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
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 \texttt{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 \texttt{letrec}s) 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 $\mathit{Var}$ 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 $\mathit{ar}(K) \geq 0$, 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 $\mathit{ar}(c_{K,i}) \geq 0$. The syntax of expressions $r,s,t \in \mathit{Expr}$ of LR is defined in Fig. 1a. We write $\mathit{FV}(s)$ for the set of free variables of an expression $s$.

Besides variables $x$, abstractions $\lambda x.s$, and applications $(t s)$ the syntax of LR comprises the following constructs: Constructor applications $(c_{K,i} s_1 \ldots s_{\mathit{ar}(c_{K,i})})$ are only allowed to occur fully saturated. We sometimes omit the index of the constructor or use vector notation and thus write e.g. $(c \seq{\vec{x}})$ instead of $(c_{K,i} s_1 \ldots s_{\mathit{ar}(c_{K,i})})$.

In a letrec-expression $\mathit{letrec} \ x_1 = s_1, \ldots, x_n = s_n \ \mathit{in} \ t$ 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 $\mathit{Env}$ (e.g. we write $(\mathit{letrec} \ \mathit{Env} \ \mathit{in} \ s)$, if the exact syntax of the bindings is not relevant). In an environment $\mathit{Env} = \{x_1 = t_1, \ldots, x_n = t_n\}$, we define $\mathit{LV}(\mathit{Env}) = \{x_1, \ldots, x_n\}$ and we sometimes write $\{x_i = t_i\}_{i=1}^{m}$ as abbreviation for such an environment. We also use $\mathit{Env}$ for parts of the environment like e.g. in $\mathit{letrec} \ \mathit{Env}_1, \mathit{Env}_2 \ \mathit{in} \ s$. 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 $\{x_i = x_{i-1}\}_{i=1}^{m}$.

In a seq-expression $(\mathit{seq} \ s t)$ 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 $(\mathit{case}_K s \ of \ ((c_{K,1} x_1 \ldots x_{\mathit{ar}(c_{K,1})}) \rightarrow t_1) \ldots ((c_{K,|D_K|} x_1 \ldots x_{\mathit{ar}(c_{K,|D_K|})}) \rightarrow t_{|D_K|}))$ where there is a case $\mathit{case}_K$ for every type constructor $K$. A case-expression has exactly one case-alternative $((c_{K,i} x_1 \ldots x_{\mathit{ar}(c_{K,i})}) \rightarrow t_i)$ for every data constructor $c_{K,i} \in D_K$. The variables $x_1, \ldots, x_{\mathit{ar}(c_{K,i})}$ in the case-pattern $((c_{K,i} x_1 \ldots x_{\mathit{ar}(c_{K,i})}) \rightarrow t_i)$ must be pairwise distinct and the scope of the variables $x_1, \ldots, x_{\mathit{ar}(c_{K,i})}$ is the expression $t_i$. We sometimes use the meta-symbol alts to abbreviate the case-alternatives and thus e.g. write $(\mathit{case}_K s \ of \ alts)$.

Definition 2.1. A context $C$ is an expression with a hole (denoted by $[]$) at expression position. Surface contexts $S$ are contexts where the hole is not in an abstraction, top contexts $T$ are surface
contexts where the hole is not in an alternative of a case, and weak top contexts are top contexts 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 $(c \overline{s})$. 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 $\beta$-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-name 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

\[^{1}\text{However, a syntactic (and rather complex) definition of reduction contexts by a context free grammar and the corresponding description of normal order reduction for the calculus LR can be found in [20].}\]
(gc1) \[(\text{letrec } x_1 = s_1, \ldots, x_n = s_n, \Env \in t) \rightarrow (\text{letrec } \Env \in t)\] if for all \(i: x_i\) does not occur in \(\Env\) nor in \(t\)

(gc2) \[(\text{letrec } x_1 = s_1, \ldots, x_n = s_n \in t) \rightarrow t\] if for all \(i: x_i\) does not occur in \(t\)

(cpx-in) \[(\text{letrec } x = y, \Env \in C[x]) \rightarrow (\text{letrec } x = y, \Env \in C[y])\] where \(y\) is a variable and \(x \neq y\)

(cpx-e) \[(\text{letrec } c, x = y, z = C[x], \Env \in t) \rightarrow (\text{letrec } x = y, z = C[y], \Env \in t)\] where \(y\) is a variable and \(x \neq y\)

(cpxax) \[(\text{letrec } x = y, \Env \in s) \rightarrow (\text{letrec } x = y, \Env[y/x] \in s[y/x])\] where \(y\) is a variable, \(x \neq y\) and \(y \in \mathbf{FV}(s, \Env)\)

(cpx-in) \[(\text{letrec } x = c \overline{T}, \Env \in C[x]) \rightarrow (\text{letrec } x = c \overline{T}, y_1 = t_1, \ldots, y_n = t_{ar(c)}, \Env \in C[c \overline{T}])\]

(cpx-e) \[(\text{letrec } x = c \overline{T}, z = C[x], \Env \in t) \rightarrow (\text{letrec } x = c \overline{T}, y_1 = t_1, \ldots, y_{ar(c)} = t_{ar(c)}, z = C[c \overline{T}], \Env \in t)\]

(abs) \[(\text{letrec } x = t, \Env \in s) \rightarrow (\text{letrec } x = t, \Env \in s)\] where \(ar(c) \geq 1\)

(abse) \[(c \overline{T}) \rightarrow (\text{letrec } x_1 = t_1, \ldots, x_{ar(c)} = t_{ar(c)} \in c \overline{T})\] where \(ar(c) \geq 1\)

(chx) \[(\text{letrec } x = t, y = x, \Env \in r) \rightarrow (\text{letrec } y = t, x = y, \Env \in r)\]

(ucp1) \[(\text{letrec } \Env, x = t \in S[x]) \rightarrow (\text{letrec } \Env \in S[t])\]

(ucp2) \[(\text{letrec } \Env, x = t, y = S[x] \in r) \rightarrow (\text{letrec } \Env, y = S[t] \in r)\]

(ucp3) \[(\text{letrec } x = t \in S[x]) \rightarrow S[t]\]

where in the (ucp)-rules, \(x\) has at most one occurrence in \(S[x]\) and no occurrence in \(\Env, t, r\) and \(S\) is a surface context

Fig. 3: Extra transformation rules

means reduction of the top term, \textbf{sub} means reduction of a subterm, \textbf{vis} marks already visited subexpressions, and \textbf{nontarg} marks already visited variables that are not target of a (cp)-reduction. For a term \(s\) the labeling algorithm starts with \(s^{\text{top}}\), 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 \textbf{sub}-labeled \texttt{letrec}-expressions. If the labeling algorithm does not fail, then a potential normal order redex is found, which can only be a superterm of the \textbf{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 \(t \xrightarrow{\text{no}} t'\) 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 \textbf{sub}, \textbf{vis} must match the labels in the expression \(t\) (it 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 \(\alpha\)-equivalence) s.t. \(t \xrightarrow{\text{no}} t'\).

We sometimes attach more information to the reduction arrow, e.g. \(t \xrightarrow{\text{no,lbeta}}\) denotes a normal order reduction using rule (lbeta). For a binary relation \(\rightarrow\) we write \(\downarrow\) for the transitive closure, and \(\ast\) for the reflexive-transitive closure of \(\rightarrow\). E.g., \(t \xrightarrow{\text{no,s}}\) denotes the reflexive-transitive closure of \(t \xrightarrow{\text{no}}\). We write \(n\rightarrow\) for exactly \(n\) \(\rightarrow\)-steps and we write \(n/m\rightarrow\) for either \(n\) or \(m\) steps. The notation \(a/v\rightarrow\) is also used for \(a\) and \(b\) being rule names, meaning the union of the rules \(a\) and \(b\). For instance, \(\xrightarrow{\text{no,lbeta}}\) also 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 \textbf{sub} or \textbf{top} by the labeling algorithm in Fig. 1b. A weak reduction context, \(R^-\), is a reduction context, where the hole is not within a \texttt{letrec}-expression.

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

An expression \(s\) converges, denoted as \(s\downarrow\), iff there exists a WHNF \(t\) s.t. \(s \xrightarrow{\text{no,s}} t\). 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 (lseq) is the union of (lseq), (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); (cpax) is the union of (cpax-in) and (cpax-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 $\xrightarrow{\text{no,llet}}$ for $\xrightarrow{\text{no,llet-in}} \lor \xrightarrow{\text{no,llet-e}}$.

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 (cpax) 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), (lseq), (gc), (cpx), (cpax), (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. $\text{rln}(t)$ is the number of lbeta, case, seq-reductions in $t \downarrow t_0$.
2. $\text{rlnall}(t)$ is the number of all reductions in $t \downarrow t_0$. 


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

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

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 

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

Let $t$ be a closed LR-expression with $t \downarrow t_0$.

1. If $t \xrightarrow{C_a} t'$, and $a \in \{\text{case, seq, lbeta, cp}\}$, then $\text{rln}(t) \geq \text{rln}(t')$ and $\text{rlnall}(t) \geq \text{rlnall}(t')$.

2. If $t \xrightarrow{S_a} t'$, and $a \in \{\text{case, seq, lbeta}\}$, then $\text{rln}(t) \geq \text{rln}(t') \geq \text{rln}(t) - 1$.

3. If $t \xrightarrow{C_a} t'$, and $a \in \{\text{lll, gc}\}$, then $\text{rln}(t) = \text{rln}(t')$ and $\text{rlnall}(t) \geq \text{rlnall}(t')$. For $a = \text{gcl}$ the equation $\text{rlnall}(t) = \text{rlnall}(t')$ holds.

4. If $t \xrightarrow{C_{acp}} t'$, and $a \in \{\text{cpx, cpax, xch, cpcx, abs}\}$, then $\text{rln}(t) = \text{rln}(t')$ and $\text{rlnall}(t) = \text{rlnall}(t')$.

5. If $t \xrightarrow{C_{acp}} t'$, then $\text{rln}(t) = \text{rln}(t')$ and $\text{rlnall}(t) \geq \text{rlnall}(t')$.

3.1 The Improvement Relation

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

**Definition 3.3 (Improvement Relation).** For $s, t \in \text{Expr}$ 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: $\text{rln}(C[s]) \leq \text{rln}(C[t])$. We write $t \succeq s$ if $s \preceq t$ holds. If $s \preceq t$ and $s \succeq t$, we write $s \simeq t$.

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 $s \bowtie_{\eta,R} t$ iff for all $X$-contexts $X$, s.t. $X[s], X[t]$ are closed: $\text{rln}(X[s]) \bowtie_{\eta,C} \text{rln}(X[t])$. In particular, $\bowtie_{\leq,C} = \leq, \bowtie_{\geq,C} = \geq$, and $\bowtie_{=} = \approx$.

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: $\text{rln}(M[s_1, \ldots, s_n]) \bowtie_{\eta,C} \text{rln}(M[t_1, \ldots, t_n])$. 
The proof is by induction on the pair \((k, k')\) where \(k\) is the number of normal order reductions of \(M[s_1, \ldots, s_n]\) to a WHNF, and \(k'\) is the number of holes of \(M\). If \(M\) (without holes) is a WHNF, 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 \(\text{rln}(M[s_1, \ldots, s_n]) = 0 = \text{rln}(M[t_1, \ldots, t_n])\).

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 \(M[t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n]\) is a reduction context. By the induction hypothesis, using the multi-context assumption we have \(\text{rln}(M[s_1, \ldots, s_n]) \subseteq \text{rln}(M[t_1, \ldots, t_n])\).

If in \(M[s_1, \ldots, s_n]\) there is no \(s_i\) in a reduction context, then \(M[s_1, \ldots, s_n] \xrightarrow{\text{no,a}} M'[s'_1, \ldots, s'_n]\), may copy or shift some of the \(s_i\) where \(s'_j = \rho(s_i)\) for some variable permutation \(\rho\). 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. \(M[t_1, \ldots, t_n] \xrightarrow{\text{no,a}} M'[t'_1, \ldots, t'_n]\) with \((s'_j, t'_j) = (\rho(s_i), \rho(t_i))\). We take for granted that the renaming can be carried through. The \(\text{rln}(t)\)-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 \(\text{rln}(M[s_1, \ldots, s_n]) = m + \text{rln}(M'[s'_1, \ldots, s'_n])\) or \(\text{rln}(M[t_1, \ldots, t_n]) = m + \text{rln}(M'[t'_1, \ldots, t'_n])\).

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 \(s \triangleright_{\eta,T} t\) (or \(s \triangleright_{\eta,S} t\)) implies that \(s \triangleright_{\eta,C} t\).

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 \(t \xrightarrow{\text{cp}} t'\) then \(\text{rln}(t) = \text{rln}(t')\).

**Proof.** This follows using correctness of \((cp)\) and the diagrams in [20, Lemmas B.8, B.9] where forking diagrams for \(t \xrightarrow{\text{no,a}} s \xrightarrow{\text{cp}} t'\) are computed. Then counting the reductions contributing to \(\text{rln}(\cdot)\) 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 \((\text{case}), (\text{seq}), (\text{lbeta}), (\text{cp}), (\text{lll}), (\text{gc}), (\text{cpz}), (\text{cpax}), (\text{xch}), (\text{abs}), (\text{ucp})\) are improvements.

Moreover, for \(a \in \{(cp), (\text{lll}), (\text{gc}), (\text{cpz}), (\text{cpax}), (\text{xch}), (\text{abs}), (\text{ucp})\}\) the inclusion \(a \subseteq \leq\) and thus the inclusion \(a \subseteq \approx\) holds.
4 Common Subexpression Elimination

Common subexpression elimination (cse) can be expressed as:

\[(\text{cse}) \, M[s, \ldots, s] \rightarrow \text{letrec} \, x = s \text{ in } M[x, \ldots, x]\]

where \(x\) is a fresh variable and the multicontext \(M\) does not capture a variable in \(s\).

We will show that \((\text{cse})\) 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 \((\text{cse})\) 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 \((\text{cse})\) \(\subseteq\) \(\geq\) we consider the general-copy rule

\[(\text{gcp}) \, \text{letrec} \, x = s \text{ in } C[x] \rightarrow \text{letrec} \, x = s \text{ in } C[s].\]

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

**Definition 4.1.** The transformation \((\text{pgc})\) is the union of the rules:

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

The transformation \((\text{pgcE})\) is the union of the following rules:

\[(\text{pgcE}_1\text{-in}) \, \text{letrec} \, \text{Env}, \text{Env}_2 \text{ in } C[\text{letrec} \, \text{Env}', \text{Env}_3 \text{ in } r] \rightarrow \text{letrec} \, \text{Env}, \text{Env}_2 \text{ in } C[\text{letrec} \, \text{Env}_3 \alpha \text{ in } r\alpha]\]
\[(\text{pgcE}_2\text{-in}) \, \text{letrec} \, \text{Env}, \text{Env}_2 \text{ in } C[\text{letrec} \, \text{Env}' \text{ in } r] \rightarrow \text{letrec} \, \text{Env}, \text{Env}_2 \text{ in } C[r\alpha]\]
\[(\text{pgcE}_1\text{-e}) \, \text{letrec} \, \text{Env}, \text{Env}_2, x = C[\text{letrec} \, \text{Env}', \text{Env}_3 \text{ in } r] \text{ in } s \rightarrow \text{letrec} \, \text{Env}, \text{Env}_2, x = C[\text{letrec} \, \text{Env}_3 \alpha \text{ in } r\alpha] \text{ in } s\]
\[(\text{pgcE}_2\text{-e}) \, \text{letrec} \, \text{Env}, \text{Env}_2, x = C[\text{letrec} \, \text{Env}' \text{ in } r] \rightarrow \text{letrec} \, \text{Env}, \text{Env}_2, x = C[r\alpha] \text{ in } s\]
\[(\text{pgcE}_3) \, \text{letrec} \, \text{Env}, \text{Env}', \text{Env}_3 \text{ in } r \rightarrow \text{letrec} \, \text{Env}, \text{Env}_3 \alpha \text{ in } r\alpha\]

where \(\text{Env}'\alpha = \text{Env}\) and \(\alpha\) only renames variables of \(\text{LV}(\text{Env}')\).

**Proposition 4.2.** \((\text{pgc})\) and \((\text{pgcE})\) are correct.

**Proof.** In [19] the following result for LR was obtained:

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

Now, since \(\text{IT}(s) = \text{IT}(t)\), correctness of both program transformation holds.
Proposition 4.3. If \( s \xrightarrow{T_{pcgE}} t \), then \( \text{rln}(s) \geq \text{rln}(t) \) and \( \text{rlnall}(s) \geq \text{rlnall}(t) \). Moreover, the transformation \((pcgE)\) is an improvement, i.e. \((pcgE) \subseteq \geq\).

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 \( \text{rlnall}(s) \) we show that \( \text{rlnall}(t) \leq \text{rlnall}(s) \) and \( \text{rln}(t) \leq \text{rln}(s) \). If \( \text{rlnall}(s) = 0 \) then \( s \) is a WHNF, and \( t \) must also be a WHNF and \( \text{rlnall}(t) = \text{rln}(t) = 0 \). If \( \text{rlnall}(s) > 0 \), then let \( s \xrightarrow{no,a} s' \).

- For diagram (1) we can apply the induction hypothesis to \( s' \xrightarrow{T_{pcgE}} t' \), since \( \text{rlnall}(s') = \text{rlnall}(s) - 1 \). This shows that \( \text{rlnall}(t) \leq \text{rlnall}(s) \) and also that \( \text{rln}(t) \leq \text{rln}(s) \).

- For diagram (2) we can apply the induction hypothesis to \( s' \xrightarrow{T_{pcgE}} t \) which shows \( \text{rlnall}(t) \leq \text{rlnall}(s') = \text{rlnall}(s) - 1 \) and \( \text{rln}(t) \leq \text{rln}(s') = \text{rln}(s) \).

- For diagram (3) we have \( \text{rlnall}(s) > \text{rlnall}(s') \) and \( \text{rln}(s) = \text{rln}(s') \). By Theorem 3.2 (3) \( \text{rlnall}(s'') \leq \text{rlnall}(s') \) and \( \text{rln}(s'') \leq \text{rln}(s') \). We can apply the induction hypothesis to \( s'' \xrightarrow{T_{pcgE}} t' \) which shows \( \text{rlnall}(t'') \leq \text{rlnall}(s'') \) and \( \text{rln}(t') \leq \text{rlnall}(s'') \). This implies \( \text{rlnall}(t) \leq \text{rlnall}(s) \) and also \( \text{rln}(t) \leq \text{rln}(s) \).

- For diagram (4) we have \( \text{rlnall}(s') < \text{rlnall}(s) \) and \( \text{rln}(s') \leq \text{rln}(s) \) (or \( \text{rln}(s') < \text{rln}(s) \) if \( a \in \{\text{case, beta, seq}\} \)). Theorem 3.2 shows that \( \text{rlnall}(s'') \leq \text{rlnall}(s') \) and \( \text{rln}(s'') \leq \text{rln}(s') \).

- For diagram (5) we have \( \text{rlnall}(s) = \text{rlnall}(t) \) and \( \text{rln}(s) = \text{rln}(t) \). By Theorem 3.2 (4) we have \( \text{rln}(s'') \leq \text{rln}(s') \) and \( \text{rln}(s'') \leq \text{rln}(s) \). Applying the induction hypothesis to \( s'' \xrightarrow{T_{pcgE}} t' \) yields \( \text{rlnall}(t'') \leq \text{rlnall}(s'') \) and \( \text{rln}(t'') \leq \text{rln}(s'') \). This implies both \( \text{rlnall}(t) \leq \text{rlnall}(s) \) as well as \( \text{rln}(t) \leq \text{rln}(s) \).

- For diagram (6) we have \( \text{rlnall}(s') < \text{rlnall}(s) \) and \( \text{rln}(s') = \text{rln}(s) \). Applying the induction hypothesis to \( s' \xrightarrow{T_{pcgE}} s'' \) yields \( \text{rlnall}(s'') \leq \text{rlnall}(s') \) and \( \text{rln}(s'') \leq \text{rln}(s') \). Applying the induction hypothesis to \( s'' \xrightarrow{T_{pcgE}} t' \) yields \( \text{rlnall}(t'') \leq \text{rlnall}(s'') \) and \( \text{rln}(t'') \leq \text{rln}(s'') \). Since \( t \xrightarrow{no,a} t' \), this shows \( \text{rlnall}(t) \leq \text{rlnall}(s) \) and \( \text{rln}(t) \leq \text{rln}(s) \).

- For diagram (7) we have \( \text{rln}(s'') < \text{rln}(s) \) and \( \text{rln}(s) < \text{rln}(s') \). By Theorem 3.2 (4) we have \( \text{rln}(s'') \leq \text{rln}(s') \) and \( \text{rln}(s'') \leq \text{rln}(s) \). Applying the induction hypothesis to \( s'' \xrightarrow{T_{pcgE}} t' \) yields \( \text{rlnall}(t'') \leq \text{rlnall}(s'') \) and \( \text{rln}(t'') \leq \text{rln}(s'') \) which shows \( \text{rlnall}(t) \leq \text{rlnall}(s) \) and \( \text{rln}(t) \leq \text{rln}(s) \).

- For diagram (8) we have \( \text{rlnall}(s') < \text{rlnall}(s) \) and \( \text{rln}(s') < \text{rln}(s) \). By Theorem 3.2 (1), (3), (4) we have \( \text{rln}(s'') \leq \text{rln}(s) \) and \( \text{rln}(s'') = \text{rln}(s) \). Applying the induction hypothesis to \( s'' \xrightarrow{T_{pcgE}} t' \) yields \( \text{rlnall}(t'') \leq \text{rlnall}(s'') \) and \( \text{rln}(t'') \leq \text{rln}(s'') \) which shows \( \text{rlnall}(t) \leq \text{rlnall}(s) \) and \( \text{rln}(t) \leq \text{rln}(s) \).

Since \((pcgE)\) is correct (Proposition 4.2), the context lemma for improvement (Corollary 3.5) shows \((pcgE) \subseteq \geq\).
Lemma 4.4. If \( s \xrightarrow{T,p cg} s' \), where \( s \) is a WHNF, then either \( s' \) is a WHNF, or \( s' \xrightarrow{\text{no,cp}} s'' \) where \( s'' \) is a WHNF.

Lemma 4.5. For closed \( s \) with \( s \xrightarrow{T,p cg} s' \), \( \text{rln}(s) \geq \text{rln}(s') \).

Proof. Let \( s_1 \), and \( s \xrightarrow{T,p cg} s' \). We show \( \text{rln}(s) \geq \text{rln}(s') \) by induction on the measure \((\text{rln}(s), \text{rlnall}(s))\), ordered lexicographically. For the base case \( \text{rln}(s) = \text{rlnall}(s) = 0 \), the expression \( s \) is a WHNF and Lemma 4.4 shows that \( \text{rln}(s') = 0 \).

For the induction step, Fig. 5 shows the overlappings between normal order reduction-steps and a \( T,p cg \)-transformation (see Appendix C.2 for details) where the cases that the \((T,p cg)\)-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:

![Fig. 6: Machine transitions](image)

\begin{enumerate}
  \item If \( s \xrightarrow{\text{no,case} \lor \text{seq} \lor \text{ifbeta}} t_1 \), then one of the diagrams (9), (12), (13), (15), or (16) of Fig. 5 holds. The diagrams can be summarized as follows where \( a \in \{\text{case}, \text{seq}, \text{ifbeta}\} \):

  \[
  \begin{array}{c}
  s \\
  \xrightarrow{T,p cg} \\
  \xrightarrow{\text{no,a}} \\
  t_1 \\
  \xrightarrow{T,p cg \ast} \\
  t_2 \\
  \xrightarrow{T,p cg \ast} \\
  t_3 \\
  \xrightarrow{T,p cg \ast} \\
  t_4 \\
  \xrightarrow{T,p cg \ast} \\
  t_5
  \end{array}
  \]

  We have \( \text{rln}(t_1) = \text{rln}(s) - 1 \) and by Theorem 3.2 we have \( \text{rln}(t_3) \leq \text{rln}(t_1) \). We apply the induction hypothesis for every step in \( t_3 \xrightarrow{T,p cg \ast} t_4 \) and we derive \( \text{rln}(t_4) \leq \text{rln}(t_1) < \text{rln}(s) \).

  Theorem 3.2 shows that \( \text{rln}(t_4) = \text{rln}(t_3) \), and thus \( \text{rln}(s') = \text{rln}(t_3) + 1 \leq \text{rln}(t_1) + 1 = \text{rln}(s) \).

  \item Let \( s \xrightarrow{\text{no,cp}} t_1 \). If \( t_1 \) is a WHNF, then by Lemma 4.4 \( s' \xrightarrow{\text{no,cp} \ast \text{if}} t_1' \) where \( t_1' \) is a WHNF and \( \text{rln}(s') \leq \text{rln}(s) \) holds. If \( t_1 \) is not a WHNF, then \( t_1 \xrightarrow{\text{no,a}} t_2 \) where \( a \in \{\text{ifbeta}, \text{seq}\} \) and diagram (10) or (11) of Fig. 5 holds.

  For diagram (10) we have:

  \[
  \begin{array}{c}
  s \\
  \xrightarrow{T,p cg} \\
  \xrightarrow{\text{no,cp}} \\
  t_1 \\
  \xrightarrow{T,p cg \ast} \\
  t_2 \\
  \xrightarrow{T,p cg \ast} \\
  t_3 \\
  \xrightarrow{T,p cg \ast} \\
  t_4
  \end{array}
  \]

  Then \( \text{rln}(t_2) < \text{rln}(s) \) and we apply the induction hypothesis to \( t_2 \xrightarrow{T,p cg \ast} t_3 \) which shows \( \text{rln}(t_3) \leq \text{rln}(t_2) < \text{rln}(s) \). We then apply the induction hypothesis to \( t_3 \xrightarrow{T,p cg \ast} t_4 \) which shows \( \text{rln}(t_4) \leq \text{rln}(t_2) < \text{rln}(s) \) and \( \text{rln}(s') \leq \text{rln}(s) \).

  Similarly, in diagram (11) the situation is: \( t_2 \xrightarrow{\text{no,a}} t_1 \xrightarrow{\text{no,cp}} s \xrightarrow{p cg} s' \xightarrow{\text{no,cp}} t'_1 \xrightarrow{\text{no,a}} t_5 \), and \( t_2 \xrightarrow{C,cp} t_3 \xrightarrow{C,a} t_4 \xrightarrow{T,p cg} t_5 \). Then \( \text{rln}(t_2) < \text{rln}(s) \) and by Theorem 3.2 (1) \( \text{rln}(t_1) \leq \text{rln}(t_2) \).

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

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

\[
\begin{array}{c}
\text{s} \\
\xrightarrow{T,pcg} \\
\text{t}_1 \xrightarrow{\text{no,ill}} \text{s} \xrightarrow{T,pcg} \text{t}_2 \xrightarrow{pcg} \text{t}_3
\end{array}
\]

Then \( rln(t_1) = rln(s) \), and \( rlnall(t_1) = rlnall(s) - 1 \).

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

**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 \( \xrightarrow{\text{gc},pcg} \), and since \( \text{(gc)} \subseteq \approx \) 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 \( rln(\cdot) \) and the counting measures used in [9, 4] for their improvement relations. In Theorem 5.11 we show that \( rln(\cdot) \) 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 \( f, g : E \rightarrow \mathbb{N} \), we write \( f \in O(g) \), if there is a constant \( c > 0 \), s.t. for all \( e \in E \) : \( f(e) \leq c \cdot g(e) \).

**Definition 5.1.** Let \( K_1 = (E_1, \sim_1, \text{size}_1, \mu_1) \), \( K_2 = (E_2, \sim_2, \text{size}_2, \mu_2) \) be two calculi with sets of expressions, contextual equivalences, size-measures for expressions, and measures for reduction length of expressions.

Then a translation \( \phi_1 : K_1 \rightarrow K_2 \) is size-preserving, iff \( \text{size}_1(e) \in O(\text{size}_2(\phi(e))) \) and \( \phi \) 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 \( \phi : K_1 \rightarrow K_2 \) is asymptotically resource-preserving, if \( \phi \) is a size-preserving translation such that there exists an \( n \in \mathbb{N} \) with \( \mu_1(e_1) \in O(\text{size}_2(\phi_1(e_1)))^n \ast (\mu_2(\phi_1(e_1)) + 1) \).

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 \( (s t) \), \( \text{seq}-\)expressions\(^2\) \( (\text{seq} s t) \), and constructor applications \( (c t_1 \ldots t_{\text{ar}(c)}) \) the expressions \( t, t_i \) must be variables.

\(^2\) Note that the syntax in [9] does not have \( \text{seq} \)-expressions.
Lemma 5.3. The translation \( \psi \) from arbitrary LR-expressions into machine expressions is

\[
\psi(x) := x, \quad \text{if } x \in \text{Var}
\]

\[
\psi(s_t) := \text{letrec } x = \psi(t) \text{ in } (\psi(s) x)
\]

\[
\psi(\text{seq } s_t) := \text{letrec } x = \psi(t) \text{ in } (\text{seq } \psi(s) x)
\]

\[
\psi(c_{s_1 \ldots s_n}) := \text{letrec } x_1 = \psi(s_1), \ldots, x_n = \psi(s_n) \text{ in } (c_{x_1 \ldots x_n})
\]

\[
\psi(M[s_1, \ldots, s_n]) := M[\psi(s_1), \ldots, \psi(s_n)]
\]

where the multicontext \( M \) is \text{letrec } \( x_1 = [], \ldots, x_n = [] \text{ in } [], \lambda x.[x] \), or \text{case}_K [] of (\text{pat}_1 \rightarrow [\cdot]) \ldots (\text{pat}_n \rightarrow [\cdot]).

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

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

A state \( Q \) of the machine is a tuple \( \langle \Gamma \mid s \mid S \rangle \), where \( \Gamma \) is an environment of bindings (like a letrec-environment), \( s \) is a machine expression, and \( S \) is a stack, with entries \#upd(x), \#app(x), \#seq(x), \#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. \( \langle \emptyset \mid t \mid [] \rangle \xrightarrow{\text{C}} \langle \Gamma \mid s \mid S \rangle \). We define a mapping \( \phi \) from reachable machine states to machine expressions:

\[
\phi(\langle \Gamma \mid s \mid \#upd(x) : S \rangle) = \phi(\langle \Gamma, x = s \mid x \mid S \rangle)
\]

\[
\phi(\langle \Gamma \mid s \mid \#app(x) : S \rangle) = \phi(\langle \Gamma \mid (s x) \mid S \rangle)
\]

\[
\phi(\langle \Gamma \mid s \mid \#seq(x) : S \rangle) = \phi(\langle \Gamma \mid (\text{seq } s x) \mid S \rangle)
\]

\[
\phi(\langle \Gamma \mid s \mid \#case(alts) : S \rangle) = \phi(\langle \Gamma \mid (\text{case } s \text{ of } alts) \mid S \rangle)
\]

\[
\phi(\langle \Gamma \mid s \mid [] \rangle) = \text{letrec } \Gamma \text{ in } s
\]

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

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

Note that the improvement theory in [9] is based on the measure \( \text{mlnall}(\cdot) \), whereas the measure in [4] is \( \text{mlnlook}(\cdot) \).
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 \( \text{rln}(s) = \text{mln}(s) \). With Lemma 5.3 this also implies that for every LR-expression \( s \) the equation \( \text{rln}(s) = \text{mln}(\phi(s)) \) holds.

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

\[
\begin{align*}
\text{Q Lookup/Unwind} & : Q \xrightarrow{\phi} Q' \\
\phi(Q) = \phi(Q') & \quad \phi(Q) \xrightarrow{\text{no,case-c}} s \xrightarrow{\text{T,cpax,* \ T,gc,*}} Q' \\
\text{Update} & : Q \xrightarrow{\phi} Q' \\
\phi(Q) \xrightarrow{\text{T,cp}} s \xrightarrow{\text{T,lll}} & \phi(Q') \\
\phi(Q) \xrightarrow{\text{Letrec}} & \phi(Q') \\
\phi(Q) \xrightarrow{\text{Seq}} & \phi(Q') \\
\phi(Q) \xrightarrow{\text{Subst}} & \phi(Q') \\
\phi(Q) \xrightarrow{\text{T,gc}} & \phi(Q') \\
\end{align*}
\]

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

**Proof.** We consider the sequence of machine transformations from \( \langle \emptyset | s | \emptyset \rangle \) to an accepting state and construct a sequence of \((\text{no,lbeta})\)-, \((\text{no,case-c})\)-, \((\text{no,seq})\)-, \((\text{T,cp})\)-, \((\text{T,cpax})\)-, \((\text{T,lll})\)-, and \((\text{T-gc})\)-transformations from \( s \) to a WHNF.

So let \( Q_0 = \langle \emptyset | s | \emptyset \rangle \xrightarrow{\phi} Q_k \) 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 \( Q_0 \xrightarrow{} Q_1 \) 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 \((\text{no,lbeta})\)-, \((\text{no,case-c})\)-, and \((\text{no,seq})\)-steps is \( n \).

Now iteratively apply Theorems 3.2 and 3.6 from right to left to every transformation which is not a \((\text{cp})\)- or \((\text{cp}, \text{ax})\)- or \((\text{T,lll})\)- or \((\text{T,cpax})\)- or \((\text{T,gc})\)-reduction in a machine order reduction sequence from \( s \) to an MWHNF, and \( \text{rln}_{\text{mo}}(s) = \infty \) otherwise.

**Lemma 5.7.** If the machine expression \( s \) is a WHNF, then either \( s \) is also an MWHNF, or \( s \xrightarrow{\text{no,cpv,}} s' \) where \( s' \) is an MWHNF.

**Lemma 5.8.** Let \( s \xrightarrow{\text{no,a}} t \) where \( a \) is not a \((\text{cp})\), or \( s \xrightarrow{\text{no,cpv,}} s' \xrightarrow{\text{no,seq/lbeta}} t \). The diagrams in Fig. 8 show how at least one machine order reduction can be performed for \( s \), s.t. \( s \xrightarrow{\text{mo,+}} r \) and how \( t \) and \( r \) are joinable by program transformations.

**Proposition 5.9.** Let \( s \) be a closed machine expression with \( \text{rln}(s) = n \). Then \( \text{rln}_{\text{mo}}(s) = n \).

**Proof.** Let \( s \xrightarrow{\text{no,k}} t_k \) where \( t_k \) is a WHNF. We use induction on \( (\text{rln}(s), \text{rlnall}(s)) \) to show the claim. If \( \text{rlnall}(s) = 0 \), then \( s \) is a WHNF, and \( s \xrightarrow{\text{no,cpv,}} s' \) where \( s' \) is an MWHNF and thus
Improvements in a Functional Core Language with Call-By-Need Operational Semantics

\[ \text{Fig. 8: Diagrams for transferring normal order reductions into machine order reductions} \]

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

For diagram (20) we have \( \text{rln}(t) = \text{rln}(s) \) and \( \text{rlnall}(t) < \text{rlnall}(s) \). By Theorem 5.13 (3) \( \text{rln}(r) = \text{rln}(s) \) and \( \text{rlnall}(r) < \text{rlnall}(s) \). We apply the induction hypothesis to \( r \) and get \( \text{rln}_{\text{mo}}(r) = \text{rln}(s) \) and thus \( \text{rln}_{\text{mo}}(s) = \text{rln}(s) \).

If diagram (18), (19), or (17) is applied, then \( \text{rln}(t) = \text{rln}(s) - 1 \) and Theorem 5.12 shows that \( \text{rln}(r) = \text{rln}(t) \). Applying the induction hypothesis to \( r \) shows \( \text{rln}_{\text{mo}}(r) = \text{rln}(s) - 1 \).

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

\[ \square \]

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

Proof. From \( \text{rln}(s) = n \) we get \( \text{rln}_{\text{mo}}(s) = n \) by Proposition 5.9. Let \( s \xrightarrow{\text{no,k}} s' \) 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. \( Q_0 \xrightarrow{\text{s}} Q_m \) and \( \text{mln}(Q_0) = n \). If \( k = 0 \), then \( Q_0 \xrightarrow{\text{Letrec,0|1}} Q' \) where \( Q' \) is accepting. For \( k > 0 \), let \( s \xrightarrow{\text{no}} s_0 \xrightarrow{\text{no,k-1}} s' \). The following diagrams (where \( \text{UL} \) is \( \text{(Unwind)} \lor (\text{Lookup}) \)) show the relationship between \( s \xrightarrow{\text{no}} s_0 \) and the machine transition for \( Q_0 \):

\[ \begin{align*}
\phi^1 & : Q_0 \xrightarrow{\text{Update}} Q_1 \\
\phi^2 & : Q_0 \xrightarrow{\text{Subst}} Q_1 \\
\phi^3 & : Q_0 \xrightarrow{\text{Seq}} Q_1 \\
\phi^4 & : Q_0 \xrightarrow{\text{Branch}} Q_1 \\
\phi^5 & : Q_0 \xrightarrow{\text{Subst}} Q_1 \\
\phi^6 & : Q_0 \xrightarrow{\text{Seq}} Q_1 \\
\phi^7 & : Q_0 \xrightarrow{\text{Branch}} Q_1 \\
\phi^8 & : Q_0 \xrightarrow{\text{Seq}} Q_1
\end{align*} \]

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

By Propositions 5.6 and 5.10 and Lemma 5.3 we have:

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

Corollary 5.12. The translation \( \psi \) seen as a translation from \( \text{LR} \) to the abstract machine of [9] in the variant presented in this paper is fully-abstraction.

5.2 Relating Essential and All Transition Steps

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

Theorem 5.13. Let \( s \) be a closed machine expression with \( s \downarrow \). Then \( \text{mlnall}(s) \leq 3 \times (\text{size}(s) + 2) \times (\text{mln}(s) + 1) \), where \( \text{size}(\cdot) \) is the size of an expression (viewed as syntax tree).
Proof. Let \( \text{mln}(s) = n \) and \( Q_0 = (\emptyset \mid s \mid []) \xrightarrow{m^*} Q_m \), where \( Q_m \) is an accepting state and the sequence contains \( n \) (SBS)-transitions.

The number of (Update)-transitions is equal to the number of (Lookup)-transitions. The number of (Unwind)-transitions is equal to the number of (SBS)-transitions. It remains to count the (Letrec)- and the (Lookup)-transitions. First observe that letrec-expressions and -bindings which are generated by an (SBS)-transition are a copy of a subexpression which exists in \( s \) (where variables may be permuted). Since a (Letrec)-transition removes the letrec, there are at most \((n+1) \cdot \text{size}(s)\) (Letrec)-transitions. The same argument applies to the first (Lookup)-transitions of a binding \( x = \ldots \). The number of other (Lookup)-transitions (which are not the first for a binding \( x = \ldots \)) is bounded by \( n + 1 \), since for such a (Lookup) the binding must be \( x = v \), where \( v \) is a value, which implies, that no other (Lookup) transition can follow before another (SBS)-transition is performed. Thus, in total there are at most \((n+1) \cdot \text{size}(s) + 1\) (Lookup)-transitions.

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

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 \( \text{mlnall}(s) \leq (2 \cdot \text{size}(s) \cdot (\text{mlnlook}(\phi(s)) + 1)) \).

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

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

**Remark 5.16.** Theorem 5.15 justifies our claim that common subexpression elimination (also called \( \beta \)-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. \( \text{mln}(.) \) 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 \( \text{rln} \); (ii) Moran-Sands calculus with \( \text{mlnall} \); (iii) Moran-Sands calculus with \( \text{mln} \); and (iv) the Moran-Sands calculus with \( \text{mlnlook} \).

Note that in LR switching from \( \text{rln}(.) \) to \( \text{rlnall}(.) \) is not resource-preserving, since there are LR-expressions \( s \) s.t. \( \text{rlnall}(s) \in O(\text{size}(s)^n(\text{rln}(s)+1)) \) 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: \(a, a_1 \in TVar\)  
Types: \(\tau \in Typ := a | (\tau_1 \rightarrow \tau_2) | K \tau_1 \ldots \tau_{ar(K)}\)

Term variables: \(x, x_i \in \text{Var}\)  
Polymorphic types: \(\rho \in TTyp := \tau | \Lambda a. \rho\)

Patterns: \(\text{pat}_{K,i} := (c_{K,i} : \tau_1 : x_1 : \ldots : x_{ar(c_{K,i})) : \tau_{ar(c_{K,i}))})\)  
Polymorphic abstracts \(\in PE\xrightarrow{\rho} := Aa_1, \ldots, Aa_k, \Lambda x.s\)

Expressions: \(s, t \in \text{Expr}_\rho := u | x : \rho | (s \tau) | (c_{K,i} : \tau s_1 \ldots s_{ar(c_{K,i}))} | (\text{seq} s t) \mid (\text{letrec} x_1 : \rho_1 = s_1, \ldots, x_n : \rho_n = s_n \in t) \mid (\text{caseK} \ s \ of \ (\text{pat}_{K,1} \rightarrow t_1) \ldots (\text{pat}_{K,|D_K|} \rightarrow t_{|D_K|}))\)

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

\[
\begin{align*}
\forall \tau_1 : \tau_2 & \quad (\lambda x : \tau_1.s) \rightarrow \tau_2 \\
\forall \tau_1 : \tau_2 & \quad \lambda a. \rho : \tau_1 \rightarrow \tau_2 \\
\forall \tau_1 : \tau_2 & \quad (s \tau) \rightarrow \tau_1 \\
\forall \tau_1 : \tau_2 & \quad (s t) \rightarrow \tau_1 \\
\forall \tau_1 : \tau_2 & \quad \text{typeof} \ s \rightarrow \tau_1 \\
\forall \tau_1 : \tau_2 & \quad \text{typeof} \ t \rightarrow \tau_1 \\
\end{align*}
\]

(b) Typing Rules for LRP

\[
\begin{align*}
(s t)^\text{sub\top} & \rightarrow (s^\text{sub} t^\text{vis}) \quad (s \neq Aa. e') \\
((Aa. u)^\text{sub\top} & \rightarrow ((Aa. u)^\text{vis}) \quad \text{then stop with success} \\
\text{letrec Env in s}^\text{top} & \rightarrow \text{letrec Env in s}^\text{vis} \\
\text{letrec x = s, Env in C[x^\text{sub}]} & \rightarrow \text{letrec x = s^\text{sub}, Env in C[x^\text{vis}]} \\
\text{letrec x = s, y = C[x^\text{sub}], Env in t} & \rightarrow \text{letrec x = s^\text{sub}, y = C[x^\text{vis}], Env in t} \quad \text{where C \neq [\]} \\
\text{letrec x = s, y = x^\text{sub}, Env in t} & \rightarrow \text{letrec x = s^\text{sub}, y = x^\text{nontarg}, Env in t} \\
\text{seq s t}^\text{sub\top} & \rightarrow \text{seq s t}^\text{vis} \\
\text{letrec x = s^\text{nontarg}, y = C[x^\text{sub}], Env in t} & \rightarrow \text{Fail} \\
\text{letrec x = C[x^\text{sub}], Env in t} & \rightarrow \text{Fail} \\
\end{align*}
\]

(c) Computing reduction positions using labels in LRP, where \(a \lor b\) 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 \(\text{rln}(\cdot)\)-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 equivalences 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 \(\text{type}(c)\) of the form \(\lambda a_1, \ldots, a_k. \tau_1 \rightarrow \ldots \tau_{ar(c)} \rightarrow K(a_1, \ldots, a_k)\). The typing rules are in Fig. 9b- All expressions of a polymorphic type \(\lambda a. \rho\) are of the form \(x : \rho, \Lambda e, (c \tau)\), and \(\text{letrec Env in t}\), 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:

\[
(T\beta) \quad ((\lambda a. u)^\text{sub} \tau) \rightarrow u[\tau/a]
\]

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.

\[
\text{letrec } x_1 = u^{\text{sub}}, \{x_i = x_{i-1}\}_{i\geq 2}^m, \text{Env in } C[x_m'^{\text{vis}}] \\
\rightarrow (\text{letrec } x_1 = u, \{x_i = x_{i-1}\}_{i\geq 2}^m, \text{Env in } C[u]), \text{ where } u \text{ is an abstraction or a polymorphic abstraction.}
\]

The rules of the labeling algorithm are in Fig. 9c. If the labeling algorithm terminates without \textbf{Fail}, then either a normal order redex is found, which is a superterm of the \textbf{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 \( \frac{\text{LRP}}{} \) 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 \textbf{sub}, \textbf{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 \((\text{letrec Env in } v)\), where \( v \) is a value, or an expression of the form \((\text{letrec } x_1 = (c \ t), \{x_i = x_{i-1}\}_{i\geq 2}^m, \text{Env in } x_m)\).

An LRP-expression \( s \) converges, denoted as \( s \downarrow \), iff there exists a WHNF \( t \) such that \( s \xrightarrow{\text{LRP}*} t \). Let \( s, t \) be two LRP-expressions of the same type \( \rho \). 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 \( s :: \lambda a. \rho \sim_c t :: \lambda a. \rho \), then also \((s :: \lambda a. \rho)[t :: \lambda a. \rho] \sim_c (t :: \lambda a. \tau) \).

**Definition 6.4.** The type erasure function \( \varepsilon : \text{LRP} \rightarrow \text{LR} \) maps LRP-expressions to LR-expressions by removing the types, the type information and the \( \Lambda \)-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 \( \varepsilon \) to LR-normal-order reductions where exactly the \( (T\beta) \)-reductions are omitted.

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

The translation \( \varepsilon \) is not fully abstract. An example are the equivalences by \( (\text{caseId}) \), see below.
6.2 Improvement in LRP

The measure for estimating the time consumption of computation also in LRP is $rln(t)$. 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 daggs 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 $\rho$. We define the improvement relation $\preceq$ for LRP: Let $s \preceq t$ iff $s \sim_c t$ and for all contexts $C[: \rho]$; if $C[s],C[t]$ are closed, then $rln(C[s]) \leq rln(C[t])$. If $s \preceq t$ and $t \preceq s$, we write $s \simeq t$.

The following facts are valid and can easily be verified:
1. For closed LRP-expressions $s$, the equation $rln(s) = rln(\varepsilon(s))$ 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 $\rho$ the relation $s \triangleright_{\eta,X} t$ (in LRP) holds iff for all $X$-contexts $X[:: \rho]$; if $X[s],X[t]$ are closed, then $rln(X[s]) \eta rln(X[t])$. 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 $\rho$. Then $s \triangleright_{\eta,R} t$ (or $s \triangleright_{\eta,S} t$ or $s \triangleright_{\eta,T} t$) implies $s \triangleright_{\eta,C} t$.

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

\[ (\text{case}_K \ s \text{ of } (\text{pat}_1 \to \text{pat}_1) \ldots (\text{pat}_{|D_K|} \to \text{pat}_{|D_K|})) \to s \]

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

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

Lemma 6.9. If $s_1 \downarrow \land s \xrightarrow{T,\text{caseId}} t$, then $t_1 \downarrow$ and $rln(s) \geq rln(t)$.

Proof. Let $s \xrightarrow{T,\text{caseId}} t$ 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 $s_1 \to s$ can be closed. For diagram (21) we apply the induction hypothesis to $s_1 \to t_1$ which shows $t_1 \downarrow, rln(s_1) \geq rln(t_1)$ and thus also $t_1 \downarrow$ and $rln(s) \geq rln(t)$. For diagram (22) the induction hypothesis shows the claim. For diagram (23) we have $t \downarrow$, since (abse) is correct. Moreover, $t \xrightarrow{T,\text{abs},s} s'$ is equivalent to $s' \xrightarrow{T,\text{case},\eta} t$ and Theorem 3.2 (3) and (5) show $rln(s') = rln(t)$. Thus also $rln(s) \geq rln(t)$. For diagram (24) we have $t \downarrow$, since (cpx),(gc), and (cpx) are correct. Theorem 3.2 shows that $rln(s) > rln(s') = rln(t)$, since (cpx),(cpx) and (gc) do not change the measure $rln(\cdot)$. 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 $s \xrightarrow{T_{\text{caseId}}} t$ 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

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].

\[
\begin{align*}
\text{(26)} & \quad \text{if } a \in \{\text{\((lbeta), (case), (seq)\)}\} \\
\text{(27)} & \quad \text{if } a \in \{\text{\((lbeta), (seq)\)}\}
\end{align*}
\]

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 \( rln(t) = rln(t') \).

**Proof.** We use the context lemma 3.5 for improvement for the relation \( \approx \), i.e., we show \( (cp) \subseteq \approx_{iS,C} \) to derive \( \approx \subseteq \approx_{iS,C} \) = \( \approx \). 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 \( rln(s) = rln(s') \) by induction on \( rln(s) \) 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 \{\text{\((lbeta), (case), (seq)\)}\} \), then \( rln(s_1) = rln(s) - 1 \). 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 \( rln(s_2) = rln(s) - 1 \). 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 \( rln(s') = rln(s_1) \), we obtain \( rln(s) = rln(s') \).

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 \((\text{\((lbeta)\)}, \text{(case)}\), and \((\text{seq})\)-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 \( \text{Env} \) and a number \( n \geq 1 \), let \( s = \text{letrec Env in } (id_1 \ldots id_n) \). Then the equality \( rlnall(s) = \frac{n \cdot (n + 3) - 4}{2} \) and \( rln(s) = n - 1 \) 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 rln(s) = rlnall(s) = 0. For n = 2 the normal order reduction is as follows:

\[
\text{letrec } Env \text{ in } (id_1 id_2) \\
\rightarrow_{\text{no,lbeta}} \text{letrec } Env \text{ in } \text{letrec } x_1 = id_2 \text{ in } x_1 \\
\rightarrow_{\text{no,let}} \text{letrec } Env, x_1 = id_2 \text{ in } x_1 \\
\rightarrow_{\text{no,cp}} \text{letrec } Env, x_1 = id_2 \text{ in } id_2
\]

and thus rln(s) = 1 and rlnall(s) = 3.

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

\[
\text{letrec } Env \text{ in } id_1 id_2 \ldots id_n \\
\rightarrow_{\text{no,lbeta}} \text{letrec } Env \text{ in } (\text{letrec } x_1 = id_2 \text{ in } x_1) \text{ id}_3 \ldots id_n \\
\rightarrow_{\text{no,lapp,n-2}} \text{letrec } Env \text{ in } (\text{letrec } x_1 = id_2 \text{ in } (x_1 \text{ id}_3 \ldots id_n)) \\
\rightarrow_{\text{no,let}} \text{letrec } Env, x_1 = id_2 \text{ in } (x_1 \text{ id}_3 \ldots id_n) \\
\rightarrow_{\text{no,cp}} \text{letrec } Env, x_1 = id_2 \text{ in } (id_2 \text{ id}_3 \ldots id_n)
\]

By the induction hypothesis, we have

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

and 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

\[
s_{2^m} := \text{letrec } n = (S^{2^m} Z), f = F \text{ in } f \ n
\]

where F := λx.(case x of ((S y) → f y id) (Z → id))

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

\[
t_{2^m} := \text{letrec } x_{2^n} = \lambda h. S \ h, \{x_{2^i} = \lambda h. x_{2^i-1}(x_{2^i-1}h)\}_{i=1}^{m}, \\
h_0 = Z, h_{2^m} = x_{2^m} h_0, \\
f = F
\]

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

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

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 \( k + 2 \cdot 2^j = i \).

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 \( r\ln(\cdot) \) 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
   \[
   h_i = (x_{2^j} \ h_k) \\
   \xrightarrow{cp} \ h_i = (\lambda h. x_{2^j-1} (x_{2^j-1} \ h)) \ h_k \\
   \xrightarrow{lbeta, llet} \ h_i = x_{2^j-1} (x_{2^j-1} \ h), \ h = h_k \\
   \xrightarrow{cpx, gc} \ h_i = x_{2^j-1} (x_{2^j-1} \ h_k).
   \]
   Thus with one (lbeta) step, we derive a binding of type (3).
   (b) Assume that \( j = 0 \). Then the evaluation is
   \[
   h_i = (x_{2^0} \ h_{i-1}) \\
   \xrightarrow{cp} \ h_i = (\lambda h. S \ h) \ h_{i-1} \\
   \xrightarrow{lbeta, llet} \ h_i = S \ h, \ h = h_{i-1} \\
   \xrightarrow{cpx, gc} \ h_i = S \ h_{i-1}.
   \]
   Thus with one (lbeta) step, we derive a binding of type (1).
3. For case (3), the reduction is
   \[
   h_i = (x_{2^j} \ (x_{2^j} \ h_k)) \\
   \xrightarrow{cp} \ h_i = (\lambda h_{2^j+k} (x_{2^j-1} (x_{2^j-1} \ h_{2^j+k}))) \ (x_{2^j} \ h_k) \\
   \xrightarrow{lbeta, llet} \ h_i = (x_{2^j-1} (x_{2^j-1} \ h_{2^j+k})), \ h_{2^j+k} = (x_{2^j} \ h_k).
   \]
   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 \( 3 \cdot 2^m - 2 \) (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 letrec Env in \( (id_1 \ldots id_{2^{m+1}}) \) (Lemma B.1) yields \( r\ln(t_{2^m}) = 3 \cdot 2^m - 2 + 2 \cdot (2^m + 1) + ((2^m + 1) - 1) = 6 \cdot 2^m \).

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

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 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 \(\text{letrec } x = \lambda y. y \text{ in } (\lambda y'. y') (\lambda z. z) \xrightarrow{T,pcg} \text{letrec } x = \lambda y.y \text{ in } x (\lambda z. z)\) and \(\text{letrec } x = \lambda y.y, x = y, z = y \text{ in } r \xrightarrow{T,pcg} \text{letrec } x = \lambda y.y, x = y, z = x \text{ in } r\).

- The normal order reduction step is a \((no,cp)\)-reduction and leads to a WHNF, e.g. \(\text{letrec } x = \lambda y.y, z = \lambda y.y \text{ in } x \xrightarrow{no,cp} \text{letrec } x = \lambda y.y, z = \lambda y.y \text{ in } \lambda y.y\)

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 \text{seq} or in an unused alternative of a \text{case}-expression. A typical case is given in Fig. 13f.

Diagram (14) covers the case that the duplicated subexpression is a \text{letrec}-expression which is deconstructed by an \((no,let)\)-reduction. A typical case is given in Fig. 13g. Diagram (15) covers the case that the scrutinee of a \text{case}-expression is one of the duplicated expressions. A typical case is given in Fig. 13h.
Fig. 12: Typical cases for the diagrams for (pcgE), cont’d.
\[
\text{letrec } z = r'', x = \lambda x'.r'' \text{ in } (x \ u) \xrightarrow{T_{\text{seq}}} \text{letrec } z = r'', x = \lambda x'.z \text{ in } (x \ u)
\]

\[
\text{letrec } z = r'', x = \lambda x'.r'' \text{ in } ((\lambda x'.r'') \ u) \xrightarrow{n.o,\text{cp}} \text{letrec } z = r'', x = \lambda x'.z \text{ in } ((\lambda x'.z) \ u)
\]

\[
\text{letrec } z = r'', x = \lambda x'.r'' \text{ in } \text{letrec } x' = u \text{ in } r'' \xrightarrow{T_{\text{seq}}} \text{letrec } z = r'', x = \lambda x'.z \text{ in } \text{letrec } x' = u \text{ in } z
\]

(a) Typical case for diagram (10)

\[
\text{letrec } z = r'', x = \lambda x'.r'' \text{ in } (\text{seq } x \ u) \xrightarrow{T_{\text{seq}}} \text{letrec } z = r'', x = \lambda x'.z \text{ in } (\text{seq } x \ u)
\]

\[
\text{letrec } z = r'', x = \lambda x'.r'' \text{ in } (\text{seq } (\lambda x'.r'') \ u) \xrightarrow{n.o,\text{cp}} \text{letrec } z = r'', x = \lambda x'.z \text{ in } (\text{seq } (\lambda x'.z) \ u)
\]

\[
\text{letrec } z = r'', x = \lambda x'.r'' \text{ in } u \xrightarrow{T_{\text{seq}}} \text{letrec } z = r'', x = \lambda x'.z \text{ in } u
\]

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

\[
\text{letrec } z = x \ u, x = \lambda x'.r \xrightarrow{T_{\text{seq}}} \text{letrec } z = x \ u, x = \lambda x'.r \text{ in } z
\]

\[
\text{letrec } z = x \ u, x = \lambda x'.r \text{ in } ((\lambda x'.r) \ u) \xrightarrow{n.o,\text{cp}} \text{letrec } z = ((\lambda x'.r) \ u), x = \lambda x'.r \text{ in } z
\]

\[
\text{letrec } z = x \ u, x = \lambda x'.r \text{ in } \text{letrec } x' = u \text{ in } r \xrightarrow{C,\text{cp},C,\text{beta}} \text{letrec } z = (\text{letrec } x' = u \text{ in } r), x = \lambda x'.r \text{ in } z
\]

(c) Typical case for diagram (11)

\[
\text{letrec } z = \text{seq } x \ u, x = \lambda x'.r \xrightarrow{T_{\text{seq}}} \text{letrec } z = \text{seq } x \ u, x = \lambda x'.r \text{ in } z
\]

\[
\text{letrec } z = \text{seq } x \ u, x = \lambda x'.r \text{ in } (\text{seq } \lambda x'.r) \ u) \xrightarrow{n.o,\text{cp}} \text{letrec } z = (\text{seq } (\lambda x'.r) \ u), x = \lambda x'.r \text{ in } z
\]

\[
\text{letrec } z = \text{seq } x \ u, x = \lambda x'.r \text{ in } u \xrightarrow{C,\text{cp},C,\text{seq}} \text{letrec } z = \text{seq } x \ u, x = \lambda x'.r \text{ in } u
\]

(d) Typical case for diagram (11) with (seq)
Fig. 13: Typical cases for the diagrams for (pg), cont’d.