcg-in)} & \texttt{letrec}\ x = s, Env\ \texttt{in}\ C[s] \\ & \quad \to\ \texttt{letrec}\ x = s, Env\ \texttt{in}\ C[x] \\ \textit{(pcg-e)} & \texttt{letrec}\ x = s, Env, y = C[s]\ \texttt{in}\ r \\ & \quad \to\ \texttt{letrec}\ x = s, Env, y = C[x]\ \texttt{in}\ r\end{array}$$

The transformation (pcgE) is the union of the following rules:

$$\begin{array}{ll}\textit{(pcgE}_1\textit{-in)} & \texttt{letrec}\ Env, Env_2\ \texttt{in}\ C[\texttt{letrec}\ Env', Env_3\ \texttt{in}\ r] \\ & \quad \to \texttt{letrec}\ Env, Env_2\ \texttt{in}\ C[\texttt{letrec}\ Env_3\alpha\ \texttt{in}\ r\alpha] \\ \textit{(pcgE}_2\textit{-in)} & \texttt{letrec}\ Env, Env_2\ \texttt{in}\ C[\texttt{letrec}\ Env'\ \texttt{in}\ r] \\ & \quad \to \texttt{letrec}\ Env, Env_2\ \texttt{in}\ C[r\alpha] \\ \textit{(pcgE}_1\textit{-e)} & \\ & \texttt{letrec}\ Env, Env_2, x = C[\texttt{letrec}\ Env', Env_3\ \texttt{in}\ r]\ \texttt{in}\ s \\ & \to \texttt{letrec}\ Env, Env_2, x = C[\texttt{letrec}\ Env_3\alpha\ \texttt{in}\ r\alpha]\ \texttt{in}\ s \\ \textit{(pcgE}_2\textit{-e)} & \texttt{letrec}\ Env, Env_2, x = C[\texttt{letrec}\ Env'\ \texttt{in}\ r] \\ & \quad \to \texttt{letrec}\ Env, Env_2, x = C[r\alpha]\ \texttt{in}\ s \\ \textit{(pcgE}_3\textit{)} & \texttt{letrec}\ Env, Env', Env_3\ \texttt{in}\ r \\ & \quad \to \texttt{letrec}\ Env, Env_3\alpha\ \texttt{in}\ r\alpha\end{array}$$

where $Env'\alpha = Env$ and α only renames variables of $LV(Env')$.

Proposition 4.2. *(pcg) and (pcgE) are correct.*

Proof. In [19] 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 two expressions s, t (where $=$ is syntactic equality modulo α-equivalence on infinite trees), then $s \sim_c t$.*

Now, since $s \xrightarrow{pcg \vee pcgE} t$ implies that $IT(s) = IT(t)$, correctness of both program transformation holds.

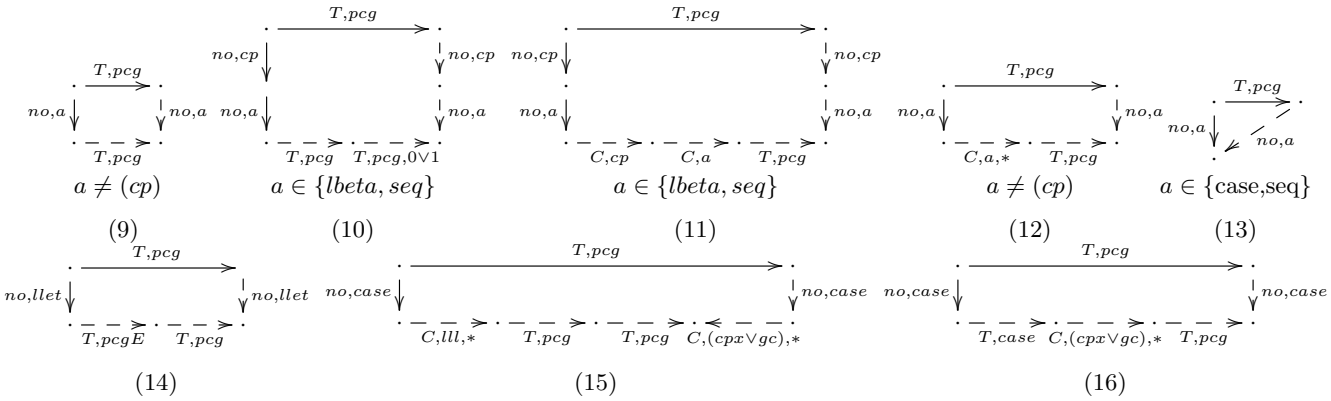


Fig. 5: Forking-diagrams for (pcg)

Proposition 4.3. *If* $s \xrightarrow{T,pcgE} t$, *then* $\texttt{rln}(s) \geq \texttt{rln}(t)$ *and* $\texttt{rlnall}(s) \geq \texttt{rlnall}(t)$. *Moreover, the transformation (pcgE) is an improvement, i.e.* $(pcgE) \subseteq \succeq$.

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. 4 (details are in Appendix C.1) where solid lines are the given reductions and dashed lines are the existential reductions. By induction on $\texttt{rlnall}(s)$ we show that $\texttt{rlnall}(t) \leq \texttt{rlnall}(s)$ and $\texttt{rln}(t) \leq \texttt{rln}(s)$. If $\texttt{rlnall}(s) = 0$ then s is a WHNF, and t must also be a WHNF and $\texttt{rlnall}(t) = \texttt{rln}(t) = 0$. If $\texttt{rlnall}(s) > 0$, then let $s \xrightarrow{no,a} s'$.

- For diagram (1) we can apply the induction hypothesis to $s' \xrightarrow{T,pcgE} t'$, since $\texttt{rlnall}(s') = \texttt{rlnall}(s) - 1$. This shows that $\texttt{rlnall}(t) \leq \texttt{rlnall}(s)$ and also that $\texttt{rln}(t) \leq \texttt{rln}(s)$.
- For diagram (2) we can apply the induction hypothesis to $s' \xrightarrow{T,pcgE} t$ which shows $\texttt{rlnall}(t) \leq \texttt{rlnall}(s') = \texttt{rlnall}(s) - 1$ and $\texttt{rln}(t) \leq \texttt{rln}(s') = \texttt{rln}(s)$.
- For diagram (3) we have $\texttt{rlnall}(s) > \texttt{rlnall}(s')$ and $\texttt{rln}(s) = \texttt{rln}(s')$. By Theorem 3.2 (3) $\texttt{rlnall}(s'') \leq \texttt{rlnall}(s')$ and $\texttt{rln}(s'') \leq \texttt{rln}(s')$. We can apply the induction hypothesis to $s'' \xrightarrow{T,pcgE} t'$ which shows $\texttt{rlnall}(t') \leq \texttt{rlnall}(s'')$ and $\texttt{rln}(t') \leq \texttt{rlnall}(s'')$. This implies $\texttt{rlnall}(t) \leq \texttt{rlnall}(s)$ and also $\texttt{rln}(t) \leq \texttt{rln}(s)$.
- For diagram (4) we have $\texttt{rlnall}(s') < \texttt{rlnall}(s)$ and $\texttt{rln}(s') \leq \texttt{rln}(s)$ (or $\texttt{rln}(s') < \texttt{rln}(s)$ if $a \in \{case, beta, seq\}$). Theorem 3.2 shows that $\texttt{rlnall}(s'') \leq \texttt{rlnall}(s')$ and $\texttt{rln}(s'') \leq \texttt{rln}(s')$. Applying the induction hypothesis to $s'' \xrightarrow{T,pcgE} t'$ yields $\texttt{rlnall}(t') \leq \texttt{rlnall}(s'')$ and $\texttt{rln}(t') \leq \texttt{rln}(s'')$. This implies both $\texttt{rlnall}(t) \leq \texttt{rlnall}(s)$ as well as $\texttt{rln}(t) \leq \texttt{rln}(s)$.
- For diagram (5) obviously $\texttt{rlnall}(s) = \texttt{rlnall}(t)$ and $\texttt{rln}(s) = \texttt{rln}(t)$ hold.
- For diagram (6) we have $\texttt{rlnall}(s') < \texttt{rlnall}(s)$ and $\texttt{rln}(s) = \texttt{rln}(s')$. Applying the induction hypothesis to $s' \xrightarrow{T,pcgE} s''$ yields $\texttt{rlnall}(s'') \leq \texttt{rlnall}(s')$ and $\texttt{rln}(s'') \leq \texttt{rln}(s')$. Applying the induction hypothesis to $s'' \xrightarrow{T,pcgE} t'$ yields $\texttt{rlnall}(t') \leq \texttt{rlnall}(s'')$ and $\texttt{rln}(t'') \leq \texttt{rln}(s'')$. Since $t \xrightarrow{no,cp} t'$, this shows $\texttt{rlnall}(t) \leq \texttt{rlnall}(s)$ and $\texttt{rln}(t) \leq \texttt{rln}(s)$.
- For diagram (7) we have $\texttt{rlnall}(s') < \texttt{rlnall}(s)$ and $\texttt{rln}(s) < \texttt{rln}(s')$. By Theorem 3.2 (4) we have $\texttt{rln}(s'') \leq \texttt{rln}(s')$ and $\texttt{rlnall}(s'') \leq \texttt{rlnall}(s')$. Applying the induction hypothesis to $s'' \xrightarrow{T,pcgE} t'$ yields $\texttt{rlnall}(t') \leq \texttt{rlnall}(s'')$ and $\texttt{rln}(t') \leq \texttt{rln}(s'')$ which shows $\texttt{rlnall}(t) \leq \texttt{rlnall}(s)$ and $\texttt{rln}(t) \leq \texttt{rln}(s)$.
- For diagram (8) we have $\texttt{rlnall}(s') < \texttt{rlnall}(s)$ and $\texttt{rln}(s) < \texttt{rln}(s')$. By Theorem 3.2 (1), (3), (4) we have $\texttt{rln}(s''') \leq \texttt{rln}(s')$ and $\texttt{rlnall}(s''') = \texttt{rlnall}(s')$. Applying the induction hypothesis to $s''' \xrightarrow{T,pcgE} t'$ yields $\texttt{rlnall}(t') \leq \texttt{rlnall}(s''')$ and $\texttt{rln}(t') \leq \texttt{rln}(s'')$ which shows $\texttt{rlnall}(t) \leq \texttt{rlnall}(s)$ and $\texttt{rln}(t) \leq \texttt{rln}(s)$.

Since (pcgE) is correct (Proposition 4.2), the context lemma for improvement (Corollary 3.5) shows $(pcgE) \subseteq \succeq$.

Lemma 4.4. *If $s \xrightarrow{T,pcg} s'$, where s is a WHNF, then either s' is a WHNF, or $s' \xrightarrow{no,cp} s''$ where s'' is a WHNF.*

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

Proof. Let $s{\downarrow}$, and $s \xrightarrow{T,pcg} s'$. We show $\mathtt{rln}(s) \geq \mathtt{rln}(s')$ by induction on the measure $(\mathtt{rln}(s), \mathtt{rlnall}(s))$, ordered lexicographically. For the base case $\mathtt{rln}(s) = \mathtt{rlnall}(s) = 0$, the expression s is a WHNF and Lemma 4.4 shows that $\mathtt{rln}(s') = 0$.

For the induction step, Fig. 5 shows the overlappings between normal order reduction-steps and a $\xrightarrow{T,pcg}$-transformation (see Appendix C.2 for details) 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 3.2 (4), or Theorem 3.6 show the claim. We consider the remaining cases:

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

Fig. 6: Machine transitions

1. If $s \xrightarrow{no,case\vee seq\vee lbeta} t_1$, then one of the diagrams (9), (12), (13), (15), or (16) of Fig. 5 holds. The diagrams can be summarized as follows where $a \in \{case, seq, lbeta\}$:

$$\begin{array}{ccccccccc} s & & \xrightarrow{\qquad\qquad T,pcg \qquad\qquad} & & & & & & s' \\ {\scriptstyle no,a}\downarrow & & & & & & & & \downarrow{\scriptstyle no,a} \\ t_1 & \xrightarrow[C,a,*\vee C,lll,*]{} & t_2 & \xrightarrow[C,gc\vee cpx,*]{} & t_3 & \xrightarrow[T,pcg,*]{} & t_4 & \xleftarrow[C,gc\vee cpx,*]{} & t_5 \end{array}$$

   We have $\mathtt{rln}(t_1) = \mathtt{rln}(s) - 1$ and by Theorem 3.2 we have $\mathtt{rln}(t_3) \leq \mathtt{rln}(t_1)$. We apply the induction hypothesis for every step in $t_3 \xrightarrow{T,pcg,*} t_4$ and we derive $\mathtt{rln}(t_4) \leq \mathtt{rln}(t_1) < \mathtt{rln}(s)$. Theorem 3.2 shows that $\mathtt{rln}(t_4) = \mathtt{rln}(t_5)$, and thus $\mathtt{rln}(s') = \mathtt{rln}(t_5) + 1 \leq \mathtt{rln}(t_1) + 1 = \mathtt{rln}(s)$.
2. Let $s \xrightarrow{no,cp} t_1$. If t_1 is a WHNF, then by Lemma 4.4 $s' \xrightarrow{no,cp,0\vee 1} t_1'$ where t_1' is a WHNF and $\mathtt{rln}(s') \leq \mathtt{rln}(s)$ holds. If t_1 is not a WHNF, then $t_1 \xrightarrow{no,a} t_2$ where $a \in \{(lbeta), (seq)\}$ and diagram (10) or (11) of Fig. 5 holds.
   For diagram (10) we have:

$$\begin{array}{ccccc} s & & \xrightarrow{\quad T,pcg \quad} & & s' \\ {\scriptstyle no,cp}\downarrow & & & & \downarrow{\scriptstyle no,cp} \\ t_1 & & & & t_1' \\ {\scriptstyle no,a}\downarrow & & & & \downarrow{\scriptstyle no,a} \\ t_2 & \xrightarrow[T,pcg]{} & t_3 & \xrightarrow[T,pcg,0\vee 1]{} & t_4 \end{array}$$

   Then $\mathtt{rln}(t_2) < \mathtt{rln}(s)$ and we apply the induction hypothesis to $t_2 \xrightarrow{T,pcg} t_3$ which shows $\mathtt{rln}(t_3) \leq \mathtt{rln}(t_2) < \mathtt{rln}(s)$. We then apply the induction hypothesis to $t_3 \xrightarrow{T,pcg,0\vee 1} t_4$ which shows $\mathtt{rln}(t_4) \leq \mathtt{rln}(t_2) < \mathtt{rln}(s)$ and $\mathtt{rln}(s') \leq \mathtt{rln}(s)$.
   Similarly, in diagram (11) 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 $\mathtt{rln}(t_2) < \mathtt{rln}(s)$ and by Theorem 3.2 (1) $\mathtt{rln}(t_4) \leq \mathtt{rln}(t_2)$

and we apply the induction hypothesis to $t_4 \xrightarrow{T,pcg} t_5$ and have $\texttt{rln}(t_5) \leq \texttt{rln}(t_4) < \texttt{rln}(s)$ and $\texttt{rln}(s') \leq \texttt{rln}(s)$.

3. If $s \xrightarrow{no,lll} t_1$, then the one of the diagrams (9), (12), or (14) of Fig. 5 holds, which can be summarized as follows:

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

Then $\texttt{rln}(t_1) = \texttt{rln}(s)$, and $\texttt{rlnall}(t_1) = \texttt{rlnall}(s) - 1$.Theorem 3.2 (3), and Proposition 4.3 show that $\texttt{rln}(t_2) \leq \texttt{rln}(t_1)$ and $\texttt{rlnall}(t_2) \leq \texttt{rlnall}(t_1)$. Thus we can apply the induction hypothesis to $t_2 \xrightarrow{T,pcg} t_3$ which yields $\texttt{rln}(t_3) \leq \texttt{rln}(t_2)$. Since $s' \xrightarrow{no,lll} t_3$, we have $\texttt{rln}(s') = \texttt{rln}(t_3) \leq \texttt{rln}(t_2) = \texttt{rln}(t_1) = \texttt{rln}(s)$ which shows the claim. □

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

Proof. For (pcgE) the claim is proved in Proposition 4.3 and for (pcg) this follows from Lemma 4.5, correctness of (pcg) (Proposition 4.2) and the context lemma for improvement (Corollary 3.5).

As already demonstrated, the transformation (cse) can be represented as a sequence $\xleftarrow{gc} . \xrightarrow{pcg,*}$ and since $(gc) \subseteq \approx$ by Theorem 3.7, this shows that (cse) is an improvement.

5 The Improvement Theory of Moran & Sands

We investigate the relationship between our measure $\texttt{rln}(\cdot)$ and the counting measures used in [9, 4] for their improvement relations. In Theorem 5.11 we show that $\texttt{rln}(\cdot)$ coincides with the number of essential transition steps of the abstract machine of [9]. In Theorem 5.15 we compare the number of all transitions steps (the measure used by [9]) with our measure and the measure used by [4].

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 $e \in E$: $f(e) \leq c * g(e)$.

Definition 5.1. *Let $\mathcal{K}_1 = (E_1, \sim_1, \texttt{size}_1, \mu_1)$, $\mathcal{K}_2 = (E_2, \sim_2, \texttt{size}_2, \mu_2)$ be two calculi with sets of expressions, contextual equivalences, size-measures for expressions, and measures for reduction length of expressions.*

Then a translation $\phi_1 : \mathcal{K}_1 \to \mathcal{K}_2$ is size-preserving*, iff $\texttt{size}_1(e) \in O(\texttt{size}_2(\phi(e)))$ and ϕ is fully abstract; i.e., for all $e, e' \in E_1$: $e \sim_1 e' \iff \phi_1(e) \sim_2 \phi_1(e')$.*

Then $\phi : \mathcal{K}_1 \to \mathcal{K}_2$ is asymptotically resource-preserving*, if ϕ is a size-preserving translation such that there exists an $n \in \mathbb{N}$ with $\mu_1(e_1) \in O(\texttt{size}_2(\phi_1(e_1))^n * (\mu_2(\phi_1(e_1)) + 1))$.*

At the very end of this section (Theorem 5.17) we prove several results on asymptotic resource-preserving translations between the calculus LR and the abstract machine of [9] w.r.t. different measures for reduction lengths.

We first recall the abstract machine used by [9]. 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[2] $(\texttt{seq}\ s\ t)$, and constructor applications $(c\ t_1 \dots t_{ar(c)})$ the expressions t, t_i must be variables.

[2] Note that the syntax in [9] does not have `seq`-expressions.

(mo,cpv-e)	$(\texttt{letrec}\ Env, x = v^{\mathsf{sub}}, y = C[x^{\mathsf{sub}\vee\mathsf{nontarg}}]\ \texttt{in}\ s) \to (\texttt{letrec}\ Env, x = v, y = C[v]\ \texttt{in}\ s)$	where v is a value
(mo,cpv-in)	$(\texttt{letrec}\ Env, x = v^{\mathsf{sub}}\ \texttt{in}\ C[x^{\mathsf{sub}}]) \to (\texttt{letrec}\ Env, x = v\ \texttt{in} C[v])$	where v is a value
(mo,β-var)	$C[(\lambda x.s)^{\mathsf{sub}}\ y] \to C[s[y/x]]$	
(mo,casecx)	$C[(\texttt{case}_K\ (c_{K,i}\ \overrightarrow{x})^{\mathsf{sub}}\ \texttt{of}\ \ldots((c_{K,i}\ \overrightarrow{y})\ \texttt{->}\ s)\ldots)] \to C[s[\overrightarrow{x}/\overrightarrow{y}]]$	
(mo,seq-c)	$C[(\texttt{seq}\ v^{\mathsf{sub}}\ x)] \to C[x]$	
(mo,glletm)	$R^-[(\texttt{letrec}\ Env\ \texttt{in}\ s)^{\mathsf{sub}}] \to (\texttt{letrec}\ Env\ \texttt{in}\ R^-[s])$	
(mo,gllet-in)	$(\texttt{letrec}\ Env_1\ \texttt{in}\ C[(\texttt{letrec}\ Env_2\ \texttt{in}\ s)^{\mathsf{sub}}]) \to (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ C[s])$	
(mo,gllet-e)	$(\texttt{letrec}\ y = C[(\texttt{letrec}\ Env_1\ \texttt{in}\ s)^{\mathsf{sub}}], Env_2\ \texttt{in}\ t) \to (\texttt{letrec}\ y = C[s], Env_1, Env_2\ \texttt{in}\ t)$	

Fig. 7: Machine order reduction rules

Definition 5.2. *The translation ψ from arbitrary* LR*-expressions into machine expressions is*

$$\begin{array}{lll}
\psi(x) & := x, \quad \textit{if } x \in \textit{Var} \\
\psi(s\ t) & := \texttt{letrec}\ x = \psi(t)\ \texttt{in}\ (\psi(s)\ x) \\
\psi(\texttt{seq}\ s\ t) & := \texttt{letrec}\ x = \psi(t)\ \texttt{in}\ (\texttt{seq}\ \psi(s)\ x) \\
\psi(c\ s_1\ \ldots\ s_n) & := \texttt{letrec}\ x_1 = \psi(s_1), \ldots, x_n = \psi(s_n) \\
 & \quad\ \texttt{in}\ (c\ x_1\ \ldots\ x_n) \\
\psi(M[s_1, \ldots, s_n]) & := M[\psi(s_1), \ldots, \psi(s_n)]
\end{array}$$

where the multicontext M is $\texttt{letrec}\ x_1 = [\cdot], \ldots, x_n = [\cdot]\ \texttt{in}\ [\cdot]$, $\lambda x.[\cdot]$, *or* $\texttt{case}_K\ [\cdot]\ \texttt{of}\ (pat_1\ \texttt{->}[\cdot]) \ldots (pat_n\ \texttt{->}[\cdot])$.

Since $\psi(t) \xrightarrow{C,ucp,*} t$, Theorem 3.2 (5) implies:

Lemma 5.3. *For all closed $t \in Expr$:* $\texttt{rln}(t) = \texttt{rln}(\psi(t))$.

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 `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. 6. 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:

$$\begin{array}{ll}
\phi(\langle \Gamma \mid s \mid \#\mathsf{upd}(x) : S \rangle) & = \phi(\langle \Gamma, x = s \mid x \mid S \rangle) \\
\phi(\langle \Gamma \mid s \mid \#\mathsf{app}(x) : S \rangle) & = \phi(\langle \Gamma \mid (s\ x) \mid S \rangle) \\
\phi(\langle \Gamma \mid s \mid \#\mathsf{seq}(x) : S \rangle) & = \phi(\langle \Gamma \mid (\texttt{seq}\ s\ x) \mid S \rangle) \\
\phi(\langle \Gamma \mid s \mid \#\mathsf{case}(alts) : S \rangle) & = \phi(\langle \Gamma \mid (\texttt{case}\ s\ \texttt{of}\ alts) \mid S \rangle) \\
\phi(\langle \Gamma \mid s \mid [] \rangle) & = \texttt{letrec}\ \Gamma\ \texttt{in}\ s
\end{array}$$

Note that $\phi(\langle \Gamma \mid v \mid [] \rangle) = \texttt{letrec}\ \Gamma\ \texttt{in}\ v$ and thus accepting states are mapped to WHNFs.

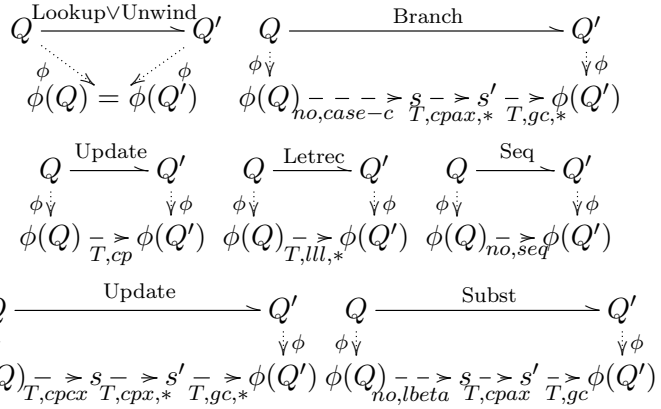
Definition 5.4. *Let s be a closed machine expression such that $\langle \emptyset \mid s \mid [] \rangle \xrightarrow{n} Q$ where Q is an accepting state. Then* $\texttt{mlnall}(s) = n$ *and* $\texttt{mln}(s)$ *is the sum of all (*Subst*)-, (*Branch*)-, and (*Seq*)-steps in the sequence and* $\texttt{mlnlook}(s)$ *is the number of all (*Lookup*)-transitions. If no such sequence exists for s, then* $\texttt{mlnall}(s) = \texttt{mln}(s) = \texttt{mlnlook}(s) = \infty$. *We use* $\texttt{mln}(\cdot)$ *with the same meaning also for reachable states Q of the machine.*

Note that the improvement theory in [9] is based on the measure $\texttt{mlnall}(\cdot)$, whereas the measure in [4] is $\texttt{mlnlook}(\cdot)$.

5.1 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 $\texttt{rln}(s) = \texttt{mln}(s)$. With Lemma 5.3 this also implies that for every LR-expression s the equation $\texttt{rln}(s) = \texttt{mln}(\phi(s))$ holds.

Lemma 5.5. *Let Q be a reachable state, and $Q \rightharpoonup Q'$. Then for $\phi(Q)$ and $\phi(Q')$ one of the following cases holds:*



Proposition 5.6. *Let s be a closed machine expression with $\texttt{mln}(s) = n$. Then $\texttt{rln}(s) = n$.*

Proof. 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 5.5 to $Q_0 \rightharpoonup Q_1$ and then apply the induction 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 3.2 and 3.6 from right to left to every transformation which is not a normal order reduction. Since all these steps leave the measure $\texttt{rln}(\cdot)$ unchanged, and the normal order step increases the measure by 1, this shows $\texttt{rln}(s) = n$.

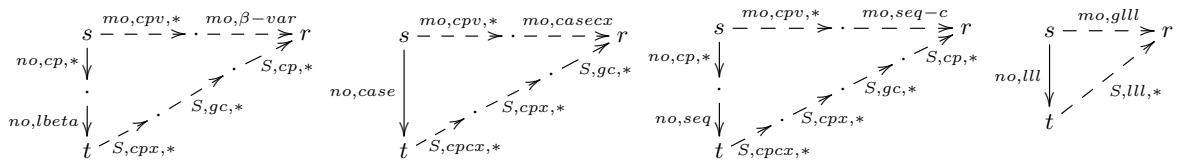
To show that for a closed machine expression s the equation $\texttt{rln}(s) = n$ also implies $\texttt{mln}(s) = n$, we define a variant of the normal order reduction for machine expressions – called *machine order reduction*: It uses the reduction rules shown in Fig. 7 and the machine order redex is found by the labeling algorithm in Definition 2.2. 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 `letrec` *Env* `in` v where v is a value. For a closed expression s, let $\texttt{rln}_{mo}(s)$ be the number of (mo,β-var)-, (mo,casecx)-, (mo,seqc)-reductions in a machine order reduction sequence from s to an MWHNF, and $\texttt{rln}_{mo}(s) = \infty$ otherwise.

Lemma 5.7. *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 5.8. *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. 8 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 5.9. *Let s be a closed machine expression with $\texttt{rln}(s) = n$. Then $\texttt{rln}_{mo}(s) = n$.*

Proof. Let $s \xrightarrow{no,k} t_k$ where t_k is a WHNF. We use induction on $(\texttt{rln}(s), \texttt{rlnall}(s))$ to show the claim. If $\texttt{rlnall}(s) = 0$, then s is a WHNF, and $s \xrightarrow{mo,cpv,*} s'$ where s' is an MWHNF and thus



(17) (18) (19) (20)

Fig. 8: Diagrams for transferring normal order reductions into machine order reductions

$\mathtt{rln}_{mo}(s) = 0$. Now assume $\mathtt{rlnall}(s) > 0$. If $s \xrightarrow{no,cp} s'$, where s' is a WHNF, then $s \xrightarrow{mo,cpv,*} s''$, where s'' is a WHNF, and so $\mathtt{rln}(s) = \mathtt{rln}_{mo}(s'') = 0$. In the other cases we apply a diagram from Lemma 5.8 to a prefix of $s \xrightarrow{no,k} t_k$.

For diagram (20) we have $\mathtt{rln}(t) = \mathtt{rln}(s)$ and $\mathtt{rlnall}(t) < \mathtt{rlnall}(s)$. By Theorem 3.2 (3) $\mathtt{rln}(r) = \mathtt{rln}(s)$ and $\mathtt{rlnall}(r) < \mathtt{rlnall}(s)$. We apply the induction hypothesis to r and get $\mathtt{rln}_{mo}(r) = \mathtt{rln}(s)$ and thus $\mathtt{rln}_{mo}(s) = \mathtt{rln}(s)$.

If diagram (18), (19), or (17) is applied, then $\mathtt{rln}(t) = \mathtt{rln}(s) - 1$ and Theorem 3.2 shows that $\mathtt{rln}(r) = \mathtt{rln}(t)$. Applying the induction hypothesis to r shows $\mathtt{rln}_{mo}(r) = \mathtt{rln}(s) - 1$.

Since $s \xrightarrow{mo,*} r$ where exactly one (mo,casecx), (mo,seq-c), or (mo,β-var) is in the sequence, this shows $\mathtt{rln}_{mo}(s) = \mathtt{rln}(s)$. □

Proposition 5.10. *If* $\mathtt{rln}(s) = n$*, then* $\mathtt{mln}(s) = n$*.*

Proof. From $\mathtt{rln}(s) = n$ we get $\mathtt{rln}_{mo}(s) = n$ by Proposition 5.9. 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) ∨ (Lookup)) show the relationship between $s \xrightarrow{mo} s_0$ and the machine transition for Q_0:

$$\begin{array}{ccc} s & \xrightarrow{mo,cpv} & s_0 \\ \phi\uparrow & & \uparrow\phi \\ Q_0 & \xrightarrow[UL,*\ \text{Update}]{} & Q_1 \end{array} \qquad \begin{array}{ccc} s & \xrightarrow{mo,\beta-var} & s_0 \\ \phi\uparrow & & \uparrow\phi \\ Q_0 & \xrightarrow[UL,*\ \text{Subst}]{} & Q_1 \end{array} \qquad \begin{array}{ccc} s & \xrightarrow{mo,seqc} & s_0 \\ \phi\uparrow & & \uparrow\phi \\ Q_0 & \xrightarrow[UL,*\ \text{Seq}]{} & Q_1 \end{array}$$

$$\begin{array}{ccc} s & \xrightarrow{mo,casecx} & s_0 \\ \phi\uparrow & & \uparrow\phi \\ Q_0 & \xrightarrow[UL,*\ \text{Branch}]{} & Q_1 \end{array} \qquad \begin{array}{ccc} s & \xrightarrow{mo,glll} & s_0 \\ \phi\uparrow & & \uparrow\phi \\ Q_0 & \xrightarrow[UL,*\ \text{Letrec}]{} & Q_1 \end{array}$$

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

By Propositions 5.6 and 5.10 and Lemma 5.3 we have:

Theorem 5.11. *For any closed* $s \in$ *Expr:* $\mathtt{rln}(s) = \mathtt{mln}(\psi(s))$*.*

Corollary 5.12. *The translation* ψ *seen as a translation from* LR *to the abstract machine of [9] in the variant presented in this paper is fully-abstract.*

5.2 Relating Essential and All Transition Steps

We write (ULLU) for the union of (Unwind), (Letrec), (Lookup), (Update) and (SBS) for the union of (Subst), (Branch), (Seq).

Theorem 5.13. *Let* s *be a closed machine expression with* $s{\downarrow}$*. Then* $\mathtt{mlnall}(s) \leq 3 * (\mathtt{size}(s) + 2) * (\mathtt{mln}(s) + 1)$*, where* $\mathtt{size}(\cdot)$ *is the size of an expression (viewed as syntax tree).*

Proof. Let $\texttt{mln}(s) = n$ 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 $(n+1)*\texttt{size}(s)$ (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 $(n + 1) * (\texttt{size}(s) + 1)$ (Lookup)-transitions.

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

We analyse whether counting the number of (Lookup)-transitions is appropriate as claimed in [9] and used in [4].

Proposition 5.14. *Let s be a closed* LR*-expression with $s \downarrow$. Then* $\texttt{mlnall}(s) \leq (2 * \texttt{size}(s) * (\texttt{mlnlook}(\phi(s)) + 1))$.

Proof. Consider a valid transition subsequence without a (Lookup)-transition. For every intermediate machine state $m_i = \langle \Gamma \mid s_i \mid S_i \rangle, i = 1, \ldots, n$ consider the expression $u_i = \phi(\langle \emptyset \mid s_i \mid S_i \rangle)$. Then $\texttt{size}(u_i)$ 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 $\texttt{size}(s)$, (as already argued) hence $\texttt{mln}(s)$ + (number of (Update)s) + (number of (Letrec)s) is not greater than $\texttt{size}(s) * (\texttt{mlnlook}(\phi(s)) + 1)$. Since the overall number of (Unwind)s is exactly $\texttt{mln}(s)$, we obtain $\texttt{mlnall}(s) \leq (2 * \texttt{size}(s) * (\texttt{mlnlook}(\phi(s)) + 1))$.

Theorem 5.15. *Let s be a closed machine expression. Then* $\texttt{mlnlook}(s) \leq \texttt{mlnall}(s) \leq (2 * \texttt{size}(s) * (\texttt{mlnlook}(s) + 1))$, *and* $\texttt{mln}(s) \leq \texttt{mlnall}(s) \leq 3 * (\texttt{size}(s) + 4) * (\texttt{mln}(s) + 1)$.

Remark 5.16. Theorem 5.15 justifies our claim that common subexpression elimination (also called β-expand) is an improvement in [9] and also in [4]. However, our proofs only show that this is the case if improvement is defined w.r.t. `mln`(.) in their calculus. Note that also the size is not increased (up to the initial inverse gc) by common subexpression elimination.

The results in this section imply:

Theorem 5.17. *The following calculi allow asymptotically resource-preserving translations into each other: (i)* LR *with* `rln`*; (ii) Moran-Sands calculus with* `mlnall`*; (iii) Moran-Sands calculus with* `mln`*; and (iv) the Moran-Sands calculus with* `mlnlook`*.*

Note that in LR switching from `rln`(.) to `rlnall`(.) is not resource-preserving, since there are LR-expressions s s.t. $\texttt{rlnall}(s) \in O(\texttt{size}(s)^n(\texttt{rln}(s)+1))$ is false for all n (see 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.

6 The Polymorphically Typed Calculus LRP

In this section we consider the polymorphically typed variant LRP of the calculus LR. The type erasing translation from LRP into LR is adequate: equivalences and improvements in LR will also be

Type variables: $a, a_i \in TVar$ Types: $\tau \in Typ := a \mid (\tau_1 \to \tau_2) \mid K\ \tau_1 \ldots \tau_{ar(K)}$
Term variables: $x, x_i \in Var$ Polymorphic types: $\rho \in PTyp := \tau \mid \boldsymbol{\lambda} a.\rho$
Patterns: $pat_{K,i} := (c_{K,i} :: \tau\ x_1 :: \tau_1 \ldots x_{ar(c_{K,i})} :: \tau_{ar(c_{K,i})})$ Polymorphic abstractions: $s \in PExpr_F := \Lambda a_1 \ldots . \Lambda a_k . \lambda x.s$
Expressions: $s, t \in Expr_F := u \mid x :: \rho \mid (s\ \tau) \mid (s\ t) \mid (c_{K,i} :: \tau\ s_1\ \ldots\ s_{ar(c_{K,i})}) \mid (\texttt{seq}\ s\ t)$
$\mid (\texttt{letrec}\ x_1 :: \rho_1 = s_1, \ldots, x_n :: \rho_n = s_n\ \texttt{in}\ t) \mid (\texttt{case}_K\ s\ \texttt{of}\ (pat_{K,1}\ \texttt{->}\ t_1) \ldots (pat_{K,|D_K|}\ \texttt{->}\ t_{|D_K|}))$

(a) Types and expressions of the language LRP (see also [18]).

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

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

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

(b) Typing Rules for LRP

$(s\ t)^{\mathsf{sub} \vee \mathsf{top}}$	$\to$	$(s^{\mathsf{sub}}\ t)^{\mathsf{vis}}$	$s \neq \Lambda a.e'$
$((\Lambda a.u)\ \tau)^{\mathsf{sub} \vee \mathsf{top}}$	$\to$	$((\Lambda a.u)^{\mathsf{sub}}\ \tau)^{\mathsf{vis}}$;	then stop with success
$(\texttt{letrec}\ Env\ \texttt{in}\ s)^{\mathsf{top}}$	$\to$	$(\texttt{letrec}\ Env\ \texttt{in}\ s^{\mathsf{sub}})^{\mathsf{vis}}$	
$(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C[x^{\mathsf{sub}}])$	$\to$	$(\texttt{letrec}\ x = s^{\mathsf{sub}}, Env\ \texttt{in}\ C[x^{\mathsf{vis}}])$	
$(\texttt{letrec}\ x = s, y = C[x^{\mathsf{sub}}], Env\ \texttt{in}\ t)$	$\to$	$(\texttt{letrec}\ x = s^{\mathsf{sub}}, y = C[x^{\mathsf{vis}}], Env\ \texttt{in}\ t)$	where $C \neq [\cdot]$
$(\texttt{letrec}\ x = s, y = x^{\mathsf{sub}}, Env\ \texttt{in}\ t)$	$\to$	$(\texttt{letrec}\ x = s^{\mathsf{sub}}, y = x^{\mathsf{nontarg}}, Env\ \texttt{in}\ t)$	
$(\texttt{seq}\ s\ t)^{\mathsf{sub} \vee \mathsf{top}}$	$\to$	$(\texttt{seq}\ s^{\mathsf{sub}}\ t)^{\mathsf{vis}}$	
$(\texttt{case}_K\ s\ \texttt{of}\ alts)^{\mathsf{sub} \vee \mathsf{top}}$	$\to$	$(\texttt{case}_K\ s^{\mathsf{sub}}\ \texttt{of}\ alts)^{\mathsf{vis}}$	
$\texttt{letrec}\ x = s^{\mathsf{vis} \vee \mathsf{nontarg}}, y = C[x^{\mathsf{sub}}], Env\ \texttt{in}\ t$	$\to$	$\texttt{Fail}$	
$(\texttt{letrec}\ x = C[x^{\mathsf{sub}}], Env\ \texttt{in}\ t)$	$\to$	$\texttt{Fail}$	

(c) Computing reduction positions using labels in LRP, where $a \vee b$ means label a or label b. The algorithm does not overwrite labels.

Fig. 9: Syntax, Typing rules, and Labeling for the Calculus LRP

equivalences and improvements in LRP, provided that they are well-typed. Reduction sequences in LRP are mapped into LR-reduction sequences, of the same $\texttt{rln}(.)$-length, since we do not include type reductions in the length measure.

However, there are more equivalences and improvements in LRP than in LR (see Sect. 6.2), since a smaller set of contexts is taken into account: only those contexts need to be considered which leave the expressions well-typed. Since LRP is a core language of (pure) Haskell [8] our results are applicable there.

The extensions of LRP 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 [2, 12, 11, 23]. See also [18] for an analysis of typed LR of simulations as a tool for correctness.

The calculus LRP is related to PolyPCF [12], a polymorphically typed PCF. Differences are that LRP observes convergence in every context, while PolyPCF only observes convergence to list contexts; LRP employs a cyclic let, and the `seq`-operator, but not a fix-operator. These syntactical and operational differences make contextual equivalencess essentially different in the two calculi.

The syntax of the calculus LRP is defined in Fig. 9a, where every data constructor $c \in D_K$ has a polymorphic type $type(c)$ of the form $\boldsymbol{\lambda} a_1, \ldots a_k.\tau_1 \to \ldots \tau_{ar(c)} \to K(a_1, \ldots, a_k)$. The typing rules are in Fig. 9b- All expressions of a polymorphic type $\boldsymbol{\lambda} a.\rho$ are of the form $x :: \rho$, $\Lambda.e$, $(e\ \tau)$, and $(\texttt{letrec}\ Env\ \texttt{in}\ e)$, other forms are not possible. A *polymorphic abstraction* is an expression of the form $\Lambda a_1, \ldots, a_k.\lambda x.e$, and a *value* is defined as an abstraction, a polymorphic abstraction, or a constructor application.

6.1 Semantics of LRP

The reduction rules in LRP extend the rules of LR by reductions for type abstractions and applications:

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

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

and all other rules from LR *(Fig. 2), extended by types and the extended syntax as follows:*

- *Every variable is labeled with a type.*
- *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.*
- *In rule (cp) also polymorphic abstractions can be copied, i.e.*
  (cp-in) $(\texttt{letrec}\ x_1 = u^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \ \texttt{in}\ C[x_m^{\mathsf{vis}}]) \to (\texttt{letrec}\ x_1 = u, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \ \texttt{in}\ C[u])$*, where u is an abstraction or a polymorphic abstraction.*

The rules of the labeling algorithm are in Fig. 9c. 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. For reductions we use the same notational conventions as for LR.

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

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 $C[\cdot :: \rho]$: $C[s]{\downarrow} \iff C[t]{\downarrow}$.*

One can verify that contextual equivalence also satisfies the type substitution properties of logical relations (see for example [12]): If $s :: \tau \sim_c t :: \rho$, then also $(s :: \rho)[\tau'/a] \sim_c t :: \rho[\tau'/a]$, and if $s :: \boldsymbol{\lambda} a.\rho \sim_c t :: \boldsymbol{\lambda} a.\rho$, then also $(s :: \boldsymbol{\lambda} a.\rho)\ \tau \sim_c (t :: \boldsymbol{\lambda} a.\rho)\ \tau$.

Definition 6.4. *The type erasure function ϵ : 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.

Proposition 6.5. *As a translation of calculi, ε is adequate, i.e. $\varepsilon(e_1) \sim_{\text{LRP}} \varepsilon(e_2) \implies e_1 \sim_c e_2$; and it is resource-preserving.*

The translation ε is not fully abstract. An example are the equivalences by (caseId), see below.

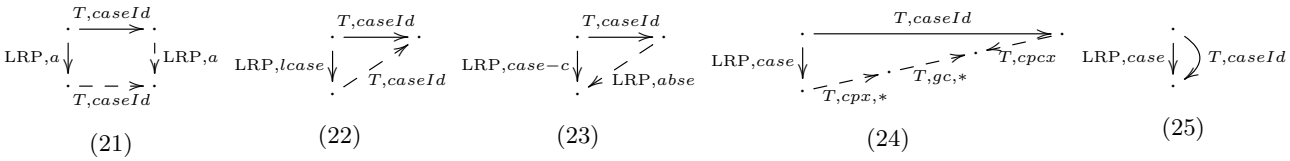


Fig. 10: Diagrams for (caseId)

6.2 Improvement in LRP

The measure for estimating the time consumption of computation also in LRP is $\mathtt{rln}(t)$. It is not necessary to count (TBeta)-reductions, since every normal-order reduction sequence of s consisting only of type-reductions terminates, and there are most $O(n)$ such steps where n is the size of the expression s. The size of type expressions may grow large by the call-by-name type reduction. However, using dags for compressing the types leads to a polynomial size grow of types.

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

The following facts are valid and can easily be verified:

1. For closed LRP-expressions s, the equation $\mathtt{rln}(s) = \mathtt{rln}(\varepsilon(s))$ 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.

For $\eta \in \{\leq, =, \geq\}$ and a class of contexts X we define: For s, t of type ρ the relation $s \bowtie_{\eta,X} t$ (in LRP) holds iff for all X-contexts $X[\cdot : \rho]$: if $X[s], X[t]$ are closed, then $\mathtt{rln}(X[s])\ \eta\ \mathtt{rln}(X[t])$. 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 $s \bowtie_{\eta,R} t$ (or $s \bowtie_{\eta,S} t$ or $s \bowtie_{\eta,T} t$) implies $s \bowtie_{\eta,C} t$,*

We end this section by showing that the transformation (caseId) is an improvement in LRP, where (caseId) is defined as:

$$(\mathtt{case}_K\ s\ \mathtt{of}\ (pat_1 \to pat_1) \ldots (pat_{|D_K|} \to pat_{|D_K|})) \to s$$

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 $s \xrightarrow{\text{LRP},lll,*} \xrightarrow{\text{LRP},case,0\vee1} \xrightarrow{\text{LRP},lll,*} s'$ where s' is a WHNF.*

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

Proof. Let $s \xrightarrow{T,caseId} t$ and $s \xrightarrow{\text{LRP},k} s'$ where s' is a WHNF. We use induction on k. For $k = 0$ Lemma 6.8 shows the claim. For the induction step, let $s \xrightarrow{\text{LRP}} s_1$. The diagrams in Fig. 10 describe all cases how the fork $s_1 \xleftarrow{\text{LRP}} s \xrightarrow{T,caseId}$ can be closed. For diagram (21) we apply the induction hypothesis to $s_1 \xrightarrow{T,caseId} t_1$ which shows $t_1\!\downarrow$, $\mathtt{rln}(s_1) \geq \mathtt{rln}(t_1)$ and thus also $t\!\downarrow$ and $\mathtt{rln}(s) \geq \mathtt{rln}(t)$. For diagram (22) the induction hypothesis shows the claim. For diagram (23) 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 3.2 (3) and (5) show $\mathtt{rln}(s') = \mathtt{rln}(t)$. Thus also $\mathtt{rln}(s) \geq \mathtt{rln}(t)$. For diagram (24) we have $t\!\downarrow$, since (cpcx),(gc), and (cpx) are correct. Theorem 3.2 shows that $\mathtt{rln}(s) > \mathtt{rln}(s') = \mathtt{rln}(t)$, since (cpcx),(cpx) and (gc) do not change the measure $\mathtt{rln}(\cdot)$. For diagram(25) the claim obviously holds.

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

Proof. Lemma 6.8 and the diagrams in Fig. 10 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 22 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 $\sim_c$, see e.g. [17]) 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 [9]. Since counting in [9] is based on an abstract machine, and our counting on a subset of the reduction rules, we analysed the differences and proved that these are not substantial. We defined a polymorphic call-by-need lambda calculus LRP as a variant of LR, and defined a corresponding improvement theory.

Future work is to extend the improvement theory and application to more program transformations. Work on resource usage like space in call-by-need calculi is [3], which can be used as a starting point for further research on other forms of improvements.

References

1. Richard Bird. *Thinking functionally with Haskell.* Cambridge University Press, 2014.
2. J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types.* Cambridge University Press, 1994.
3. Jörgen Gustavsson and David Sands. Possibilities and limitations of call-by-need space improvement. In *ICFP '01*, pages 265–276, 2001.
4. Jennifer Hackett and Graham Hutton. Worker/wrapper/makes it/faster. In *ICFP '14*, pages 95–107, 2014.
5. John Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989.
6. Patricia Johann and Janis Voigtländer. The impact of seq on free theorems-based program transformations. *Fund. Inform.*, 69(1–2):63–102, 2006.
7. Arne Kutzner and Manfred Schmidt-Schauß. A nondeterministic call-by-need lambda calculus. In *ICFP '98*, pages 324–335, 1998.
8. Simon Marlow, editor. *Haskell 2010 – Language Report.* 2010.
9. A. K. D. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *POPL '99*, pages 43–56, 1999.
10. Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *J. Funct. Programming*, 12(4+5):393–434, 2002.
11. Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.
12. Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Math. Structures Comput. Sci.*, 10:321–359, 2000.
13. 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.
14. David Sabel and Manfred Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In *PPDP '11*, pages 101–112, 2011.
15. David Sands. Improvement theory and its applications. In *Higher order operational techniques in semantics*, pages 275–306. Cambridge university press, 1998.
16. M. Schmidt-Schauß, E. Machkasova, and D. Sabel. Extending Abramsky's lazy lambda calculus: (non)-conservativity of embeddings. In *RTA '13*, volume 21 of *LIPIcs*, pages 239–254, 2013.
17. 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.
18. Manfred Schmidt-Schauß and David Sabel. Contextual equivalences in call-by-need and call-by-name polymorphically typed calculi (preliminary report). In *WPTE '14*, volume 40 of *OASICS*, pages 63–74, 2014.
19. 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.
20. Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.
21. Neil Sculthorpe, Andrew Farmer, and Andy Gill. The HERMIT in the tree: Mechanizing program transformations in the GHC core language. In *IFL '12*, volume 8241 of *LNCS*, pages 86–103, 2013.

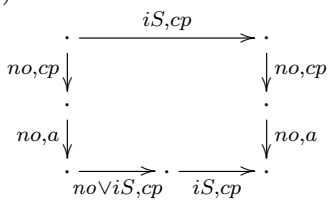
22. Peter Sestoft. Deriving a lazy abstract machine. *J. Funct. Programming*, 7(3):231–264, 1997.
23. Dimitrios Vytiniotis and Simon Peyton Jones. Evidence Normalization in System FC (Invited Talk). In *RTA '13*, volume 21 of *LIPIcs*, pages 20–38, 2013.

A Proof of Theorem 3.6

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 [20][3].

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

(26) (27)



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

(28)

We will use these diagrams to prove Theorem 3.6 which is repeated here:

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

Proof. We use the context lemma 3.5 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 $\mathtt{rln}(s) = \mathtt{rln}(s')$ by induction on $\mathtt{rln}(s)$ 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 $\mathtt{rln}(s_1) = \mathtt{rln}(s) - 1$. Either diagram (26) or (27) 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 (28). Then $s_1 \xrightarrow{no,a} s_2$ and $\mathtt{rln}(s_2) = \mathtt{rln}(s) - 1$. Hence we can apply the induction hypothesis twice, and obtain the claim.

If $s \xrightarrow{no,lll} s_1$, then diagram (26) applies, and we can apply the induction hypothesis, we have $s' \xrightarrow{no,lll} s_1$, and since $\mathtt{rln}(s') = \mathtt{rln}(s_1)$, we obtain $\mathtt{rln}(s) = \mathtt{rln}(s')$.

B On the number of `rlnall`-reductions in LR

We show by a counter example that the identity-translation from LR with `rlnall` into LR with `rln` 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 $s =$ `letrec` Env `in` $(id_1\ \ldots\ id_n)$. Then the equality $\mathtt{rlnall}(s) = \dfrac{n \cdot (n+3) - 4}{2}$ and $\mathtt{rln}(s) = n - 1$ holds.*

[3] we do not distinguish between (cpd)- and (cpt)-transformations as in [20] and simply write (cp)

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

$$\begin{array}{ll} & \texttt{letrec}\ Env\ \texttt{in}\ (id_1\ id_2) \\ \xrightarrow{no,lbeta} & \texttt{letrec}\ Env\ \texttt{in}\ \texttt{letrec}\ x_1 = id_2\ \texttt{in}\ x_1 \\ \xrightarrow{no,llet} & \texttt{letrec}\ Env, x_1 = id_2\ \texttt{in}\ x_1 \\ \xrightarrow{no,cp} & \texttt{letrec}\ Env, x_1 = id_2\ \texttt{in}\ id_2 \end{array}$$

and thus $\texttt{rln}(s) = 1$ and $\texttt{rlnall}(s) = 3$.

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

$$\begin{array}{ll} & \texttt{letrec}\ Env\ \texttt{in}\ id_1\ id_2 \ldots id_n \\ \xrightarrow{no,lbeta} & \texttt{letrec}\ Env\ \texttt{in}\ ((\texttt{letrec}\ x_1 = id_2\ \texttt{in}\ x_1)\ id_3\ \ldots id_n) \\ \xrightarrow{no,lapp,n-2} & \texttt{letrec}\ Env\ \texttt{in}\ (\texttt{letrec}\ x_1 = id_2\ \texttt{in}\ (x_1\ id_3\ \ldots id_n)) \\ \xrightarrow{no,llet} & \texttt{letrec}\ Env, x_1 = id_2\ \texttt{in}\ (x_1\ id_3\ \ldots id_n) \\ \xrightarrow{no,cp} & \texttt{letrec}\ Env, x_1 = id_2\ \texttt{in}\ (id_2\ id_3\ \ldots id_n) \end{array}$$

By the induction hypothesis, we have

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

and $\texttt{rln}(s) = n - 2 + 1 = n - 1$.

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

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

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

Then $s_{2^m} \xrightarrow{no,*} \texttt{letrec}\ Env\ \texttt{in}\ (id_1\ \ldots\ id_{2^m+1})$ 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

$$\begin{array}{rll} t_{2^m} := \texttt{letrec} & x_{2^0} = \lambda h.S\ h, \{x_{2^i} = \lambda h.x_{2^{i-1}}(x_{2^{i-1}}h)\}_{i=1}^{m}, \\ & h_0 = Z, h_{2^m} = x_{2^m}\ h_0, \\ & f = F \\ \texttt{in}\ f\ h_{2^m} & \end{array}$$

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

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

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 $\texttt{rln}(\cdot)$ 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

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

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

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

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

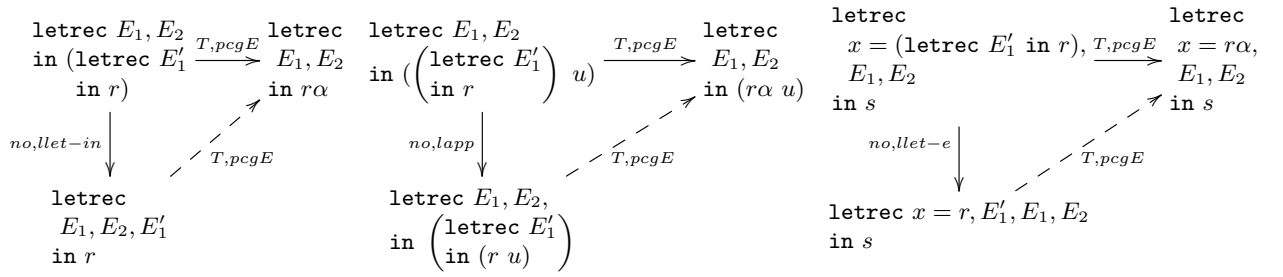
$$\begin{array}{ll} & h_i = (x_{2^j}\ (x_{2^j}\ h_k)) \\ \xrightarrow{cp} & h_i = (\lambda h_{2^j+k}.(x_{2^{j-1}}\ (x_{2^{j-1}}\ h_{2^j+k}))\ (x_{2^j}\ h_k)) \\ \xrightarrow{lbeta,llet} & h_i = (x_{2^{j-1}}\ (x_{2^{j-1}}\ h_{2^j+k})), h_{2^j+k} = (x_{2^j}\ h_k). \end{array}$$

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

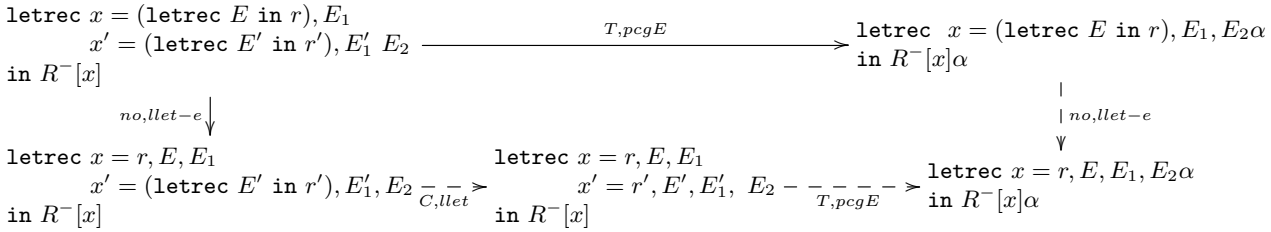
The following ideas help to prove that the generation of h_i is unique. First note that generating bindings of type (1) terminates, since the indices get smaller in every step. For uniqueness, which means that every h_i is generated only once, the following invariant can be used in an induction proof: 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 $\texttt{letrec}\ Env\ \texttt{in}\ (id_1\ \ldots\ id_{2^m+1})$ (Lemma B.1) yields $\texttt{rln}(t_{2^m}) = 3 * 2^m - 2 + 2 * (2^m + 1) + ((2^m + 1) - 1) = 6 * 2^m$.

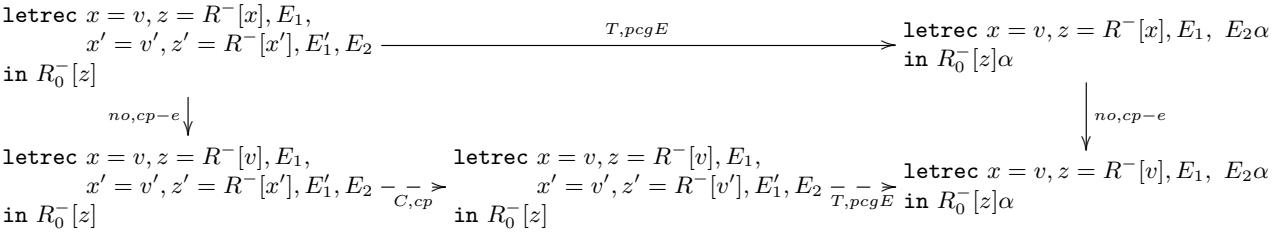
However, by Lemma B.1 $\texttt{rlnall}(t_{2^m}) \geq c * 2^{2*m} = c * (2^m)^2$ for some constant c, and $\texttt{size}(t_{2^m}) = d * m$ for some constant d. Since for all positive integers k, $(2^m)^2$ is asymptotically larger than $m^k * 2^m$, the translation from LR with measure $\texttt{rln}(\cdot)$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



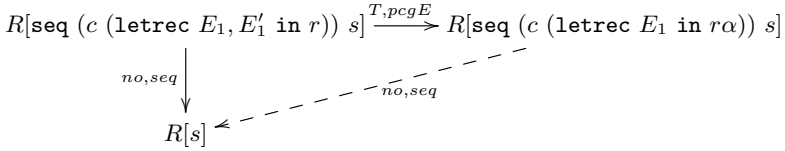
(a) Typical cases for diagram (2)



(b) Typical case for diagram (3)



(c) Typical case for diagram (4)



(d) Typical case for diagram (5)

Fig. 11: Typical cases for the diagrams for (pcgE)

C Diagrams for pcgE and pcg

C.1 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. 4.

Diagram (1) describes the case of a non-critical overlap where the steps can be commuted. Diagram (2) 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. 11a

Diagram(3) 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. 11b.

Diagram (4) 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. 11c.

Diagram (5) covers the case where the (pcgE)-redex is removed by the normal order reduction, an example is given in Fig. 11d. Diagram (6) covers the case where the (pcgE)-redex is copied by the normal order reduction, an example is given in Fig. 12a.

The seventh diagram (7) covers the case where the environment shared by (pcgE) contains a constructor application used by (no,case) reduction. An example is given in Fig. 12b.

The seventh diagram (8) covers the case where the redex of a (no,case) is shared by (pcgE). An example is given in Fig. 12c.

C.2 Diagrams for pcg

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:

- 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. 5 is applicable.

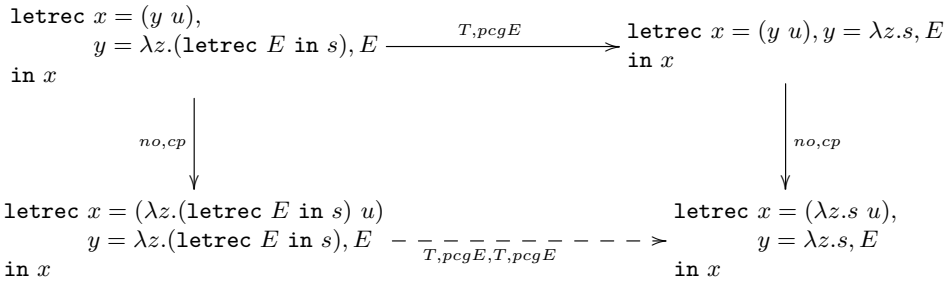
We explain the diagrams in Fig. 5 and gives exemplary instances of the diagrams:

Diagram (9) describes the case where the reductions can be commuted. Diagrams (10) and (11) 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. 13a and 13c.

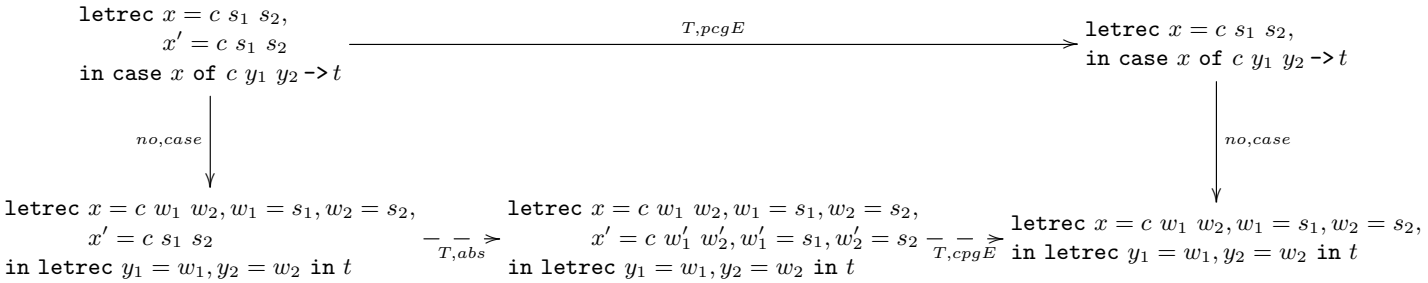
Diagram (12) describes the cases where the normal order reduction modifies the subexpression which occurs twice and is shared by the (cpg)-transformation. Three prototypical expressions and overlappings for diagram (12) are show in Fig. 13e

Diagram(13) covers the case, that the duplicated expression is inside the first argument of `seq` or in an unused alternative of a `case`-expression. A typical case is given in Fig. 13f.

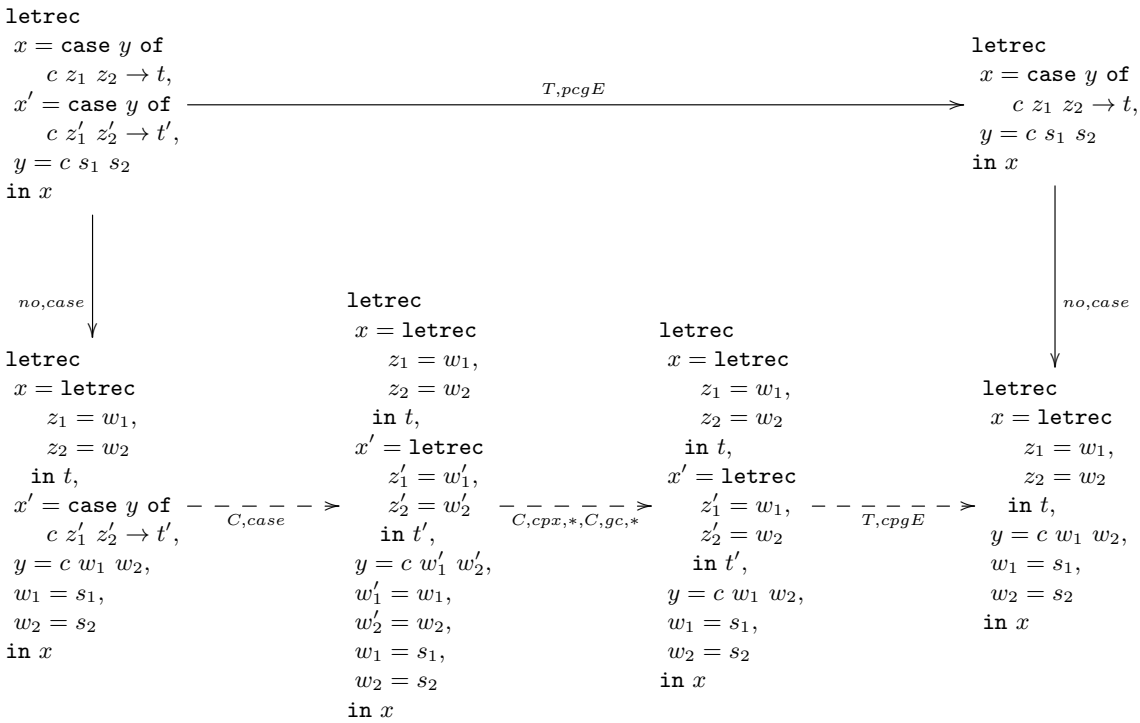
Diagram (14) covers the case that the duplicated subexpression is a `letrec`-expression which is deconstructed by an $(no, llet)$-reduction. A typical case is given in Fig. 13g. Diagram (15) covers the case that the scrutinee of a `case`-expression is one of the duplicated expressions. A typical case is given in Fig. 13h.



(a) Typical case for diagram (6)

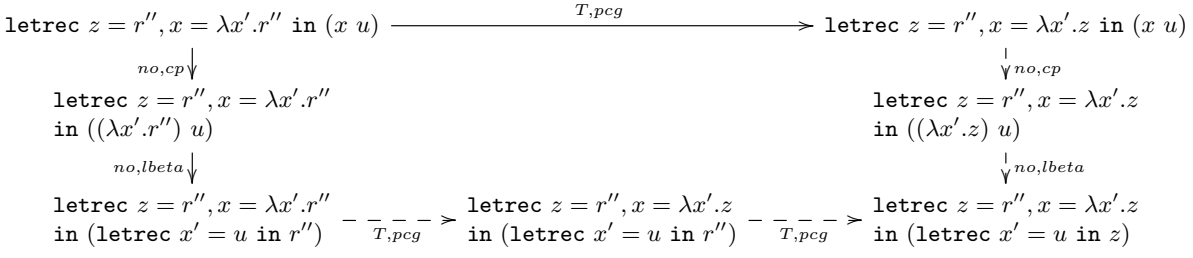


(b) Typical case for diagram (7)

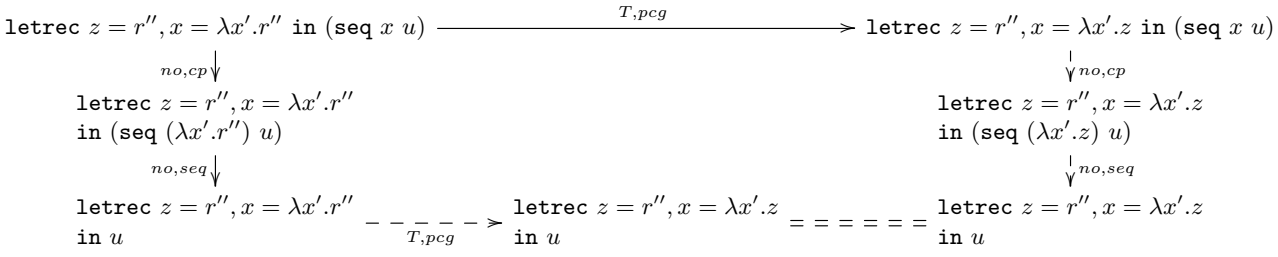


(c) Typical case for diagram (8)

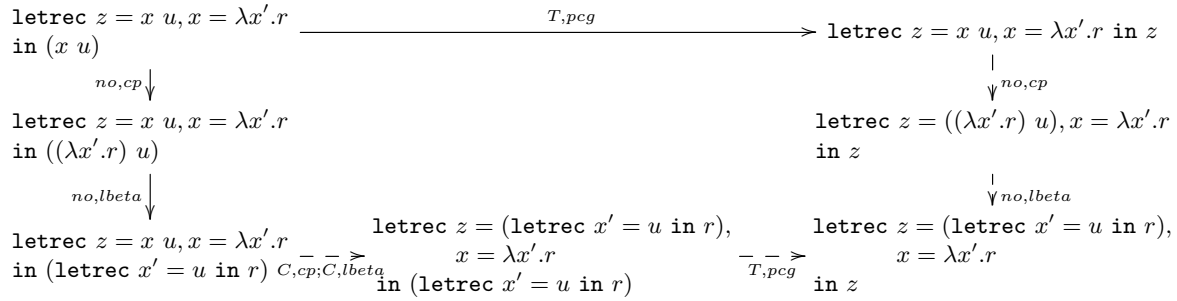
Fig. 12: Typical cases for the diagrams for (pcgE), cont'd.



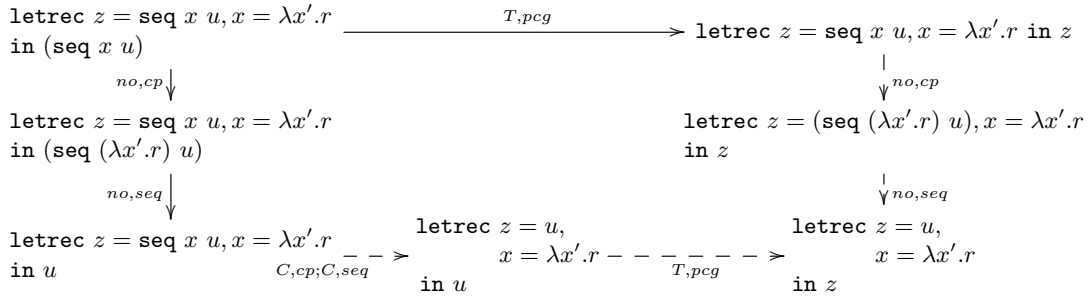
(a) Typical case for diagram (10)



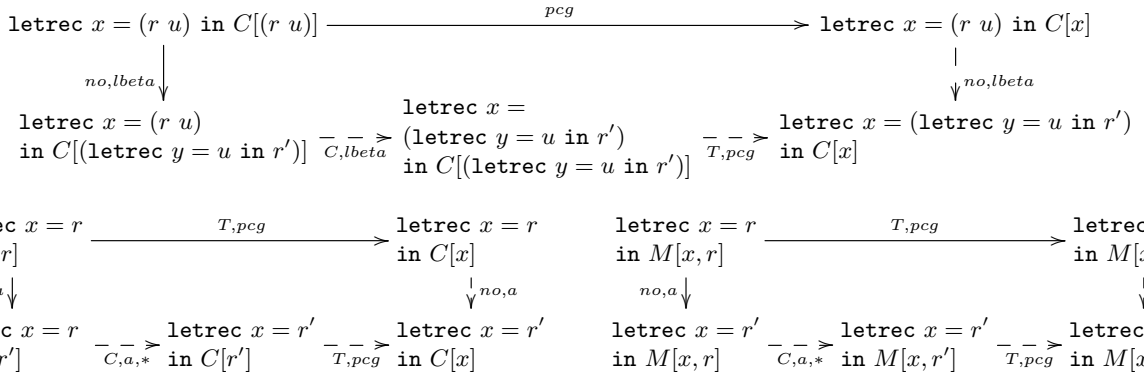
(b) Typical case for diagram (10) with seq



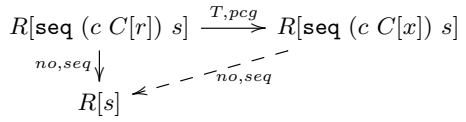
(c) Typical case for diagram (11)



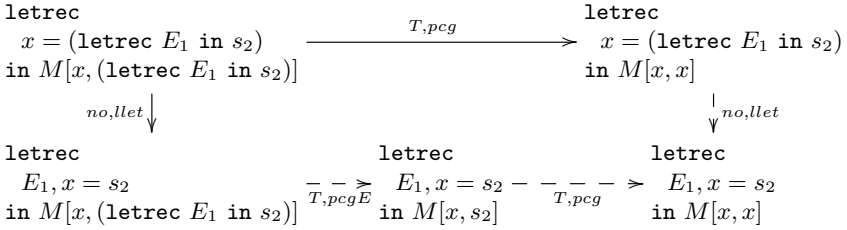
(d) Typical case for diagram (11) with (seq)



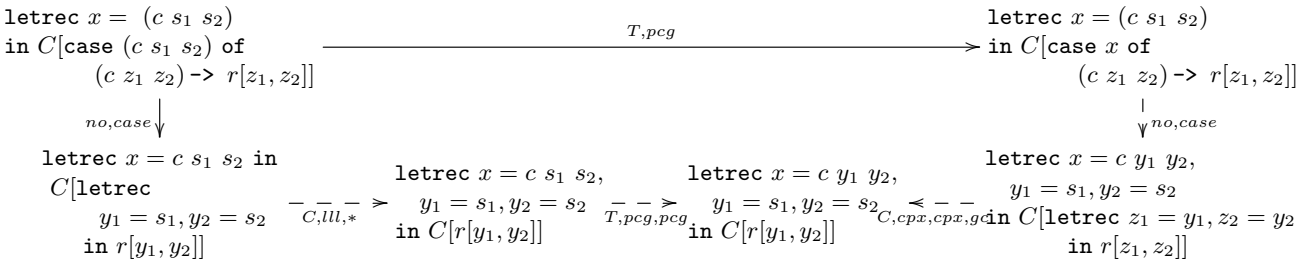
(e) Typical cases for diagram (12)



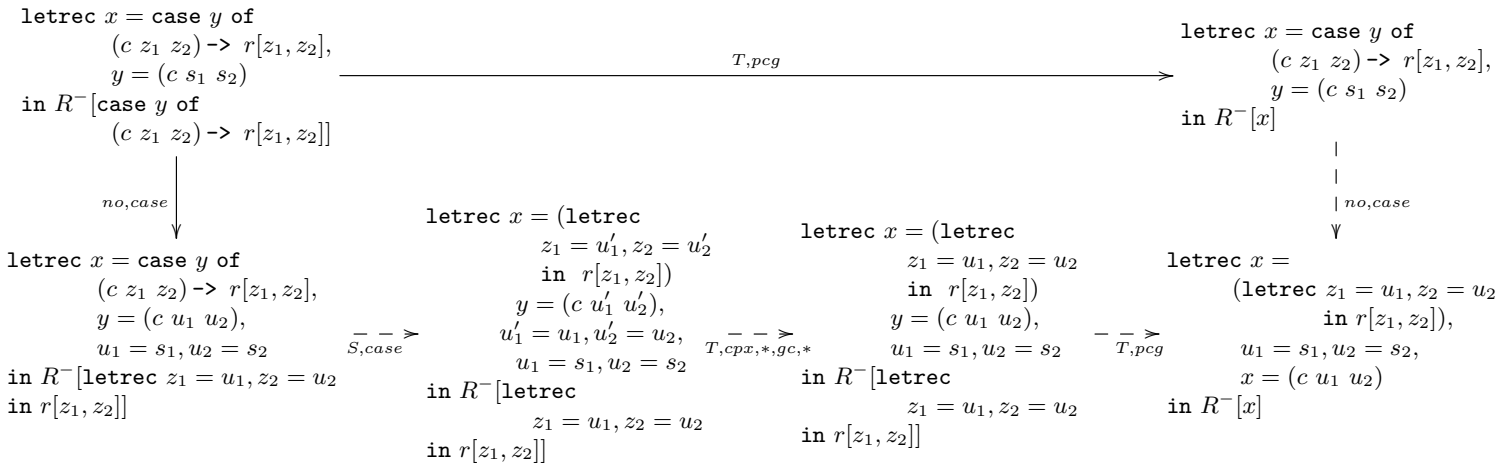
(f) Typical case is for diagram (13)



(g) Typical case for diagram (14)



(h) Typical case for diagram (15)



(i) Typical case for diagram (16)

Fig. 13: Typical cases for the diagrams for (pcg), cont'd.