Space Improvements and Equivalences in a Polymorphically Typed Functional Core Language: Context Lemmas and Proofs

Manfred Schmidt-Schauß and Nils Dallmeyer

Goethe-University, Frankfurt, Germany

Technical Report Frank-57 (V2)

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

30th October 2017

Abstract. We explore space improvements in LRP, a polymorphically typed call-by-need functional core language. A relaxed space measure is chosen for the maximal size usage during an evaluation. It abstracts from the details of the implementation via abstract machines, but it takes garbage collection into account and thus can be seen as a realistic approximation of space usage. The results are: a context lemma for space improving translations and for space equivalences; all but one reduction rule of the calculus are shown to be space improvements, and the exceptional one, the copy-rule, is shown to increase space only moderately. Several further program transformations are shown to be space improvements or space equivalences, in particular the translation into machine expressions is a space equivalence. Some space-worsening program transformations are classified as space-safe upto or as space-leaks. These results are a step forward in making predictions about the change in runtime space behavior of optimizing transformations in call-by-need functional languages.

1 Introduction

The focus of this paper is on providing methods for analyzing optimizations for call-by-need functional languages. Haskell [11, 4] is a functional programming language that uses lazy evaluation, and employs a polymorphic type system. Programming in Haskell is declarative, which avoids overspecifying imperative details of the actions at run time. Together with the type system this leads to a compact style of high level programming and avoids several types of errors.

The declarative features must be complemented with a more sophisticated compiler including optimization methods and procedures. Declarativeness in connection with lazy evaluation (which is call-by-need [1, 13] as a sharing variant of call-by-name) gives a lot of freedom to the exact execution and is accompanied by a semantically founded notion of correctness of the compilation. Compilation is usually a process that translates the surface program into a core language, where the optimization process can be understood as a sequence of transformations producing a final program.

Evaluation of a program or of an expression in a lazily evaluating functional language is connected with variations in the evaluation sequences of the expressions in the function body, depending on the arguments. The optimization exploits this and usually leads to faster evaluation. The easy to grasp notion of time improvements is contrasted by an opaque behavior of the evaluation w.r.t. space usage, which in the worst case may lead to space leaks. Programmers may experience this as unpredictability of space usage, generating rumors like "Haskell's space usage prediction is a black art" and in fact a loss of trust into the optimization. [7, 8, 2] observed that semantically correct modifications of the sequence of evaluation (for example due to strictness information) may have a dramatic effect on space usage. An example is (`head` xs) `eqBool` (`last` xs) vs. (`last` xs) `eqBool` (`head` xs) for an epression xs that generates a long list of Booleans (using the Haskell-conventions).

Early work on space improvements by Gustavsson and Sands [7, 8] provides deep insights and founded methods to analyze the dynamic space usage of programs in call-by-need functional languages. Our work is a reconsideration of the same issues, but there are some differences: Their calculus has a restricted syntax (for example the arguments of function calls must be variables), whereas our calculus is unrestricted; they investigate an untyped calculus, whereas we investigate a typed calculus. Measuring space is also slightly different: whereas [7, 8] counts only the bindings, we count the whole expression, but instead omit parts of the structure (for example variables are not counted). The difference in space measuring appears to be arbitrary, however, our measure turns out to ignore the right amount of noise and subtleties of space behavior, but nevertheless sufficiently models the reality. Also, their weak improvement relation that increases space only by linear function, is misleading, since a linear number of weak improvement transformations may lead to an exponential size explosion (see Section 9.1). This is not the case for our defined max-space improvement notion, which is related to the strong (space-) improvement notion in [6, 7].

The focus of this paper is to contribute to a better understanding of the space usage of lazy functional languages and to enable tools for a better predictability of space requirements. The approach is to analyze a polymorphically typed and call-by-need functional core language LRP that is a lambda calculus extended with the constructs letrec, case, constructors, seq, type abstractions, and with call-by-need evaluation. Call-by-need evaluation has a sharing regime and due to the recursive bindings by a letrec, in fact a sharing graph is used. Focussing on space usage enforces us to include garbage collection into the model, i.e. the core language. This model is our calculus *LRPgc*.

The **contributions and results** of this paper are: A definition (Def. 4.4) of the space measure *spmax* as an abstract version of the maximally used space by an abstract machine during an evaluation, and a corresponding definition of transformations to be max-space-improvements or -equivalences, where the criterion is that this holds for the application in every context. Two context lemmas (Props. 4.8 and 4.12), are proved that ease proofs of transformations being space improvements or equivalences. The main result is a classification in Section 5 of the rules of the calculus used as transformations, and of further transformations as max-space improvements and/or max-space equivalences. These results, in particular the space-equivalence of (ucp) (Thm. 6.3), then imply that the transformation into machine expressions keeps the max-space usage (Thm. 10.2), which also holds for the evaluations on the abstract machine (Prop. 10.3). In addition, for the (cp)-rule, which is not a max-space improvement, we give a criterion (Prop. 7.3), exhibiting the worst case of max-space increase of a version of (cp) not copying into abstractions. A generalisation is Theorem 7.4 that shows that n application of copying abstraction increases the maximally used space at most by a summand of n times the size of the initial program. The type-dependent rule (caseId) is shown to be a max-space improvement (Prop. 6.6), which is a minimal example for the space behavior of larger class of type-dependent transformations. We also classify in Section 7 some space-worsening transformations as well-behaved (space-safe upto) or as space-leaks. This is a contribution to predicting the space behavior of optimizing transformations, which in the future may lead also to a better control of powerful, but space-hazardous, transformations like inlining and common subexpression elimination (see Example 7.2). We also reconsider space properties of recursive function definitions in 9.1, and show that our relaxed measure leads to better results.

We discuss some **previous work** on time and space behavior for call-by-need functional languages. Examples for research on the correctness of program transformations are e.g. [14, 10, 21], and examples for the use of transformations in optimization in functional languages are [15] and the work on Hermit [22]. A theory of (time) optimizations of call-by-need functional languages was started in [12] for a call-by-need higher order language, also based on a variant of Sestoft's abstract machine [23]. Clearly there are other program transformations with a high potential to improve efficiency. An example transformation with a high potential to improve efficiency is common subexpression elimination, which is considered in [12], and which is proved correct in [17]. Hackett and Hutton [9] applied the improvement theory of [12] to argue that optimizations are indeed improvements, with a particular focus on (typed) worker/wrapper transformations (see e.g. [3] for more examples). The work of [9] uses

Syntax of expressions and types: Let type variables $a, a_i \in TVar$ and term variables $x, x_i \in Var$. Every type constructor K has an arity $ar(K) \geq 0$ and a finite set D_K of data constructors $c_{K,i} \in D_K$ with an arity $ar(c_{K,i}) \geq 0$.

Types Typ and polymorphic types $PTyp$ are defined as follows:
$\tau \in Typ \quad ::= a \mid (\tau_1 \to \tau_2) \mid (K\ \tau_1\ \ldots\ \tau_{ar(K)})$
$\rho \in PTyp ::= \tau \mid \forall a.\rho$

Expressions $Expr$ are generated by this grammar with $n \geq 1$ and $k \geq 0$:

$s, t \in Expr ::= u \mid x :: \rho \mid (s\ \tau) \mid (s\ t) \mid (\texttt{seq}\ s\ t) \mid (c_{K,i} :: (\tau)\ s_1\ \ldots\ s_{ar(c_{K,i})}) \mid (\texttt{letrec}\ x_1 :: \rho_1 = s_1, \ldots, x_n :: \rho_n = s_n\ \texttt{in}\ t)$
$\quad \mid (\texttt{case}_K\ s\ \texttt{of}\ \{(Pat_{K,1}\ \texttt{->}\ t_1)\ \ldots\ (Pat_{K,|D_K|}\ \texttt{->}\ t_{|D_K|})\})$
$Pat_{K,i} \quad ::= (c_{K,i} :: (\tau)\ (x_1 :: \tau_1)\ \ldots\ (x_{ar(c_{K,i})} :: \tau_{ar(c_{K,i})}))$
$u \in PExpr ::= (\Lambda a_1.\Lambda a_2. \ldots \Lambda a_k.\lambda x :: \tau.s)$

Fig. 1. Syntax of expressions and types of LRP

the same call-by-need abstract machine as [12] with a slightly modified measure for the improvement relation. Further work that analyses space-usage of a lazy functional language is [2], for a language without letrec and using a term graph model, and comparing different evaluators.

The **structure of the paper** is to first define the calculi LRP and *LRPgc* in Section 2. Sect. 3 contains garbage collection variants of the calculi. Sect. 4 defines space improvements and contains the context lemmata. Sect. 5 contains a detailed treatment of space improving transformations. Sect. 6 analyses unique copying and a particular typed max-space improvement. Sect. 7 considers space-worsening transformations. Sect. 8 summarizes the results. Sect. 9 analyses the impact of some time-improvements on space usage, and in Sect. 10 there are some remarks on the abstract machine and the translation into machine language. Missing proofs are in the Appendix.

(lbeta) $((\lambda x.s)^{\text{sub}}\ r) \to (\texttt{letrec}\ x = r\ \texttt{in}\ s)$
(Tbeta) $((\Lambda a.u)^{\text{sub}}\ \tau) \to u[\tau/a]$
(cp-in) $(\texttt{letrec}\ x_1 = v^{\text{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, E\ \texttt{in}\ C[x_m^{\text{vis}}]) \to (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^m, E\ \texttt{in}\ C[v])$
where v is a polymorphic abstraction
(cp-e) $(\texttt{letrec}\ x_1 = v^{\text{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, E, y = C[x_m^{\text{vis}}]\ \texttt{in}\ r) \to (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^m, E, y = C[v]\ \texttt{in}\ r)$
where v is a polymorphic abstraction
(llet-in) $(\texttt{letrec}\ E_1\ \texttt{in}\ (\texttt{letrec}\ E_2\ \texttt{in}\ r)^{\text{sub}}) \to (\texttt{letrec}\ E_1, E_2\ \texttt{in}\ r)$
(llet-e) $(\texttt{letrec}\ E_1, x = (\texttt{letrec}\ E_2\ \texttt{in}\ t)^{\text{sub}}\ \texttt{in}\ r) \to (\texttt{letrec}\ E_1, E_2, x = t\ \texttt{in}\ r)$
(lapp) $((\texttt{letrec}\ E\ \texttt{in}\ t)^{\text{sub}}\ s) \to (\texttt{letrec}\ E\ \texttt{in}\ (t\ s))$
(lcase) $(\texttt{case}_K\ (\texttt{letrec}\ E\ \texttt{in}\ t)^{\text{sub}}\ \texttt{of}\ alts) \to (\texttt{letrec}\ E\ \texttt{in}\ (\texttt{case}_K\ t\ \texttt{of}\ alts))$
(seq-c) $(\texttt{seq}\ v^{\text{sub}}\ t) \to t \qquad$ if v is a value
(seq-in) $(\texttt{letrec}\ x_1 = (c\ \vec{s})^{\text{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, E\ \texttt{in}\ C[(\texttt{seq}\ x_m^{\text{vis}}\ t)])$
$\to (\texttt{letrec}\ x_1 = (c\ \vec{s}), \{x_i = x_{i-1}\}_{i=2}^m, E\ \texttt{in}\ C[t])$
(seq-e) $(\texttt{letrec}\ x_1 = (c\ \vec{s})^{\text{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, E, y = C[(\texttt{seq}\ x_m^{\text{vis}}\ t)]\ \texttt{in}\ r)$
$\to (\texttt{letrec}\ x_1 = (c\ \vec{s}), \{x_i = x_{i-1}\}_{i=2}^m, E, y = C[t]\ \texttt{in}\ r)$
(lseq) $(\texttt{seq}\ (\texttt{letrec}\ E\ \texttt{in}\ s)^{\text{sub}}\ t) \to (\texttt{letrec}\ E\ \texttt{in}\ (\texttt{seq}\ s\ t))$
(case-c) $(\texttt{case}_K\ c^{\text{sub}}\ \texttt{of}\ \{\ldots(c\ \texttt{->}\ t)\ldots\}) \to t \quad$ if $ar(c) = 0$, otherwise:
$(\texttt{case}_K\ (c\ \vec{s})^{\text{sub}}\ \texttt{of}\ \{\ldots((c\ \vec{y})\ \texttt{->}\ t)\ldots\}) \to (\texttt{letrec}\ \{y_i = s_i\}_{i=1}^{ar(c)}\ \texttt{in}\ t)$
(case-in) $(\texttt{letrec}\ x_1 = c^{\text{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, E\ \texttt{in}\ C[(\texttt{case}_K\ x_m^{\text{vis}}\ \texttt{of}\ \{(c\ \texttt{->}\ r)\ldots\})])$
$\to (\texttt{letrec}\ x_1 = c, \{x_i = x_{i-1}\}_{i=2}^m, E\ \texttt{in}\ C[r]) \qquad$ if $ar(c) = 0$; otherwise:
$(\texttt{letrec}\ x_1 = (c\ \vec{t})^{\text{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, E\ \texttt{in}\ C[(\texttt{case}_K\ x_m^{\text{vis}}\ \texttt{of}\ \{((c\ \vec{z})\ \texttt{->}\ r)\ldots\})])$
$\to (\texttt{letrec}\ x_1 = (c\ \vec{y}), \{y_i = t_i\}_{i=1}^{ar(c)}, \{x_i = x_{i-1}\}_{i=2}^m, E\ \texttt{in}\ C[\texttt{letrec}\ \{z_i = y_i\}_{i=1}^{ar(c)}\ \texttt{in}\ r])$
(case-e) $(\texttt{letrec}\ x_1 = c^{\text{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, u = C[(\texttt{case}_K\ x_m^{\text{vis}}\ \texttt{of}\ \{(c\ \texttt{->}\ r_1)\ldots\})],\ E\ \texttt{in}\ r_2)$
$\to (\texttt{letrec}\ x_1 = c, \{x_i = x_{i-1}\}_{i=2}^m, u = C[r_1], E\ \texttt{in}\ r_2)$ if $ar(c) = 0$; otherwise:
$(\texttt{letrec}\ x_1 = (c\ \vec{t})^{\text{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, u = C[(\texttt{case}_K\ x_m^{\text{vis}}\ \texttt{of}\ \{\ldots((c\ \vec{z})\ \texttt{->}\ r)\ldots\})], E\ \texttt{in}\ s)$
$\to (\texttt{letrec}\ x_1 = (c\ \vec{y}), \{y_i = t_i\}_{i=1}^{ar(c)}, \{x_i = x_{i-1}\}_{i=2}^m, u = C[\texttt{letrec}\ \{z_i = y_i\}_{i=1}^{ar(c)}\ \texttt{in}\ r], E\ \texttt{in}\ s)$
The variables y_i are fresh ones.

Fig. 2. Basic LRP-reduction rules [16]

2 Lazy Lambda Calculi, Polymorphic and Untyped

We introduce the polymorphically typed LRP, since numerous complex transformations have their nice space improving property under all circumstances (in all contexts) only in a typed language, and then introduce LR, its untyped reduct, where reasoning is in general simpler. We will also consider the effect of typed transformations in the untyped calculus.

2.1 LRP: The Polymorphic Variant

We recall the polymorphically typed lazy lambda calculus (LRP) [18, 17, 16] as a language. We also motivate and introduce several necessary extensions for supporting realistic space analyses.

LRP [16] is LR (an extended call-by-need lambda calculus with letrec, e.g. see [21]) extended with types. I.e. LRP is an extension of the lambda calculus by polymorphic types, recursive `letrec`-expressions, `case`-expressions, `seq`-expressions, data constructors, type abstractions $\Lambda a.s$ to express polymorphic functions and type applications $(s\ \tau)$ for type instantiations. The syntax of expressions and types of LRP is defined in Fig. 1. Note that type-abstractions are restricted to abstractions which implies a typed progress lemma and makes the reduction of LRP and LR compatible.

An expression is well-typed if it can be typed using typing rules that are defined in [16]. LRP is a core language of Haskell and is simplified compared to Haskell, because it does not have type classes and is only polymorphic in the bindings of `letrec` variables. But LRP is strong enough to express polymorphically typed lists and functions working on such data structures.

From now on we use Env as abbreviation for a `letrec`-environment, $\{x_{g(i)} = s_{f(i)}\}_{i=j}^{m}$ for $x_{g(j)} = s_{f(j)}, \ldots, x_{g(m)} = s_{f(m)}$ and $alts$ for `case`-alternatives. We use $FV(s)$ and $BV(s)$ to denote free and bound variables of an expression s and $LV(Env)$ to denote the binding variables of a `letrec`-environment. Furthermore we abbreviate $(c_{K,i}\ s_1\ \ldots\ s_{ar(c_{K,i})})$ with $c\ \vec{s}$ and $\lambda x_1 . \ldots \lambda x_n . s$ with $\lambda x_1, \ldots, x_n . s$. The data constructors `Nil` and `Cons` are used to represent lists, but we may also use the Haskell-notation `[]` and (:) instead. A *context* C is an expression with exactly one (typed) hole $[\cdot :: \tau]$ at expression position. A *surface context*, denoted S, is a context where the hole is not within an abstraction, and a top context, denoted T, is a context where the hole is not in an abstraction or a case alternative. A *reduction context* is a context where reduction may take place, and it is defined using the labeling algorithm of [16]. Reduction contexts are for example $[\cdot]$, $([\cdot]\ e)$, (`case` $[\cdot]$ ...) and `letrec` $x = [\cdot], y = x, \ldots$ `in` $(x$ `True`$)$. Note that reduction contexts are surface contexts.

A *value* is an abstraction $\lambda x.s$, a type abstraction u or a constructor application $c\ \vec{s}$.

The classical β-reduction is replaced by the sharing (lbeta). (Tbeta) is used for type instantiations concerning polymorphic type bindings. The rules (cp-in) and (cp-e) copy abstractions which areneeded when the reduction rule has to reduce an application $(f\ g)$ where f is an abstraction defined in a `letrec`-environment. The rules (llet-in) and (llet-e) are used to merge nested `letrec`-expressions; (lapp), (lcase) and (lseq) move a `letrec`-expression out of an application, a `seq`-expression or a `case`-expression; (seq-c), (seq-in) and (seq-e) evaluate `seq`-expressions, where the first argument has to be a value or a value which is reachable through a `letrec`-environment. The rules (seq-in), (seq-e) are specialized to constructed values since we aim at a uniquely defined normal-order reduction. (case-c), (case-in) and (case-e) evaluate `case`-expressions by using `letrec`-expressions to realize the insertion of the variables for the appropriate `case`-alternative.

The following abbreviations are used: (cp) is the union of (cp-in) and (cp-e); (llet) is the union of (llet-in) and (llet-e); (lll) is the union of (lapp), (lcase), (lseq) and (llet); (case) is the union of (case-c), (case-in), (case-e); (seq) is the union of (seq-c), (seq-in), (seq-e).

Normal order reduction steps and termination are defined as follows:

Definition 2.1 (Normal order reduction). *A* normal order reduction step $s \xrightarrow{\text{LRP}} t$ *is performed (uniquely) if the labeling algorithm in [16] terminates on s inserting the labels sub (subexpression) and vis (visited) and the applicable rule (i.e. matching also the labels) of Fig. 2 produces t.* $\xrightarrow{\text{LRP},*}$ *is*

(gc1) $\texttt{letrec}\ \{x_i = s_i\}_{i=1}^{n}, E\ \texttt{in}\ t \to \texttt{letrec}\ E\ \texttt{in}\ t$ if $\forall i : x_i \notin FV(t, E), n > 0$
(gc2) $\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t \to t$ if for all $i : x_i \notin FV(t)$

Fig. 3. Garbage collection transformation rules for *LRPgc*

(cpx-in) $(\texttt{letrec}\ x = y, E\ \texttt{in}\ C[x]) \to (\texttt{letrec}\ x = y, E\ \texttt{in}\ C[y])$ where y is a variable and $x \neq y$
(cpx-e) $(\texttt{letrec}\ x = y, z = C[x], E\ \texttt{in}\ t) \to (\texttt{letrec}\ x = y, z = C[y], E\ \texttt{in}\ t)$ (same as above)
(cpcx-in) $(\texttt{letrec}\ x = c\ \vec{t}, E\ \texttt{in}\ C[x]) \to (\texttt{letrec}\ x = c\ \vec{y}, \{y_i = t_i\}_{i=1}^{ar(c)}, E\ \texttt{in}\ C[c\ \vec{y}])$
(cpcx-e) $(\texttt{letrec}\ x = c\ \vec{t}, z = C[x], E\ \texttt{in}\ t) \to (\texttt{letrec}\ x = c\ \vec{y}, \{y_i = t_i\}_{i=1}^{ar(c)}, z = C[c\ \vec{y}], E\ \texttt{in}\ t)$
(abs) $(\texttt{letrec}\ x = c\ \vec{t}, E\ \ \texttt{in}\ s) \to (\texttt{letrec}\ x = c\ \vec{x}, \{x_i = t_i\}_{i=1}^{ar(c)}, E\ \ \texttt{in}\ s)$ where $ar(c) \geq 1$
(abse) $(c\ \vec{t}) \to (\texttt{letrec}\ \{x_i = t_i\}_{i=1}^{ar(c)}\ \texttt{in}\ c\ \vec{x})$ where $ar(c) \geq 1$
(xch) $(\texttt{letrec}\ x = t, y = x, E\ \texttt{in}\ r) \ \to\ (\texttt{letrec}\ y = t, x = y, E\ \texttt{in}\ r)$
(ucp1) $(\texttt{letrec}\ E, x = t\ \texttt{in}\ S[x]) \to (\texttt{letrec}\ E\ \texttt{in}\ S[t])$
(ucp2) $(\texttt{letrec}\ E, x = t, y = S[x]\ \texttt{in}\ r) \to (\texttt{letrec}\ E, y = S[t]\ \texttt{in}\ r)$
(ucp3) $(\texttt{letrec}\ x = t\ \texttt{in}\ S[x]) \to S[t]$ where in the three (ucp)-rules, x has at most one occurrence in $S[x]$, no occurrence in E, t, r; and S is a surface context.

Fig. 4. Extra transformation rules

the reflexive, transitive closure, $\xrightarrow{\text{LRP},+}$ *is the transitive closure of* $\xrightarrow{\text{LRP}}$ *and* $\xrightarrow{\text{LRP},k}$ *denotes k normal order steps.*

In Fig. 2 we omit the types in all rules with the exception of (TBeta) for simplicity. Note that the normal-order reduction is type safe.

Definition 2.2. *A weak head normal form (WHNF) in LRP is a value, or an expression* $\texttt{letrec}\ E\ \texttt{in}\ v$*, where* v *is a value, or an expression* $\texttt{letrec}\ x_1 = c\ \vec{t}, \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \texttt{in}\ x_m$*. An expression* s converges *to an expression* t *(*$s{\downarrow}t$ *or* $s{\downarrow}$ *if we do not need* t*) if* $s \xrightarrow{\text{LRP},*} t$ *where* t *is a WHNF. Expression* s diverges *(*$s{\uparrow}$*) if it does not converge. The symbol* $\bot$ *represents a closed diverging expression, e.g.* $\texttt{letrec}\ x = x\ \texttt{in}\ x$*.*

Definition 2.3. *For* LRP*-expressions* s, t *of the same type* τ*,* $s \leq_c t$ *holds iff* $\forall C[\cdot :: \tau] : C[s]{\downarrow} \Rightarrow C[t]{\downarrow}$*, and* $s \sim_c t$ *holds iff* $s \leq_c t$ *and* $t \leq_c s$*. The relation* $\leq_c$ *is called* contextual preorder *and* $\sim_c$ *is called* contextual equivalence*.*

The following notions of reduction length are used for measuring the time behavior in LRP.

Definition 2.4. *For a closed* LRP*-expression* s *with* $s{\downarrow}s_0$*, let* $\texttt{rln}(s)$ *be the sum of the number of all (lbeta)-, (case)- and (seq)-reduction steps in* $s{\downarrow}s_0$*, let* $\texttt{rln}_{LCSC}(s)$ *be the sum of all a-reduction in* $s{\downarrow}s_0$ *with* $a \in LCSC$*, where* $LCSC = \{(lbeta), (case), (seq), (cp)\}$*.*

2.2 The Untyped Calculus LR

To be self contained, we give the necessary definitions and connections between LRP and LR as these appear in [18]. The good news is that if (TBeta)-reduction steps (that only manipulate types) are ignored, then this constitutes exactly the normal order reduction of the untyped expression.

Definition 2.5. *The calculus* LR *is defined on the set of expressions that is generated by a grammar that is derived from the one in Fig. 1 by omitting the types in the expression, but keeping the type constructor* K *at the* $\texttt{case}_K$ *constructs.*
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* $\varepsilon(c :: \rho) = c$*. We also define the type erasure for reduction sequences.*

(case-cx) $(\texttt{letrec}\ x = (c_{T,j}\ x_1 \ldots x_n), E\ \texttt{in}\ C[\texttt{case}_T\ x\ ((c_{T,j}\ y_1 \ldots y_n)\ \texttt{->}\ s)\ alts])$
$\to\ \texttt{letrec}\ x = (c_{T,j}\ x_1 \ldots x_n), E\quad \texttt{in}\ C[(\texttt{letrec}\ y_1 = x_1, \ldots, y_n = x_n\ \texttt{in}\ s)]$

(case-cx) $\texttt{letrec}\ x = (c_{T,j}\ x_1 \ldots x_n), E,\quad y = C[\texttt{case}_T\ x\ ((c_{T,j}\ y_1 \ldots y_n)\ \texttt{->}\ s)\ alts]\ \texttt{in}\ r$
$\to\ \texttt{letrec}\ x = (c\ x_1 \ldots x_n), E,\quad y = C[(\texttt{letrec}\ y_1 = x_1, \ldots, y_n = x_n\ \texttt{in}\ s)]\ \texttt{in}\ r$

(case-cx) in all other cases: like (case)

(case*) is defined as (case) if the scrutinized data expression is of the form $(c\ s_1 \ldots s_n)$, where $(s_1, \ldots, s_n)$ is not a tuple of different variables, and otherwise it is (case-cx)

(gc=) $\texttt{letrec}\ x = y, y = s, E\ \texttt{in}\ r\ \to\ \texttt{letrec}\ E\ \texttt{in}\ r$ where $x \notin FV(s, E, r)$, and $y = s$ cannot be garbage collected

(caseId) $(\texttt{case}_K\ s\ (pat_1\ \texttt{->}\ pat_1) \ldots (pat_{|D_K|}\ \texttt{->}\ pat_{|D_K|})) \to s$

Fig. 5. Variations of transformation rules (space improvements)

(cpS) is (cp) restricted such that only surface contexts S for the target context C are permitted

(cpcxT) is (cpcx) restricted such that only top contexts T for the target context C are permitted

(cse) $\texttt{letrec}\ x = s, y = s, E\ \texttt{in}\ r\ \to\ \texttt{letrec}\ x = s, E[x/y]\ \texttt{in}\ r[x/y]$ where $x \notin FV(s)$

(soec) changing the sequence of evaluation due to strictness knowledge by inserting `seq`.

Fig. 6. Some special transformation rules (space-worsening)

$$
\begin{array}{ll}
\texttt{size}(x) & = 0 \\
\texttt{size}(s\ t) & = 1 + \texttt{size}(s) + \texttt{size}(t) \\
\texttt{size}(\lambda x.s) & = 1 + \texttt{size}(s) \\
\texttt{size}(\texttt{case}\ e\ \texttt{of}\ \texttt{alt}_1 \ldots \texttt{alt}_n) & = 1 + \texttt{size}(e)\ \ + \sum_{i=1}^{n} \texttt{size}(\texttt{alt}_i) \\
\texttt{size}((c\ x_1 \ldots x_n)\ \texttt{->}\ e) & = 1 + \texttt{size}(e) \\
\texttt{size}(c\ s_1 \ldots s_n) & = 1 + \sum \texttt{size}(s_i) \\
\texttt{size}(\texttt{seq}\ s_1\ s_2) & = 1 + \texttt{size}(s_1) + \texttt{size}(s_2) \\
\texttt{size}(\texttt{letrec}\ x_1 = s_1, \ldots,\quad x_n = s_n\ \texttt{in}\ s) & = \texttt{size}(s) + \sum \texttt{size}(s_i)
\end{array}
$$

Fig. 7. Definition of `size`

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

Proposition 2.6. *The translation ε is adequate:*
$\varepsilon(e_1) \sim_c \varepsilon(e_2) \implies e_1 \sim_c e_2$.

It is not fully abstract (i.e. $e_1 \sim_c e_2$ does not imply $\varepsilon(e_1) \sim_c \varepsilon(e_2)$); an example will be the (caseId) transformation (see Section 6).

Definition 2.7. *Let s, t be two* LRP*-expressions of the same type ρ. The improvement relation $\preceq$ for* LRP *is defined as: Let $s \preceq t$ iff $s \sim_c t$ and for all contexts $C[\cdot :: \rho]$: if $C[s], C[t]$ are closed, then $\texttt{rln}(C[s]) \leq \texttt{rln}(C[t])$. If $s \preceq t$ and $t \preceq s$, we write $s \approx t$.*

The notation $s_1 \xrightarrow{C,a} s_2$ means $C[s_1]$ is transformed to $C[s_2]$ be reduction or transformation rule a. If every context C is permitted, then we may also write $\xrightarrow{a}$ instead of $\xrightarrow{C,a}$. The following facts are valid and can easily be verified or found in the literature [18, 20, 21]:

Theorem 2.8.

1. *For a closed* LRP*-expression s, the equations $\texttt{rln}(s) = \texttt{rln}(\varepsilon(s))$ and $\texttt{rln}_{LCSC}(s) = \texttt{rln}_{LCSC}(\varepsilon(s))$ hold.*
2. *The reduction rules (Fig. 2) and extra transformations (Figs. 4, 5 6) in their typed forms can also be used in* LRP*. They are correct program transformations and (time-) improvements.*
3. *If $s \xrightarrow{a} t$ where a is a reduction rule in any context, then $\texttt{rln}_{LCSC}(s) \geq \texttt{rln}_{LCSC}(t)$*
4. *If $s \xrightarrow{a} t$ where a is an extra transformation in any context, then $\texttt{rln}_{LCSC}(s) = \texttt{rln}_{LCSC}(t)$.*
5. *Common subexpression elimination applied to well-typed expressions is a (time-) improvement in* LRP *([17]).*

3 Calculi with Garbage Collection

As extra reduction rule in the normal order reduction we use garbage collection (gc), which is the union of (gc1) and (gc2), but restricted to the top letrec (see Fig. 4).

Definition 3.1 (*LRPgc*). *We define the calculus LRPgc as* LRP *modified by adding garbage collection to the normal-order reduction sequences. Let s be an* LRP*-expression (see [19, 16]). A* normal-order-gc (LRPgc) reduction step $s \xrightarrow{LRPgc} t$ *is defined by two cases:*

1. *If a (gc)-transformation is applicable to s (in the empty context), i.e. $s \xrightarrow{gc} t$, then $s \xrightarrow{LRPgc} t$, where the maximum of bindings is removed.*
2. *If* (1) *is not applicable and $s \xrightarrow{LRP} t$, then $s \xrightarrow{LRPgc} t$.*

A sequence of LRPgc-reduction steps is called a normal-order-gc reduction sequence or LRPgc-reduction sequence. *A WHNF without $\xrightarrow{LRPgc,gc}$-reduction possibility is called an* LRPgc-WHNF. *If the LRPgc-reduction sequence of an expression s halts with a LRPgc-WHNF, then we say s* converges w.r.t. LRPgc, *denoted as $s\downarrow_{LRPgc}$, or $s\downarrow$, if the calculus is clear from the context.*

The calculus LRgc is defined as the type erasure of LRPgc.

Note that an $\xrightarrow{LRPgc,gc2}$-reduction may have several subsequent $\xrightarrow{LRPgc,gc}$-reduction steps.
We will use complete sets of forking and commuting diagrams between transformation steps and the normal-order reduction steps (see [21] for more explanations). These cover all forms of overlaps of a normal-order-reduction and a transformation where also the context-class is fixed, and come with joining reduction and transformation steps. A forking is the pattern $\xleftarrow{LRPgc,a} \cdot \xrightarrow{trans}$, whereas a commuting is the pattern $\xrightarrow{trans} \cdot \xrightarrow{LRPgc,a}$.

Definition 3.2 ([21])). *The measure $\mu_{lll}(s)$ for an* LR*-expression s is defined as follows: $\mu_{lll}(s)$ is a pair $(\mu_{lll,1}(s), \mu_{lll,2}(s))$, ordered lexicographically. The measure $\mu_{lll,1}(s)$ is the number of* `letrec`*-subexpressions in s, and $\mu_{lll,2}(s)$ is the sum of* $\texttt{lrdepth}(C)$ *for all* `letrec`*-subexpressions r with $s \equiv C[r]$, where* `lrdepth` *is defined as follows, where $C_{(1)}$ is a context of hole depth* 1*:*

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

We need the following result later for inductive proofs on the steps of a reduction sequence.

Lemma 3.3. *The following inequations hold:*

1. *If $s \xrightarrow{lll} s'$, then $\mu_{lll}(s) > \mu_{lll}(s')$,*
2. *if $s \xrightarrow{T,gc} s'$, then $\mu_{lll}(s) \geq \mu_{lll}(s')$,*
3. *and if $s \xrightarrow{T,seq} s'$, then $\mu_{lll}(s) \geq \mu_{lll}(s')$.*

Proof. This is proved in [21] for $s \xrightarrow{lll} s'$, and obvious for $s \xrightarrow{gc} s'$ and $s \xrightarrow{seq} s'$.

Definition 3.4. *The syntactical size* $\texttt{synsize}(s)$ *of s is defined as:*

$$\begin{aligned}
&\texttt{synsize}(x) && = 1\\
&\texttt{synsize}(s\ t) && = 1+\texttt{synsize}(s)+\texttt{synsize}(t)\\
&\texttt{synsize}(\lambda x.s) && = 2+\texttt{synsize}(s)\\
&\texttt{synsize}(\texttt{case}\ e\ \texttt{of}\ \{\texttt{alt}_1 \ldots \texttt{alt}_n\}) && = 1+\texttt{synsize}(e)+\textstyle\sum_{i=1}^{n}\texttt{synsize}(\texttt{alt}_i)\\
&\texttt{synsize}((c\ x_1 \ldots x_n)\ \texttt{->}\ e) && = 1+n+\texttt{synsize}(e)\\
&\texttt{synsize}(c\ s_1 \ldots s_n) && = 1+\textstyle\sum \texttt{synsize}(s_i)\\
&\texttt{synsize}(\texttt{seq}\ s_1\ s_2) && = 1+\texttt{synsize}(s_1)+\texttt{synsize}(s_2)\\
&\texttt{synsize}(\texttt{letrec}\ x_1=s_1,\ldots,x_n=s_n\ \texttt{in}\ s) && = 1+n+\texttt{synsize}(s)+\textstyle\sum \texttt{synsize}(s_i)
\end{aligned}$$

Theorem 3.5. *The calculus* LRP *is convergence-equivalent to LRPgc. I.e. for all expressions s:* $s{\downarrow} \iff s{\downarrow}_{LRPgc}$.

Also, contextual equivalence and preorder for LRP *coincides with the corresponding notions in LRPgc.*

4 Space improvements

From now on we use the calculus *LRPgc* as defined in Definition 3.1. We define an adapted (weaker) size measure than `synsize`, which is useful for measuring the maximal space required to reduce an expression to a WHNF. The size-measure omits certain components. This turns into an advantage later, since this enables proofs for the exact behavior w.r.t. our space measure for a lot of transformations.

Definition 4.1. *The size* $\texttt{size}(s)$ *of an expression s is the following number:*

$$
\begin{array}{ll}
\texttt{size}(x) & = 0 \\
\texttt{size}(s\ t) & = 1 + \texttt{size}(s) + \texttt{size}(t) \\
\texttt{size}(\lambda x.s) & = 1 + \texttt{size}(s) \\
\texttt{size}(\texttt{case}\ e\ \texttt{of}\ \texttt{alt}_1 \ldots \texttt{alt}_n) & = 1 + \texttt{size}(e) \quad + \sum_{i=1}^{n} \texttt{size}(\texttt{alt}_i) \\
\texttt{size}((c\ x_1 \ldots x_n)\ \texttt{->}\ e) & = 1 + \texttt{size}(e) \\
\texttt{size}(c\ s_1 \ldots s_n) & = 1 + \sum \texttt{size}(s_i) \\
\texttt{size}(\texttt{seq}\ s_1\ s_2) & = 1 + \texttt{size}(s_1) + \texttt{size}(s_2) \\
\texttt{size}(\texttt{letrec}\ x_1 = s_1, \ldots,\ x_n = s_n\ \texttt{in}\ s) & = \texttt{size}(s) + \sum \texttt{size}(s_i)
\end{array}
$$

This measure does not count variables, and also counts bindings of a letrec only by the size of the bound expressions. Also, it ignores the type expressions and type annotations in the expressions.

The reason for defining $\texttt{size}(x)$ as 0 is that the let-reduction rules do not change the size, and that it is compatible with the size in the machine language. For example, the bindings $x = y$ do not contribute to the size-measure. This is justified, since the abstract machine ([5]) does not create $x = y$ bindings, (not even implicit ones) and instead makes an immediate substitution.

The sizes `size` and `synsize` differ only by a constant factor:

Proposition 4.2. *Let s be an* LRP*-expression. If s does not permit a garbage collection of any binding, and there are no* $x = y$*-bindings, then* $\texttt{synsize}(s) \leq (\mathit{maxarity} + 1) * \texttt{size}(s)$ *and* $\texttt{size}(s) \leq \texttt{synsize}(s)$*, where maxarity is the maximum of* 2 *and the maximal arity of constructor symbols in the language.*

Proof. It is sufficient to check every subexpression using an inductive argument.

Definition 4.3. *The space measure* $spmax(s)$ *of the reduction of a closed expression s is the maximum of those* $\texttt{size}(s_i)$*, where* $s_i \xrightarrow{LRPgc} s_{i+1}$ *is not a (gc), and where the reduction sequence is* $s = s_0 \xrightarrow{LRPgc} s_1 \xrightarrow{LRPgc} \ldots \xrightarrow{LRPgc} s_n$*, and* s_n *is a WHNF. If* $s{\uparrow}$*, then* $spmax(s)$ *is defined as* ∞.

For a (partial) reduction sequence $Red = s_1 \to \ldots \to s_n$*, we define* $spmax(Red) = \max_i\{size(s_i) \mid s_i \to s_{i+1}$ *is not a (gc)*$\}$.

Counting space only if there is no (LRPgc,gc)-reduction step possible is consistent with the definition in [8]. It also has the effect of avoiding certain small and short peaks in the space usage. The advantage is a better correspondence with the abstract machine and it leads to comprehensive results.

Definition 4.4. *Let* s, t *be two expressions with* $s \sim_c t$ *and* $s{\downarrow}$*. Then s is a* space-improvement *of t,* $s \leq_{spmax} t$*, if for all contexts C: if* $C[s]$*,* $C[t]$ *are closed then* $spmax(C[s]) \leq spmax(C[t])$*. If for all contexts C: if* $C[s]$*,* $C[t]$ *are closed then* $spmax(C[s]) = spmax(C[t])$*, then s is* space-equivalent *to t, denoted* $s \sim_{spmax} t$*. A transformation* $\xrightarrow{trans}$ *is called a space-improvement (space-equivalence) if* $s \xrightarrow{trans} t$ *implies that t is a space-improvement of (space-equivalent to, respectively) s.* □

Note that $\leq_{spmax}$ is a precongruence, i.e. it is transitive and $s \leq_{spmax} t$ implies $C[s] \leq_{spmax} C[t]$, and that $\sim_{spmax}$ is a congruence.

Lemma 4.5. *If $s \leq_{spmax} t$ for two expressions s, t, then $\texttt{size}(s) \leq \texttt{size}(t)$.*

Proof. The context $\lambda x.[\cdot]$ for a fresh variable x enforces $\texttt{size}(s) \leq \texttt{size}(t)$.

LRPgc-reduction contexts are the same as the LRP-reduction contexts.

Definition 4.6. *Let s, t be two expressions with $s \sim_c t$ and $s\downarrow$. The relation $s \leq_{R,spmax} t$ holds, provided the following holds: For all reduction contexts R and if $R[s]$, $R[t]$ are closed, then $spmax(R[s]) \leq spmax(R[t])$.*

4.1 Context Lemmas for Max-Space Improvements

The context lemmas proved below are very helpful in proving the space-improvement property, since less cases have to be considered in case analyses of the interferences between reductiosn and transformations.
The type-erasure relation between LRP and LR also holds between *LRPgc* and *LRgc*, and so the results can be transferred also to the typed language. We will now prove two context lemmas in *LRgc*, and thus we assume in the rest of this section that the expressions are LR-expressions and the calculus is *LRgc*.

The following lemma helps in the proof of the context lemma for max-space.

Lemma 4.7. *Let s be an expression with $s \xrightarrow{[\cdot],gc} s'$, i.e. it is a gc-reduction step in the empty context. Then $spmax(s) = spmax(s')$*

Proof. There are two cases:

1. If $s \xrightarrow{gc} s'$ is $s \xrightarrow{LRgc,gc} s'$, then the claim holds by applying *spmax*.
2. In the other case, the gc-transformation is a $(gc1)$-reduction, and the following diagram holds:

$$\begin{array}{ccc} s & \xrightarrow{\quad gc \quad} & s' \\ \downarrow{\scriptstyle LRgc,gc1} & \nearrow_{LRgc,gc} & \\ s_1 & & \end{array}$$

and hence $spmax(s) = spmax(s_1) = spmax(s')$. □

We will use multi-contexts, which are are expressions with several holes $\cdot_i$, such that every hole has exactly one occurrence.

Lemma 4.8 (Context Lemma for Maximal Space Improvements). *If $\texttt{size}(s) \leq \texttt{size}(t)$, $FV(s) \subseteq FV(t)$, and $s \leq_{R,spmax} t$, then $s \leq_{spmax} t$.*

Proof. Let M be a multi-context. We prove the more general claim that if for all i: $\texttt{size}(s_i) \leq \texttt{size}(t_i)$, $FV(s_i) \subseteq FV(t_i)$, $s_i \leq_{R,spmax} t_i$, and $M[s_1, \ldots, s_n]$ and $M[t_1, \ldots, t_n]$ are closed and $M[s_1, \ldots, s_n]\downarrow$, then $spmax(M[s_1, \ldots, s_n]) \leq spmax(M[t_1, \ldots, t_n])$.
By the assumption that $s_i \sim_c t_i$, we have $M[s_1, \ldots, s_n] \sim_c M[t_1, \ldots, t_n]$ and thus $M[s_1, \ldots, s_n]\downarrow \iff M[t_1, \ldots, t_n]\downarrow$. The induction proof is (i) on the number of LRgc-reduction steps of $M[t_1, \ldots, t_n]$, and as a second parameter on the number of holes of M. We distinguish the following cases:

(I) The LRgc-reduction step of $M[t_1, \ldots, t_n]$ is a (gc). If M is the empty context, then we can apply the assumption $s_1 \leq_{R,spmax} t_1$, which shows $spmax(s_1) \leq spmax(t_1)$. Now we can assume that M is not empty, hence it is a context starting with a `letrec`, and in $M[t_1, \ldots, t_n]$ the reduction (gc) removes a subset of the bindings in the top letrec, resulting in $M'[t'_1, \ldots, t'_k]$. Since $FV(s_i) \subseteq FV(t_i)$, the same set of bindings in the top `letrec` can be removed in $M[s_1, \ldots, s_n]$

by (gc) resulting in $M'[s'_1,\ldots,s'_k]$, where the pairs (s'_i,t'_i) are renamed versions of pairs (s_j,t_j). If the reduction step is a (gc2), or if it is a (gc1) with $M[s_1,\ldots,s_n] \xrightarrow{LRgc,gc1} M'[s'_1,\ldots,s'_k]$, then by induction we obtain $spmax(M'[s'_1,\ldots,s'_k]) \leq spmax(M'[t'_1,\ldots,t'_k])$. Since $spmax$ is not changed by (gc)-reduction, this shows the claim. However, in case the (gc1) step that is not a LRgc-reduction step, it does not remove the maximal set of removable bindings in $M[s_1,\ldots,s_n]$. By induction we obtain $spmax(M'[s'_1,\ldots,s'_k]) \leq spmax(M'[t'_1,\ldots,t'_k])$. We use Lemma 4.7, which shows $spmax(M'[s'_1,\ldots,s'_k]) = spmax(M[s_1,\ldots,s_n])$, and $spmax(M'[t'_1,\ldots,t'_k]) = spmax(M[t_1,\ldots,t_n])$, and thus the claim.

(II) If no hole of M is in a reduction context and the reduction step is not a (gc), then there are two cases: (i) $M[t_1,\ldots,t_n]$ is a WHNF. Then also $M[s_1,\ldots,s_n]$ is a WHNF, and by the assumption, we have $\texttt{size}(M[s_1,\ldots,s_n]) \leq \texttt{size}(M[t_1,\ldots,t_n])$. (ii) The reduction step is $M[t_1,\ldots,t_n] \xrightarrow{LRgc,a} M'[t'_1,\ldots,t'_{n'}]$, and $M[s_1,\ldots,s_n] \xrightarrow{LRgc,a} M'[s'_1,\ldots,s'_{n'}]$ with $a \neq gc$, and the pairs (s'_i,t'_i) are renamed versions of pairs (s_j,t_j). This shows $spmax(M'[s'_1,\ldots,s'_{n'}]) \leq spmax(M'[t'_1,\ldots,t'_{n'}])$ by induction. By assumption, the inequation $\texttt{size}(M[s_1,\ldots,s_n]) \leq \texttt{size}(M[t_1,\ldots,t_n])$, holds, hence by computing the maximum, we obtain $spmax(M[s_1,\ldots,s_n]) \leq spmax(M[t_1,\ldots,t_n])$.

(III) Some t_j in $M[t_1,\ldots,t_n]$ is in a reduction position, and there is no LRgc-gc-reduction of $M[t_1,\ldots,t_n]$. Then there is one hole, say i, of M that is in a reduction position. With $M' = M[\cdot,\ldots,\cdot,t_i,\cdot,\ldots,\cdot]$, we can apply the induction hypothesis, since the number of holes of M' is strictly smaller than the number of holes of M, and the number of normal-order-gc reduction steps of $M[t_1,\ldots,t_n]$ is the same as of $M'[t_1,\ldots,t_{i-1},t_{i+1},\ldots,t_n]$, and obtain: $spmax(M[s_1,\ldots,s_{i-1},t_i,s_{i+1},\ldots,s_n]) \leq spmax(M[t_1,\ldots,t_{i-1},t_i,t_{i+1},\ldots,t_n])$. Also by the assumption: $spmax(M[s_1,\ldots,s_{i-1},s_i,s_{i+1},\ldots,s_n]) \leq spmax(M[s_1,\ldots,s_{i-1},t_i,s_{i+1},\ldots,s_n])$, since $M[s_1,\ldots,s_{i-1},\cdot,s_{i+1},\ldots,s_n])$ is a reduction context. Hence $spmax(M[s_1,\ldots,s_n]) \leq spmax(M[t_1,\ldots,t_n])$. □

Example 4.9. The conditions $FV(s) \subseteq FV(t)$ and $\texttt{size}(s) \leq \texttt{size}(t)$ are necessary in the context lemma: $FV(s) \subseteq FV(t)$ is necessary: Let $s = \texttt{letrec}\ y = x\ \texttt{in True}$, and let $t = \texttt{letrec}\ y = \texttt{True in True}$. Then $s \sim_c t$, since s and t are both contextually equivalent to `True`, using garbage collection. Also $\texttt{size}(s) \leq \texttt{size}(t)$. But s is not a max-space improvement of t: Let C be the context $\texttt{letrec}\ x = s_1, z = s_2\ \texttt{in seq}\ z\ (\texttt{seq}\ (c[\cdot])\ z)$, where s_1, s_2 are closed expressions such that $\texttt{size}(s_1) \geq 2$ and the evaluation of s_2 produces a WHNF $s_{2,WHNF}$ of size at least $1 + \texttt{size}(s_1) + \texttt{size}(s_2)$. This is easy to construct using recursive list functions. Then the reduction sequence of $C[s]$ reaches the size maximum after s_2 is reduced to WHNF due to the first `seq`, which is $1 + \texttt{size}(s_1) + \texttt{size}(s_{2,WHNF}) + 3 + \texttt{size}(s)$. The reduction sequence of $C[t]$ first removes s_1, and then reaches the same maximum as s, which is $1 + \texttt{size}(s_{2,WHNF}) + 3 + \texttt{size}(t)$. Thus $spmax(C[s]) - spmax([t]) = \texttt{size}(s_1) + \texttt{size}(s) - \texttt{size}(t) = \texttt{size}(s_1) - 1 > 0$. We have to show that for all reduction contexts R, $spmax(R[s]) \leq spmax(R[t])$: Reducing $R[s]$ will first shift (perhaps in several steps) the binding $y = x$ to the top letrec and then remove it (together with perhaps other bindings) with gc. The same for $R[t]$. After this removal, the expressions are the same. Hence $spmax(R[s]) \leq spmax(R[t])$. This shows that if $FV(s) \subseteq FV(t)$ is violated, then the context lemma does not hold in general. Note that this example also shows that for arbitrary expressions s, t with $s \sim_c t$ and $s{\downarrow}$, the relation $s \leq_{R,spmax} t$ does not imply $FV(s) \subseteq FV(t)$.

$\texttt{size}(s) \leq \texttt{size}(t)$ is necessary in the context lemma: Let t be a small expression that generates a large WHNF, and let s be `seq True` t. Then $\texttt{size}(s) > \texttt{size}(t)$. Lemma 4.5 shows (by contradiction) that s cannot be a space improvement of t. For all reduction contexts R, the first non-gc reduction will join the reduction sequences of $R[s]$ and $R[t]$. Since the WHNF of s is large, we obtain $spmax(R[s]) = spmax(R[t])$, since the size difference of s, t which is 1, is too small compared with the size of the WHNF. This implies $s \leq_{R,spmax} t$, but s is not a max-space improvement of t. Thus the condition $\texttt{size}(s) \leq \texttt{size}(t)$ is necessary in the max-space-context-lemma.

An immediate consequence of the context lemma 4.8 is:

Proposition 4.10. *The context lemma also holds for all context classes that contain the reduction contexts. In particular for top-contexts T, which are all contexts where the hole is neither in an abstraction nor in an alternative of a case expression.*

We show that there are cases which do not change the max-space consumption, and also adapt the context lemma to this case.

Definition 4.11. *Let s, t be two expressions with $s \sim_c t$ and $s{\downarrow}$. The relation $s \sim_{R,spmax} t$ holds, provided the following holds: For all reduction contexts R and if $R[s]$, $R[t]$ are closed, then $spmax(R[s]) = spmax(R[t])$.*

Lemma 4.12 (Context Lemma for Maximal Space Equivalence). *If $\texttt{size}(s) = \texttt{size}(t)$, $FV(s) = FV(t)$, and $s \sim_{R,spmax} t$, then $s \sim_{spmax} t$.*

The context lemmas for max-space improvement and max-space equivalence in the polymorphic variant *LRPgc* cannot be derived from the context lemmas in *LRgc*. However, it is easy to see that the reasoning in the proofs of the context lemmas is completely analogous and so we obtain:

Proposition 4.13. *The context lemmas for max-space improvement and max-space equivalence also hold in LRPgc.*

5 Space Improving Transformations

The plan of this section is to analyze the reductions of the calculus in Fig. 2. For complete proofs, it is required to analyze several other (special) transformations (see Figs. 5 and 6) w.r.t. their max-space improvement or max-space equivalence or space-worsening properties. We will perform the analyses and proofs w.r.t. the calculus *LRgc*. This is justified since this implies the respective properties in *LRPgc*:

Proposition 5.1. *Let Q be a transformation in LRgc and let Q^P be the corresponding transformation in LRPgc. We assume that Q^P does not change the type of expressions in LRPgc. We also assume that for the type-erased relation it holds that $\varepsilon(Q^P) \subseteq Q$. Then the following holds:*

1. *If Q is a max-space improvement, then also Q^P is a max-space improvement.*
2. *If Q is a max-space equivalence then also Q^P is a max-space equivalence.*

Proof. This is obvious, since the size measure is the same, and since every type erased context from *LRPgc* is also an untyped context.

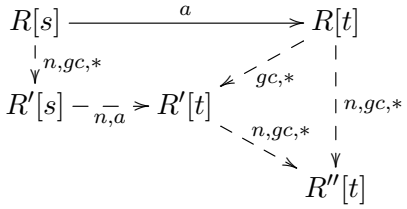
Proposition 5.2. *The (lll)-transformation steps in any context do not change the* `size` *of expressions and the (case)-, (lbeta)- and (seq)-transformations in any context strictly decrease the* `size`*.*

Proof. The reduction (llet) that merges environments does not change `size`.
The reductions (lapp), (lseq), and (lcase) only move the letrec, and so the size is the same. For (lbeta)-, (seq)- and (case-c)-reductions the claim is trivial. For (case-e) and (case-in)-reductions w.r.t. constructors of arity ≥ 1, this also holds, where it is exploited that variables do not count in the measure.

Theorem 5.3. *The transformations (lbeta), (seq-c) and (case-c) are max-space improvements.*

Proof. We apply the context lemma for max-space improvement and reduction contexts: Let $s \xrightarrow{a} t$ where $a \in \{(lbeta), (\text{seq-c}), (\text{case-c})\}$ and the transformation is w.l.o.g. on the top of the expression s. The precondition $FV(t) \subseteq FV(s)$ is satisfied. Let R be a reduction context. We consider the cases for $R[s] \xrightarrow{a} R[t]$, which is normal-order for LR, but since (gc) may be also applicable, it may be not an LRPgc-reduction. An analysis of cases shows that the following diagram is valid[1]

[1] In this diagram and the following, we abbreviate LRPgc as n.



Proposition 5.2 shows that $\texttt{size}(R[s]) \geq \texttt{size}(R[t]) \geq \texttt{size}(R''[t])$ and also all intermediate expressions in the diagram have smaller size than $\texttt{size}(R[s])$.

The definition of spmax implies: $spmax(R[s]) \geq spmax(R[t])$ for all reduction contexts R. An application of the context lemma 4.8 shows the claim.

A detailed analysis using forking-diagrams and using the context lemma (see appendix) shows:

Theorem 5.4. *The transformations (seq), (lll) and (gc) are max-space improvements.*

Remark 5.5. The reduction (cp) is in general not a space improvement. The simple argument is that a (cp)-reduction that is not normal-order, copying an abstraction, increases the size und thus it cannot be a space improvement.

The transformation not yet analyzed are (case-e) and (case-in). We will show that these are space-improvements after we have shown that several transformations are space-equivalences.

Proposition 5.6. *The transformation cpxT is a space-equivalence.*

Proof. An analysis of forking overlaps between LRPgc-reductions and *cpxS*-transformations in top contexts shows that they (almost) commute, i.e., $s_1 \xleftarrow{n,a} s \xrightarrow{T,cpxS} s'$ can be joined by $s_1 \xrightarrow{T,cpxS,0\vee1} s_1' \xleftarrow{n,a} s'$. (Here we mean by $\xrightarrow{T,cpxS,0\vee1}$ a reduction sequence consisting of 0 or 1 steps $\xrightarrow{T,cpxS}$.) We will apply the context lemma for space equivalence (Proposition 4.12), which also holds for T-contexts. Let $s_0 \xrightarrow{cpx} t_0$, and let $s = R[s_0]$ and $s' = R[t_0]$. Then $\texttt{size}(s) = \texttt{size}(s')$ as well as $FV(s) = FV(s')$. We have to show $spmax(s) = spmax(s')$, which can be shown by an induction on the number of LRPgc-reductions of $R[s_0]$. An application of the context lemma for space equivalence finishes the proof.

Definition 5.7. *Let the transformation $s \xrightarrow{gc=} s'$ be the specialization of (gc) where a single binding $x = y$ is removed, where y is not free, and there is a binding for y that cannot be garbage collected after the removal of $x = y$.*

Several specialized variants of (gc) different from (gc=) are not space equivalences, contrary to the intuition:

Lemma 5.8. *Consider a (gc)-transformation that removes the binding $x = y$ in a top context in s.*

1. *If y is free in s, then the transformation is not necessarily a space equivalence.*
2. *If the removal changes the garbage status of other bindings then the gc-transformation is not necessarily a space equivalence.*

Proof. One example is sufficient to show both claims: Consider $s = (\texttt{letrec}\ y = r_1\ \texttt{in seq}\ r_2\ (\texttt{letrec}\ x = y\ \texttt{in True}))$, and $s' = (\texttt{letrec}\ y = r_1\ \texttt{in seq}\ r_2\ \texttt{True})$, where we assume that $spmax(r_2) > \texttt{size}(r_1) + \texttt{size}(r_2)$. Then $spmax(s) = \texttt{size}(r_1) + spmax(r_2) + 2$, whereas $spmax(s')$ is $spmax(r_2) + 2$, since in s, first r_2 is evaluated, whereas in s', (gc) first removes r_1.

The interpretation for the first claim is that the context is $(\texttt{letrec}\ y = r_1 \texttt{in}\ [\cdot])$, and in the second case the whole expression, where the garbage status of $y = r_1$ changes.

Proposition 5.9. *The transformation (gc=) is a space-equivalence.*

Proposition 5.10. *The variant (case*) of case (see Fig. 5) is a space improvement.*

Theorem 5.11. *The transformation (case) is a space improvement.*

Proof. Since either (case) is the same reduction as (case*), or (case*) is the same as (case) followed by several (cpx) and several (gc=)-transformations. Note that the (gc)-transformations necessary to remove the bindings are indeed (gc=)-transformations.
Since (gc=) and (cpx) are space-equivalences due to Propositions 5.9 and 5.6, and (case*) is a space improvement by Proposition 5.10, the claim holds.

Theorem 5.12. *All reductions used as transformation in Fig. 2 with the exception of (cp) are space improvements.*

Proof. This is a summary of Theorems 5.4, 5.11, and 5.3 and of Remark 5.5.

6 The Improvements (ucp) and (caseId)

6.1 Space Properties of unique copying (ucp)

The helper transformations (abs) and (xch) are space equivalences (see the appendix).

Proposition 6.1. *The transformations (abs) and (xch) are space equivalences.*

For the analysis of (ucp) we need a "local space property" for a specialization of (cpcx). Let (cpcxT) be the transformation (cpcx) where the target is in a T-context, and let $(T, (cpcxT))$ be the application of (cpcxT) in a T-context.

Lemma 6.2. *If* $s \xrightarrow{(T,(cpcxT))} s'$, *then* $spmax(s) \leq spmax(s') \leq spmax(s) + 1$.

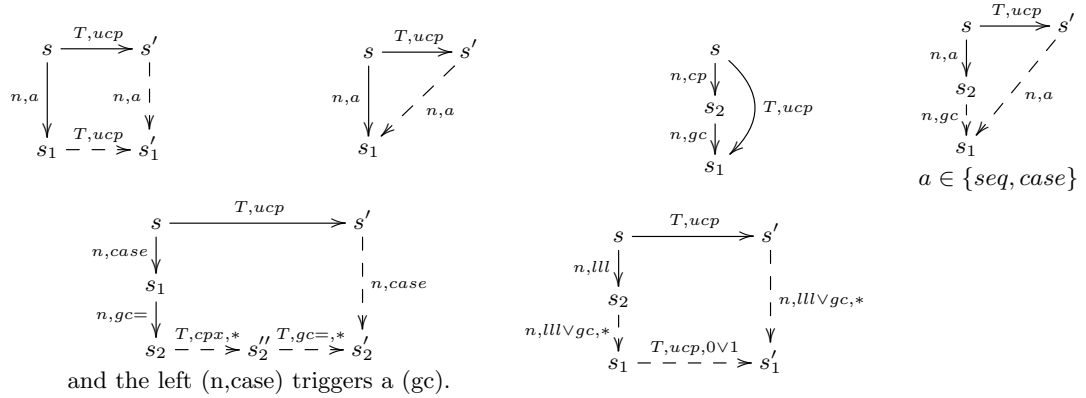


Fig. 8. Forking diagrams for (ucp)

We prove that the transformation (ucp) is a space equivalence, which is extremely helpful for proving space equivalence of the translation into machine expressions.

Theorem 6.3. *The transformation (ucp) is a space equivalence.*

Proof. We apply the forking diagrams in Fig. 8. The proof technique is by applying the context lemma for improvements. Let s_0, s_0' be expressions with $s = T[s_0] \xrightarrow{ucp} T[s_0'] = s'$. It is easy to see that $\texttt{size}(s) = \texttt{size}(s')$ and $FV(s) = FV(s')$. We show the prerequisite $spmax(s) = spmax(s')$ for the context lemma for T-contexts and space-equivalence by induction on the following measure of s: (i) μ_1, the number of $LCSC$-reductions, (ii) the measure $\mu_{lll}(s)$, and (iii) the measure $\texttt{synsize}(s)$.

See Theorem 2.8 and Lemma 3.3 for the modification and invariances of the measures μ_1, μ_2 under reductions.

If s is a LRPgc-WHNF, then the claim holds, since for $s \xrightarrow{ucp} t$: s is a WHNF iff t is a WHNF, and since (ucp) does not change the size.
If the first diagram is applicable, then the induction hypothesis can be applied to s_1, and thus $spmax(s_1) = spmax(s'_1)$. Since $\texttt{size}(s) = \texttt{size}(s')$, we obtain $spmax(s) = spmax(s')$. If the second diagram is applicable, then reasoning is obvious, since $\texttt{size}(s) = \texttt{size}(s')$. If the third diagram is applicable, then the induction hypothesis can be applied to s_1, and thus $spmax(s) = spmax(s_1)$, due to the definition of $spmax$ (Def. 4.3). If the 4th diagram is applicable, then $spmax(s) = spmax(s')$, since $\texttt{size}(s) = \texttt{size}(s')$, and (seq), (case) and (gc) do not increase the size.
Assume the 5th diagram is applicable. Then we have $\texttt{size}(s) = \texttt{size}(s')$ and $spmax(s_2) = spmax(s'_2)$, since (cpx) and (gc=) are space equivalences (see Propositions 5.9 and 5.6), hence $spmax(s) = spmax(s')$. For the 6th diagram, the claim is obvious if $s_1 = s'_1$. If $s_1 \xrightarrow{T,ucp} s'_1$, then the induction hypothesis can be applied to s_1, and thus $spmax(s_1) = spmax(s'_1)$. Since $\texttt{size}(s) = \texttt{size}(s')$ and $\texttt{size}(s_1)$, $\texttt{size}(s'_1)$, $\texttt{size}(s_2)$, are not greater than $\texttt{size}(s)$, the claim is proved. Now we can apply the context lemma 4.12 for space equivalence, which shows the overall claim.

6.2 A Typed Max-Space Improvement

In this section we show the max-space-improvement property for the transformation (caseId) in *LRPgc*. Note that in the untyped calculi LR and *LRgc* not every use of this transformation is correct. Nevertheless we make the reasoning in *LRgc*, since the condition that specific (caseId)-transformations are correct, is sufficient for valid arguments.

The rule (caseId) is also the heart of other type-dependent transformations, which are also only correct under typing. Examples for such transformations are: $(\texttt{map}\ \lambda x.x) \to \texttt{id}$, $\texttt{filter}\ (\lambda x.\texttt{True}) \to \texttt{id}$, and $\texttt{foldr}\ (:)\ [] \to \texttt{id}$, where we refer to the usual Haskell-functions and constructors. Note that these transformations are not correct in the untyped calculi. Applicative simulation can be used in LRP to prove correctness of the transformations (see [16]). These transformations are space-improvements in LRP, see the appendix for a sketch of a justification.

Definition 6.4. *The transformation (caseId) is defined as follows.*

$$\textit{(caseId)} \quad (\texttt{case}_K\ s\ (pat_1\ \texttt{->}pat_1) \dots (pat_{|D_K|}\ \texttt{->}pat_{|D_K|})) \to s$$

The transformation (caseId) is correct in LRP [18], but not in LR, which can be seen by trying the case $s = \lambda x.t$.

We only consider transformation instances $s_1 \xrightarrow{caseId} s_2$ in *LRgc*, where s_1, s_2 are contextually equivalent. In this case we say the (instance of the) (caseId)-transformation is *correct* in *LRgc*.

Lemma 6.5. *If* $\texttt{case}_K\ s\ \dots \xrightarrow{caseId} s$ *is correct, and* s *LRPgc-reduces to a LRPgc-WHNF* s'*, then* s' *is of the form* $(c\ s'_1 \dots s'_n)$*, where* c *is a constructor belonging to type* K*.*

Proposition 6.6. *Correct instances of the (caseId)-transformation are space-improvements.*

7 Space-Safe and Space-Unsafe Transformations

We consider useful program-transformations that are runtime optimizations, but may increase the space usage during runtime, and distinguish acceptable and bad behavior w.r.t. space usage.

Definition 7.1. *Let* T *be a transformation and let* $s \xrightarrow{T} t$ *be an instance with expressions* s, t*. Let* f *be a function* $f : \mathbb{R}^+ \to \mathbb{R}^+$*.*

1. *Then we say the transformation is* space-safe up to f*, if for all reduction context* R*:* $spmax(R[t]) \leq f(spmax(R[s]))$*.*

2. *In the special case that* $f(x) = x+c$*, we say that the transformation is* space-safe up to the constant c*, i.e. if for all reduction context* R*:* $spmax(R[t]) \leq c + spmax(R[s])$.
3. *The transformation is a* space leak*, iff for every* $b \in \mathbb{R}$*, there is a reduction context* R*, such that* $spmax(R[t]) \geq b + spmax(R[s])$.

Further transformations are defined and mentioned in Fig. 5 and 6: (casecx) and (case*) are variants of (case) which behave different if the tested expressions is of the form $(c\ x_1 \ldots x_n)$ by optimizing the heap-bindings; (cpS) is (cp) where the target for copying is an S-context; (cpcxT) is a variant of (cpcx), where the target context is a T-context; (caseId) is a typed transformation that detects case-expression that are trivial; (cse) means common subexpression elimination; (gc=) is a specialization of (gc) where a single binding $x = y$ in s is removed, where y is not free, and there is a binding for y that cannot be garbage collected after the removal of $x = y$; and the transformation (soec) means a correct change only of the evaluation order by inserting `seq`-expressions, due to strictness knowledge. The notation like $\xrightarrow{(T,(cpcxT))}$ means (cpcxT) applied in a top-context, and similar for others.

Our Definition 7.1 for a classification of space-worsening transformations make sense insofar as space-improvements are not space leaks and space leaks cannot be space improvements. We cannot use space-ticks like [8] since our notions of space-improvement and space-safety are contextual, and since we use look for maximal space.

We will see that there are examples for transformations that are not space-improvements but are space-safe up to a constant, and there are also transformations that are improvements w.r.t. runtime, but space-leaks, like (cp), (cse), and (soec).

Example 7.2. Common subexpression elimination may increase the max-space used in evaluations by a huge factor. We reuse an example which is similar to the example in [2]. The expression is given in a Haskell-like notation, using integers, but can also be defined in *LRPgc*: $s :=$ `if` (`last` $[1..n]$) $>$ 0 `then` $[1..n]$ `else Nil`, where $[1..n]$ is the expression that lazily generates a list $[1, \ldots, n]$. This expression evaluates the list expression until the last element is found and then evaluates the same expression again to $1 : [2..n]$. Due to eager garbage collection, it is not hard to see that the evaluation requires constant max-space, independent of n (assuming constant space for integers). In *LRPgc* the evaluation will also generate letrec-environments, perhaps with long indirection chains. It may seem unfair, since our space measure ignores these, but a simple change in evaluation (the normal-order) where such indirections are shortened (see e.g. [5]) will really use constant space also on an abstract machine.
Now let $s' =$ `letrec` $x = [1..n]$ `in if` (`last` x) > 0 `then` x `else Nil`. The evaluation of s' behaves different to s: it first evaluates the list, and stores it in full length, and then the second expression will be evaluated with an already evaluated list. The size required is a linear function in n. Seen from a complexity point of view, there is no real bound on this max-space increase: The example can be adapted using any computable function f on n by modifying the list to $[1..f(n)]$. Obviously this example is a space leak according to our definition, where the reduction contexts contains the list definition.

There may be instances of common subexpression elimination which are not space leaks, however, we leave the development of corresponding analyses for future research.

The example and arguments in [2] shows that correctly changing the sequence of evaluations may be a transformation that is a space leak: this means that (soec) is classified as a space leak.

As a final estimation we mention that the reduction (cp) used as transformation (cpS) in an S-context increases the max-space at most by `size`(v) where v is the copied abstraction. This enables very useful estimations of the effects of optimizing transformations w.r.t. their max-space-behavior for the transformations mentioned in this paper, in particular for optimizations by partial evaluation.

Proposition 7.3. *The transformation* $\xrightarrow{(S,cpS)}$ *increases max-space at most by* `size`(v)*, where* v *is the copied abstraction.*

Proof. See appendix

Note that this result cannot be formulated in terms of space-improvements, since the contexts in which this transformation can be applied are restricted.

A consequence is that the space usage of several transformations that are space-safe up to the constant c can be estimated:

Corollary 7.4. *Let t be an expression. If t is transformed into t' by an arbitrary number of space improvements that do not increase the size of abstractions, including at most n transformations that increase max-space by at most c_i for $i = 1, \ldots, n$, and also by m transformation $\xrightarrow{(S,cpS)}$, then $spmax(t') \leq spmax(t) + (\sum c_i) + m \cdot V$, where V is the maximum of the size of abstractions in t.*

Proof. This follows from Proposition 7.3 and and since $\xrightarrow{(S,cpS)}$ does not increase the size of abstractions.

Remark 7.5. Using (cp) as transformation with general contexts for the target, for example copying into an abstraction, may induce a space-leak. More exactly, the max-space of a reduction may increase linear with the number of reduction steps, and exponentially with the number of applications of the (cp)-transformation. Examples for this behavior can be constructed by as in Example 7.2.

Note that there are instances of (cp) that behave much better, for example versions of inlining or if the copied abstraction can be garbage collected after (cp) or transformed further, and also the special case of (ucp)-transformations.

8 Space Improvements

We show for concrete transformations that these are space improvements. For most of the proofs we will use complete sets of forking and commuting diagrams between transformation steps and the normal-order reduction steps (see [21] for more explanations), which are computed exploiting the context lemma. These cover all forms of overlaps of a normal-order-reduction and a transformation where also the context-class is fixed, and come with joining reduction and transformation steps.

An overview of our results for max-space improvements, -equivalences and space-worsening transformations are in the following theorem where further transformations are in Figs 3, 4, 5 and 6.

Theorem 8.1. *The following table shows the space-improvement and safety properties of the mentioned transformations.*

Improvement	*rules*
$\succeq_{spmax}$	*(lbeta), (case), (seq), (lll), (gc), (case*), (caseId)*
$\sim_{spmax}$	*(cpx), (abs), (abse), (xch), (ucp), (case-cx), (cpxT), (gc=)*
$\not\succeq_{spmax}$	*(cpcx), (cpS)*
space-safe upto 1	*(T,(cpcxT))*
space-safe upto $\|v\|$	*(S,(cpS))*
where v is the copied abstraction	
space-leak	*(cp), (cse), (soec)*

9 Optimizations with Controlled Space Usage

Our proposal for space-controlled program optimization in the call-by-need calculus *LRPgc* is as follows:

1. Consider an input program P that is to be optimized by transformations.
2. Use only program transformations that are correct w.r.t. contextual equivalence.

3. As a first filtering criterion: use only runtime-improving program transformations that are (also) space improvements. As analyzed in this paper, there is a sufficiently large set of such program transformations: First, these are all the reduction rules in Fig. 2 with the exception of (cp). Second, there are further transformation rules in Figs. 3, 4 and 5, which can be used, since we proved that all these rules are space-improvements.
   Future work may exhibit more such transformations. Since only space-improvements are applied, the guarantee is that the program itself is also not enlarged.
4. Further time-improving transformations can be applied that are not guaranteed to be space improvements, but there is an upper bound on the maximal space increase, see Fig. 6, as for example $(S, (cpS))$, and $(T, (cpcxT)$.
5. There are transformations such as unrestricted (cp) and common subexpression elimination (see Fig.6), which are time improvements (where the latter is shown in [17]), but as the concrete transformations may be space leaks, it comes with a higher risk of high size-requirements, hence further information on the max-space-effect is required for space-safe optimizations.

Some program transformations that are used for optimizing the runtime may increase the size of the programs but also max-space during evaluation. The symmetric disadvantage of max-space improving program translation is that these may increase the runtime, where an example is common subexpression elimination, see Example 7.2.

9.1 Examples for Space-Improvements with Recursive Functions

Associativity of append: In [8], the re-bracketing of $((xs \texttt{ ++ } ys) \texttt{ ++ } zs)$ was analyzed, and the results had to use several variants of their improvement orderings; in particular their observation of stack and heap space made the analysis rather complex. We got easier to obtain and grasp result due to our relaxed measure of space:

Our analysis of applying the associative law to the recursively defined append function ++ shows that $((xs \texttt{ ++ } ys) \texttt{ ++ } zs) \geq_{spmax} (xs \texttt{ ++ } (ys \texttt{ ++ } zs))$, where xs, ys, zs are variables. We know that the two expressions are contextually equivalent. The proof uses the context lemma for space improvement and in particular the space-equivalence of (ucp) which allows to inline uniquely used bindings, and an induction argument. The exact analysis shows that within reduction contexts, and under the assumption that only the appended list is inspected, the *spmax*-difference is exactly 4. The general estimation is that in reduction contexts R, we have $spmax(R[((xs \texttt{ ++ } ys) \texttt{ ++ } zs)]) \leq 4 + spmax(R[(xs \texttt{ ++ } (ys \texttt{ ++ } zs))])$.

For the **three sum-of-list-examples** in [8], the analysis using our size-measure results in comparable conclusions: They compare three functions: a plain recursively defined `sum` of a list of numbers, the tail-recursive function `sum'` with a non-strictly used accumulator and the tail-recursive `sum''` with a strictly used accumulator for the result.

`sum` requires space linear in the length of the list, and the same holds for `sum'`. However, `sum`, and `sum'` a and `sum''` as functions are not related by any improvement relation due to the change in the evaluation order of the spine and elements of the argument list, in case the list is not completely evaluated. In the latter case transforming one into the other may indeed be a space-leak, independent of the length of the list since it would be an instance of (soec).

(weak-value-beta) in Fig. 2 in [7]: As a further comparison we could check and compare our results with those for weak improvement in Fig. 2 in [7]: the claim on (weak-value-beta) there appears to be practically almost useless: copying once indeed can only increase the space by a linear function in the size of the program, even copying into an abstraction is permitted. However, repeating (weak-value-beta) n-times may increase the program exponentially (in n) by repeated doubling. The transformation rule in [7] permits $\texttt{letrec } x = V[x] \texttt{ in } C[x] \rightarrow \texttt{letrec } x = V[V[x]] \texttt{ in } C[V[x]] \rightarrow \texttt{letrec } x = V[V[V[x]]]] \texttt{ in } C[V[V[V[x]]]]$, where V is a value as context. Hence, a sequence of several weak improvement steps is not space-safe in the intuitive sense. According to our definition it is a space leak for this particular example.

Our foundations allow to improve the claims on the space-properties of last two let-shuffling rules of

[7], which are (strong) space-improvements w.r.t. our measure and definitions, since we have proved that (lll) is a space equivalence.

9.2 Additional Helpful Analyses

Let S be a set of closed expressions of the same (data-only) type, $s \in S$, and the full evaluation result is $eval(s)$. Let an *evaluation sequence* of s be a list L of all positions of $eval(s)$, such that whenever p_1 is a (proper) prefix of p_2, then p_1 is earlier in L than p_2. An evaluation of s *controlled by* L is the reduction sequence for s where the lazy evaluation evaluates the tail-nodes top-down in s, starting with the first position in L, and whenever the evaluation stops, tries the expression at the next position in L. Note that the expression s itself may evaluate deeper, for example $\texttt{let}\ xs = [1..n]\ \texttt{in seq}\ (\texttt{length}\ xs)\ xs$.

Definition 9.1. *Let S be as above. If there is a function f from integers to integers, such that for all $s \in S$, and all evaluation sequences L_s of s, the evaluation of s controlled by L_s requires at most space $\texttt{size}(eval(s)) + f(\texttt{size}(eval(s)))$, then we say the expressions of S can be evaluated in f-space. If $f(x)$ is a constant c, then it can be evaluated in constant space.*

For example, the set of list expressions $[1..n]$ can be evaluated in constant space. We conclude that for all list expressions that result in finite lists of integers, and that can be evaluated in constant space, and are non-empty, replacing `sum` by $\texttt{sum}''$ is a space-improvement.

10 Translating into Machine Language

An efficient implementation of the evaluation of programs or program expressions first translates expressions into a machine format that can be executed by an abstract machine. There are several machine models for call-by-need evaluation. We follow [12] in this respect, and discuss a variant of the Sestoft machine [23] that is extended by (eager) garbage collection. This model has the advantage that it is simple, efficient at least compared to the rules of the calculus, and makes the resource usage of evaluation explicit. A corresponding abstract machine and an environment measuring space and time is in [5]. In contrast to the pragmatics of occasionally turning on a complete garbage collector, the theoretical estimations assume that garbage is immediately collected after its generation, which means to have an eager garbage collector.

The translation of LRP into machine expressions is a mapping onto a subset of the expressions. The characteristic is that at several positions in expressions only variables are permitted, which can be achieved by putting the expressions in a letrec environment. The restricted expressions are the following: applications $(s\ x)$, case expressions $(\texttt{case}\ x\ alts)$, constructor applications $(c\ x_1 \ldots x_n)$, and seq-expressions $\texttt{seq}\ s\ x$.

Definition 10.1 (Translation to machine language). *[17] The translation ψ from arbitrary* LR*-expressions into machine expressions is defined as follows, where y, y_i are fresh variables:*

$$
\begin{array}{ll}
\psi(x) & := x \\
\psi(\lambda x.s) & := \lambda x.\psi(s) \\
\psi(s\ t) & := \texttt{letrec}\ y = \psi(t)\ \texttt{in}\ (\psi(s)\ y) \\
\psi(\texttt{seq}\ s\ t) & := \texttt{letrec}\ y = \psi(t)\ \texttt{in}\ (\texttt{seq}\ \ \psi(s)\ y) \\
\psi(c\ \vec{s}) & := \texttt{letrec}\ \{y_i = \psi(s_i)\}_{i=1}^{n}\ \texttt{in}\ (c\ \vec{y_i})
\end{array}
$$

$$
\psi\begin{pmatrix}\texttt{letrec}\ \{x_i = s_i\}_{i=1}^{n} \\ \quad \texttt{in}\ t\end{pmatrix} := \begin{pmatrix}\texttt{letrec}\{x_i = \psi(s_i)\}_{i=1}^{n} \\ \texttt{in}\ \psi(e)\end{pmatrix}
$$

$$
\psi\begin{pmatrix}\texttt{case}_K\ e\ \texttt{of} \\ \{(Pat_{K,1}\ \texttt{->} t_1) \\ \ldots \\ (Pat_{K,|D_K|}\ \texttt{->} t_{|D_K|})\}\end{pmatrix} := \begin{pmatrix}\texttt{case}_K\ \psi(e)\ \texttt{of} \\ \{(Pat_{K,1}\ \texttt{->} \psi(t_1)) \\ \ldots \\ (Pat_{K,|D_K|}\ \texttt{->} \psi(t_{|D_K|}))\}\end{pmatrix}
$$

This translation is semantically correct, since it only consists of correct transformation steps. It is also shown in [17], that counting the number of (lbeta)-, (case)- and (seq)-steps in LRP is equivalent to counting the corresponding steps on the abstract machine, and that the corresponding subsequences are the same.

The space measure is much more sensible to these changes, since space usage in the complete normal-order reduction is measured. For example the expression $((\lambda y.y)\ 0)$ evaluates in LR in one reduction step to $(\texttt{letrec}\ y = 0\ \texttt{in}\ y)$, whereas the corresponding machine expression $s_{me} := (\texttt{letrec}\ x = \lambda y.y\ \texttt{in}\ (x\ 0))$ has the *LR*-evaluation: $s_{me} \to \texttt{letrec}\ x = \lambda y.y\ \texttt{in}\ ((\lambda y.y)\ 0)$ $\to \texttt{letrec}\ x = \lambda y.y\ \texttt{in}\ (\texttt{letrec}\ y = 0\ \texttt{in}\ y)$.
Using *LRgc*, the reduction sequence is $s_{me} \to \texttt{letrec}\ x = \lambda y.y\ \texttt{in}\ (\lambda y.y)\ 0 \to (\texttt{letrec}\ y = 0\ \texttt{in}\ y)$. Since we only count space-usage after garbage collection (see Def. 4.3), as in [8], the small space peak in the second expression is ignored. The error which is made by this counting method is at most the size of the largest abstraction in the input program, which can be seen as a constant.

Theorem 10.2. *If s is translated into its corresponding machine expressions s_{me}, then s and s_{me} are space equivalent.*

Proof. A step-wise translation into machine expressions can be done by the reverse of the transformation (ucp), which is a space equivalence by Theorem 6.3.

Reducing in *LRgc* and the abstract machine is slightly different: Consider the example $s := \texttt{letrec}\ x_1 = \lambda y.y, x_2 = x_1, z = 0\ \texttt{in}\ (x_2\ z)$ in *LRgc* is as follows: $s \to \texttt{letrec}\ x_1 = \lambda y.y, x_2 = x_1, z = 0\ \texttt{in}\ ((\lambda y.y)\ z) \to \texttt{letrec}\ z = 0\ \texttt{in}\ ((\lambda y.y)\ z)\ \ldots$.
A reduction sequence corresponding to the one on the abstract machine is $s \to \texttt{letrec}\ x_1 = \lambda y.y, x_2 = \lambda y.y, z = 0\ \texttt{in}\ (x_2\ z) \to \texttt{letrec}\ x_1 = \lambda y.y, x_2 = \lambda y.y, z = 0\ \texttt{in}\ ((\lambda y.y)\ z) \to \ldots$.
The latter clearly requires more space. The remedy is twofold: machine programs are optimized by removing all $x = y$-bindings using transformations (cpx) and (gc), and second, there is a precaution in the abstract machine to avoid the generation of variable-variable bindings (see [5]). Note that these variable-variable bindings may also be implicit on the stack of the machine.

Proposition 10.3. *If $x = y$ bindings are avoided on the abstract machine, then evaluation of expressions in* LRP *and on the abstract machine in [5] have the same spmax-requirements.*

More examples and experiments on space usage that also are valid for our calculus can be found in [5].

11 Conclusion and Future Research

We successfully derived results on the space behavior of transformations in lazy functional languages, by defining a space measure and reasoning about space improvements.
Future work is to extend the analysis of transformations to larger and more complex transformations in the polymorphic typed setting. A generalisation from copy target positions and application context from top contexts to surface contexts is also of value for several transformations. To develop methods and justifications for space-improvements involving recursive definitions is also left for future work.

Acknowledgements.

We thank David Sabel for discussions and comments and anonymous referees for their diverse comments and helpful suggestions.

References

1. Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *POPL'95*, pages 233–246, San Francisco, California, 1995. ACM Press.

2. Adam Bakewell and Colin Runciman. A model for comparing the space usage of lazy evaluators. In *PPDP*, pages 151–162, 2000.
3. Richard Bird. *Thinking functionally with Haskell.* Cambridge University Press, Cambridge, UK, 2014.
4. Haskell Community. Haskell, an advanced, purely functional programming language, 2016.
5. Nils Dallmeyer and Manfred Schmidt-Schauß. An environment for analyzing space optimizations in call-by-need functional languages. In Horatiu Cirstea and Santiago Escobar, editors, *Proc. 3rd WPTE@FSCD*, volume 235 of *EPTCS*, pages 78–92, 2016.
6. Jörgen Gustavsson. *Space-Safe Transformations and Usage Analysis for Call-by-Need Languages.* PhD thesis, Department of Computing Science, Chalmers University of Technology and Göteborg University, 2001.
7. Jörgen Gustavsson and David Sands. A foundation for space-safe transformations of call-by-need programs. *Electr. Notes Theor. Comput. Sci.*, 26:69–86, 1999.
8. Jörgen Gustavsson and David Sands. Possibilities and limitations of call-by-need space improvement. In Benjamin C. Pierce, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001.*, pages 265–276, 2001.
9. Jennifer Hackett and Graham Hutton. Worker/wrapper/makes it/faster. In *ICFP '14*, pages 95–107, 2014.
10. Patricia Johann and Janis Voigtländer. The impact of seq on free theorems-based program transformations. *Fundamenta Informaticae*, 69(1–2):63–102, 2006.
11. Simon Marlow, editor. *Haskell 2010 – Language Report.* 2010.
12. A. K. D. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *POPL 1999*, pages 43–56. ACM Press, 1999.
13. Andrew K. D. Moran, David Sands, and Magnus Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, volume 1594 of *Lecture Notes in Comput. Sci.*, pages 85–102. Springer-Verlag, 1999.
14. Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4+5):393–434, July 2002.
15. Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998.
16. Manfred Schmidt-Schauß and David Sabel. Contextual equivalences in call-by-need and call-by-name polymorphically typed calculi (preliminary report). In M. Schmidt-Schauß, M.Sakai, D. Sabel, and Y. Chiba, editors, *WPTE 2014*, volume 40 of *OASICS*, pages 63–74. Schloss Dagstuhl, 2014.
17. Manfred Schmidt-Schauß and David Sabel. Improvements in a functional core language with call-by-need operational semantics. In Elvira Albert, editor, *Proc. PPDP '15*, pages 220–231, New York, NY, USA, 2015. ACM.
18. Manfred Schmidt-Schauß and David Sabel. Improvements in a functional core language with call-by-need operational semantics. Frank report 55, Institut für Informatik, Goethe-Universität Frankfurt am Main, March 2015. http://www.ki.informatik.uni-frankfurt.de/papers/frank/.
19. Manfred Schmidt-Schauß and David Sabel. Sharing decorations for improvements in a functional core language with call-by-need operational semantics. Frank report 56, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main, September 2015.
20. Manfred Schmidt-Schauß and David Sabel. Improvements in a call-by-need functional core language: Common subexpression elimination and resource preserving translations, 2016. submitted to a journal.
21. Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.
22. Neil Sculthorpe, Andrew Farmer, and Andy Gill. The HERMIT in the tree: Mechanizing program transformations in the GHC core language. In *Proceedings of the 24th Symposium on Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 86–103, 2013.
23. Peter Sestoft. Deriving a Lazy Abstract Machine. *J. Funct. Program.*, 7(3):231–264, May 1997.

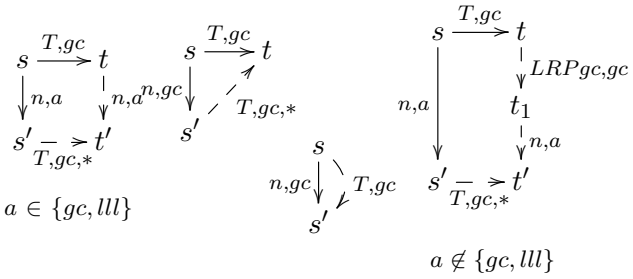
Appendix

Notation: In diagrams we label LRPgc-reductions with n, and (if necessary) non-LRPgc-reductions with i, and reductions in top-contexts with T. The straight arrows are given reductions/transformations and the dashed ones are implied existing ones.

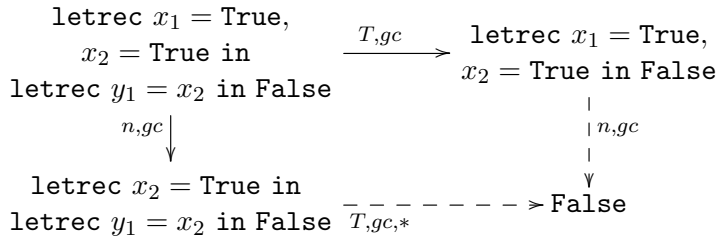
A Equivalence of LRP and *LRPgc*

We will use complete sets of forking and commuting diagrams between transformations and the normal-order reduction (see [21] for more explanations). These cover all forms of overlaps and come with joining reductions and transformations. A forking is the pattern $\xleftarrow{n,a} \cdot \xrightarrow{trans}$, whereas commuting is the pattern $\xrightarrow{trans} \cdot \xrightarrow{n,a}$.

Lemma A.1. *The forking diagrams between (T,gc)-reductions and LRPgc-reductions are the following:*



Proof. The diagrams for $a \neq gc$ can be derived from the appendix in [21]. For $s \xrightarrow{LRPgc,gc} s'$, the bottom reduction may consist of $0, 1, 2$ gc-reductions. A typical example for the latter case is:



Lemma A.2. *Let $s \xrightarrow{T,gc} t$, and $s \downarrow$. If s has an* LRP*-reduction with n LCSC-reductions, then this also holds for t.*

Proof. Let $s \xrightarrow{LRPgc,*} s'$ be an LRPgc-reduction such that s' is a WHNF, and let $s \xrightarrow{T,gc} t$. We show that claim by induction on (i) the number if $LCSC$-reductions of s to a WHNF, (ii) on the μ_{lll}-measure and (iii) on the syntactical size.

- If $s \xrightarrow{LRPgc,a} s'$ where a is a $LCSC$-reduction, then we can apply the induction hypothesis to s', and the reduction sequence to t', and obtain using diagram (1) and (4) that the number of $LCSC$-reductions of s' is the same as for s.
- If $s \xrightarrow{LRPgc,lll} s'$, then we can apply the induction hypothesis to s', and thus also to t'. From diagrams 1 and 2, we obtain that the number of $LCSC$-reductions is the same for s, t.
- If $s \xrightarrow{LRPgc,gc} s'$, then we can apply the induction hypothesis to s'. For diagram 1, the reasoning is as above. For diagram 2, we can apply the induction hypothesis to s' and t and obtain the claim. For diagram 3, the claim is obvious. □

Proposition A.3. *The calculus* LRP *is convergence-equivalent to LRPgc. I.e. for all expressions s:* $s\!\downarrow \iff s\!\downarrow_{LRPgc}$.

Proof. Let $s \downarrow_{LRPgc}$. Then it is sufficient to argue as for LR by induction on the number of normal-order reductions: If s is a WHNF, then the claim is trivial. If $s \xrightarrow{a,nogc} s_1$, then there are two cases: (i) $s \xrightarrow{LRPgc,a} s_1$ with $a \neq gc$ and we can apply the induction hypothesis to obtain $s \downarrow$. (ii) $s \xrightarrow{gc} s_1$. By induction we obtain $s_1 \downarrow$, and since (gc) is correct in LR, we also obtain $s \downarrow$.

For the reversed implication, let s be an expression with $s \downarrow$. We have to construct a LRPgc-reduction to a WHNF. The induction is on the number of $LCSC$-reductions of s, then on the measure μ_{lll}, and then on the size of s seen as syntax tree. The decrease of the measure μ_{lll} for (lll)- and for (gc)-reductions is in Lemma 3.3.

1. s is a WHNF. Then there is a sequence $s \xrightarrow{LRPgc,gc,*} s'$, where s' is a LRPgc-WHNF.
2. $s \xrightarrow{no} s_1$, and the reduction is also a LRPgc-reduction. Then obviously $s_1\!\downarrow$. If the reduction is a $LCSC$-reduction, then we can apply the induction hypothesis to s_1. If the reduction is a (lll)-reduction, then the expression s_1 is smaller w.r.t. μ_{lll}, and we can apply the induction hypothesis to s_1.
3. The third case is $s \xrightarrow{LRPgc,gc} s_1$. Then either $\mu_{lll}(s) > \mu_{lll}(s_1)$, or $\mu_{lll}(s) = \mu_{lll}(s_1)$, and the syntactical size is strictly decreased, and $s_1\!\downarrow$ by Lemmas 3.3 and A.2. Then we can apply the induction hypothesis and obtain a LRPgc-reduction of s to a WHNF. □

Corollary A.4. *Contextual equivalence and preorder coincide for* LRP *and LRPgc.*

B Space properties of (cpS) and its forking diagrams

Lemma B.1. *A complete set of forking diagrams for (cpS) in S-contexts is in Fig 9.*

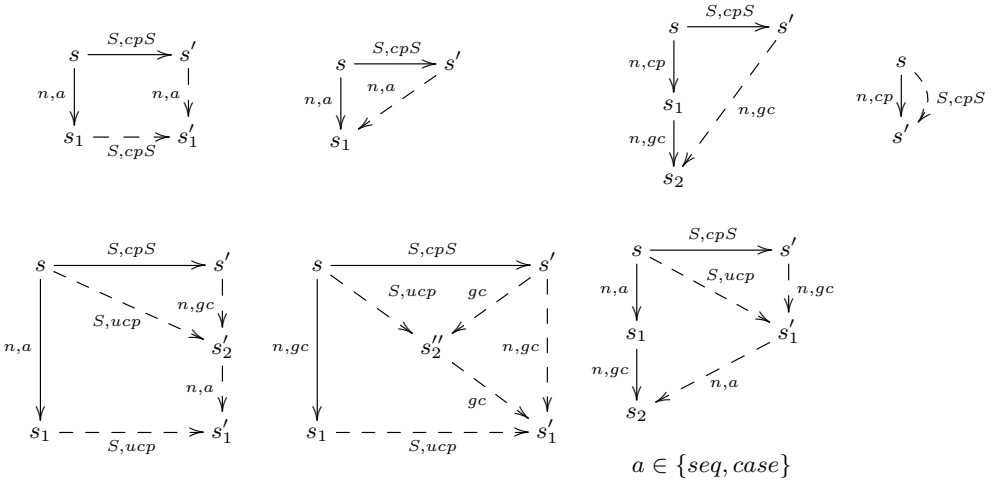


Fig. 9. Forking diagrams for (cpS) in S-contexts

Proof. We have to take into account that (cpS) may trigger a garbage collection, but only in the case that corresponds to an (ucp)-transformation. The case analysis is then straightforward.

Proposition B.2. *The translation* (S, cpS) *increases max-space at most by* `size`(v)*, where* v *is the copied abstraction.*

Proof. of Proposition 7.3:
Let $s \xrightarrow{S,cpS} s'$, where v is the copied abstraction. We use the diagrams in Lemma B.1 (i.e. Fig. 9) and prove the claim by induction on the following measure of s: (i) the number of $LCSC$-reductions $\mu_1(s)$, (ii) the measure $\mu_{lll}(s)$, and (iii) the measure $\texttt{synsize}(s)$.

If s is a LRPgc-WHNF, then s' is also a LRPgc-WHNF, and the claim holds, since $\texttt{size}(s) = \texttt{size}(v) = \texttt{size}(s')$. Now s has a LRPgc-reduction, and we check each applicable diagram in turn. If diagram 1 applies, then the induction hypothesis applies to s_1, and $\texttt{size}(s) + \texttt{size}(v) = \texttt{size}(s')$. This implies $spmax(s') \leq spmax(s) + \texttt{size}(v)$. If diagram 2, 3 or 4 applies, then the computation is similar as above but easier. If diagram 5 applies, then $spmax(s) = spmax(s')$ and $spmax(s_1) = spmax(s'_1)$ by Theorem 6.3. We have $spmax(s) = \max(\texttt{size}(s), spmax(s_1))$ and $spmax(s') = \max(\texttt{size}(s'), \texttt{size}(s'_2), spmax(s_1))$. Hence the claim holds in this case. Reasoning for diagram 6 is almost the same. If diagram 7 applies, then $spmax(s) = \max(\texttt{size}(s), spmax(s_1))$ and $spmax(s') = \max(\texttt{size}(s'), \texttt{size}(s'_1), spmax(s_2))$. Hence the claim holds. □

C Proof of the (ucp)-Diagrams

Lemma C.1. *The diagrams in Fig. 8 are a complete set of forkings diagrams for* (T, ucp).

Proof. The diagrams are adapted from [21]. There are two adaptations: (i) in diagram 3 the case $\texttt{letrec}\ x = \lambda y.r, E\ \texttt{in}\ R[x] \to \texttt{letrec}\ E\ \texttt{in}\ R[\lambda y.r]$ with an application of (cp) followed by a (gc) is covered; (ii) in diagram 7, also an intermediate (gc) may be triggered: a prototypical case is:
$\texttt{letrec}\ x = (\texttt{letrec}\ E_1\ \texttt{in}\ c), E_2\ \texttt{in}\ (\texttt{letrec}\ E_3\ \texttt{in}\ x) \xrightarrow{ucp} \texttt{letrec}\ E_2\ \texttt{in}\ (\texttt{letrec}\ E_3\ \texttt{in}\ (\texttt{letrec}\ E_1\ \texttt{in}\ c)) \xrightarrow{n,llet} \texttt{letrec}\ E_2, E_3\ \texttt{in}\ (\texttt{letrec}\ E_1\ \texttt{in}\ c))$, leading to $\texttt{letrec}\ E'_2, E'_3, E'_1\ \texttt{in}\ c))$, and the left sequence: $\xrightarrow{llet} \texttt{letrec}\ x = c, E_1, E_2\ \texttt{in}\ (\texttt{letrec}\ E_3\ \texttt{in}\ x) \xrightarrow{gc} \texttt{letrec}\ x = c, E_1, E'_2\ \texttt{in}\ (\texttt{letrec}\ E_3\ \texttt{in}\ x)$ leading to $(\texttt{letrec}\ x = c, E'_2, E'_3, E'_1\ \texttt{in}\ x)$.
We explicitly mention the cases for diagrams 4, 5 and 6:
(4): $\texttt{letrec}\ x = a, E\ \texttt{in seq}\ (c\ x)\ b \xrightarrow{ucp} \texttt{letrec}\ E\ \texttt{in seq}\ (c\ x)\ b \xrightarrow{n,seq} \texttt{letrec}\ E\ \texttt{in}\ b$.
The left sequence is: $\xrightarrow{n,seq} \texttt{letrec}\ x = a, E\ \texttt{in in}\ b \xrightarrow{n,gc} \texttt{letrec}\ E\ \texttt{in}\ b$.
(5): $\texttt{letrec}\ x = c\ r_i, E\ \texttt{in case}\ x\ (c\ y_i \texttt{->} a) \xrightarrow{ucp} \texttt{letrec}\ E\ \texttt{in case}\ (c\ r_i)\ (c\ y_i \texttt{->} a) \xrightarrow{n,case} \texttt{letrec}\ E\ \texttt{in letrec}\ y_i = r_i\ \texttt{in}\ a)$. The left sequence is:
$\xrightarrow{n,case} . \xrightarrow{n,gc} \texttt{letrec}\ z_i = r_i, E\ \texttt{in letrec}\ y_i = z_i\ \texttt{in}\ a) \xrightarrow{cpx \vee gc,*} (\texttt{letrec}\ z_i = r_i, E\ \texttt{in}\ a[z_i/y_i])$.

(6): $\texttt{letrec}\ x = c\ r_i, E\ \texttt{in seq}\ x\ a \xrightarrow{ucp} \texttt{letrec}\ E\ \texttt{in seq}\ (c\ r_i)\ a \xrightarrow{n,case} (\texttt{letrec}\ E\ \texttt{in}\ a)$. The left sequence is: $\xrightarrow{n,case} . \xrightarrow{n,gc} \texttt{letrec}\ E\ \texttt{in}\ a)$.

D (gc) is a Space-Improvement

We prove a part of Theorem 5.4:

Proposition D.1. *The reduction (gc) is a maxspace improvement.*

Proof. Let $s_0 \xrightarrow{gc} t_0$ in the empty context. We show that the context lemma for max-space can be applied: Let R be a reduction context. Then we consider the reductions of $R[s_0]$ and $R[t_0]$. Obviously $\texttt{size}(R[s_0]) \geq \texttt{size}(R[t_0])$. The diagrams in Lemma A.1 can be applied to show that $spmax(R[s_0]) \geq spmax(R[t_0])$ by induction on the number of LRPgc-reductions.

Let $s = R[s_0]$ and $t = R[t_0]$. Then $s \xrightarrow{T,gc} t$. The conditions $\texttt{size}(s_0) \geq \texttt{size}(t_0)$ and $FV(s_0) \supseteq FV(t_0)$ are satisfied. We show that $spmax(s) \geq spmax(t)$ by induction on the following measure: (i) the number of $LCSC$-reductions, (ii) the measure $\mu_{lll}()$, and (iii) the measure $\texttt{synsize}(s)$.

We make a case analysis along the diagrams in Lemma A.1.

1. If s is a gcWHNF, then t, since $s \xrightarrow{gc} t$, and $spmax(s) \geq spmax(t)$ holds.

2. If $s \xrightarrow{LRPgc,gc} s'$ and $s \xrightarrow{gc} t$ are the same reductions (i.e., the situation of diagram 3), then clearly $spmax(s) \geq spmax(s')$ holds.
3. If $s \xrightarrow{LRPgc,a} s'$ for $a \in LCSC$, then for diagram 1 we can apply the induction hypothesis along the chain $s' \xrightarrow{T,gc,*} t'$ using Lemma A.2, and obtain $spmax(s') \geq spmax(t')$. Since $\texttt{size}(s) \geq \texttt{size}(t)$, we obtain $spmax(s) \geq spmax(t)$ in this case. For diagram 4, we also get $spmax(s') \geq spmax(t')$, and also $\texttt{size}(s) \geq \texttt{size}(t) \geq \texttt{size}(t_1)$ and thus the claim holds.
(Note that $spmax(s) \geq spmax(s')$ may be wrong for (cp)-reductions.)
4. If $s \xrightarrow{LRPgc,lll} s'$, then diagrams 1,2 are relevant. For diagram 1: Since the measure μ_{lll} gets strictly smaller, we can apply the induction hypothesis to s', and since gc-reductions keeps the $LCSC$-number and do not increase the μ_{lll}-measure, the induction hypothesis can be applied to the reduction chain and we obtain $spmax(s') \geq spmax(t')$. Also, $\texttt{size}(s) \geq \texttt{size}(t)$, and the claim $spmax(s) \geq spmax(t)$ is proved. For diagram 2: the reasoning is similar, but a bit simpler.
5. If $s \xrightarrow{LRPgc,gc} s'$, then diagrams 1,2,3 are relevant. We can apply the induction hypothesis, since the $LCSC$- number is not changed in the diagram, the μ_{lll}-measure decreases along the reductions in the diagram, and also the syntactical size is strictly decreased from s to s'. Hence we can apply the induction hypothesis, to s', and obtain $spmax(s') \geq spmax(t')$. Then it is easy to see that $spmax(s) \geq spmax(t)$.
In case of diagram 2, the reasoning is similar but simpler

An application of the context lemma 4.8 now shows the claim.

E (lll) is a space improvement

E.1 Forking Diagrams of lll and justification

First of all, a complete set of forking diagrams for (lll) in the calculus LR is of the form

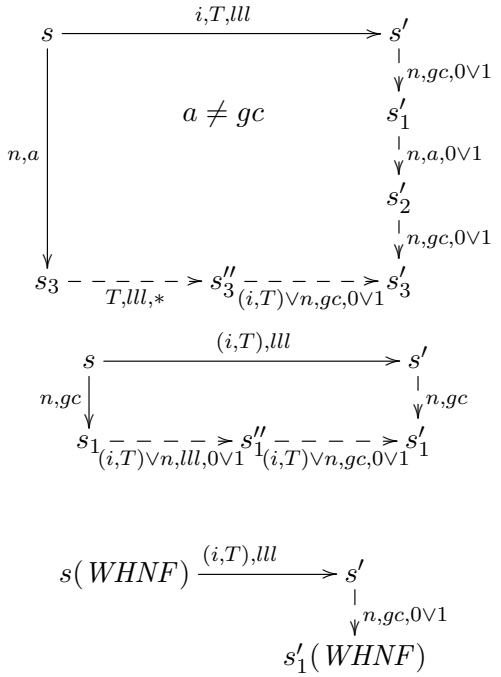
$$
\begin{array}{lll}
\xleftarrow{n,lbeta} . \xrightarrow{lll} & \rightsquigarrow \xrightarrow{lll} . \xleftarrow{n,lbeta} \\
\xleftarrow{n,cp} . \xrightarrow{lll} & \rightsquigarrow \xrightarrow{lll} . \xleftarrow{n,cp} \\
\xleftarrow{n,lll} . \xrightarrow{lll} & \rightsquigarrow \xrightarrow{lll} . \xleftarrow{n,lll} \\
\xleftarrow{n,lll} . \xrightarrow{lll} & \rightsquigarrow . \\
\xleftarrow{n,lll} . \xrightarrow{lll} & \rightsquigarrow \xrightarrow{lll} . \xrightarrow{lll} . \xleftarrow{n,lll} \\
\xleftarrow{n,case} . \xrightarrow{lll} & \rightsquigarrow \xrightarrow{lll} . \xleftarrow{n,case} \\
\xleftarrow{n,case} . \xrightarrow{lll} & \rightsquigarrow \xleftarrow{n,case} \\
\xleftarrow{n,seq} . \xrightarrow{lll} & \rightsquigarrow \xrightarrow{lll} . \xleftarrow{n,seq} \\
\xleftarrow{n,seq} . \xrightarrow{lll} & \rightsquigarrow \xleftarrow{n,seq} \\
Answer. \xrightarrow{lll} & \rightsquigarrow Answer
\end{array}
$$

The double $\xrightarrow{lll}$ occurs in the case $(\texttt{letrec}\ E_1\ \texttt{in}\ (\texttt{letrec}\ E_2\ \texttt{in}\ r))\ r' \xrightarrow{lll} (\texttt{letrec}\ E_1; E_2\ \texttt{in}\ r)\ r'$.

Lemma E.1. *An (llet)-transformation may enable a (LRPgc,gc1) but not a (LRPgc,gc2)*

Proof. The only situation where this occurs occurs when a letrec-environment is shifted to the top, and if the shifted letrec environment contains garbage.

Lemma E.2. *A complete set of forking diagrams for (lll) w.r.t. space improvement can be summarized in three diagrams where we label non-LRPgc reductions with i and label the reductions in top contexts with T.*



E.2 Proof that (lll)-transformations are space improvements

Proof. We will apply the context lemma for top-contexts (see Proposition 4.8 and 4.10). Therefore we will employ forking diagrams for (lll) in top-contexts from Lemma E.2. Let s be a closed expression with $s{\downarrow_{LRPgc}}$ and $s \xrightarrow{T,lll} s'$. We show by induction that for all top-contexts T, we have $spmax(T[s']) \leq spmax(T[s])$. The measure for induction is (μ_1, μ_2, μ_3), ordered lexicographically, where μ_1 is the number of $LRPgc, LCSC$-reductions of s to a WHNF, and μ_2 is μ_{lll} and μ_3 is the syntactical size.

The first two assumptions of the context lemma are satisfied, i.e. $\texttt{size}(s) \geq \texttt{size}(s')$ and $FV(s) \supseteq FV(s')$, which can be easily checked.
In the following we use invariances and properties:
(i) Proposition D.1 shows that (gc) is a space improvement, (ii) Lemma A.2 shows that (gc) leaves μ_1 invariant) and (iii) in [21], Theorem 2.14 it is proved that (lll) does not change μ_1.
We now go through the cases according to the diagrams.

- If s is a gcWHNF, then either s is a gcWHNF or one (LRPgc,gc) is sufficient to turn it into a gcWHNF. In both cases, $spmax(s) \geq spmax(s')$ or $spmax(s) \geq spmax(s'_1)$.
- If $a \in \{(lbeta), (case), (seq)\}$, then we can use induction and the above properties. The induction hypothesis is applicable to s_3, since $\mu_1(s) > \mu_1(s_3) \geq \mu_1(s''_3) = \mu_1(s'_3)$. This shows that s'_3 improves s''_3 which in turn improves s_3. We have $\texttt{size}(s) \geq \texttt{size}(s_3)$ as well as $\texttt{size}(s) \geq \texttt{size}(s') \geq \texttt{size}(s'_1) \geq \texttt{size}(s'_2) \geq \texttt{size}(s'_3)$. Hence also $spmax(s) \geq spmax(s')$.
- If $a = (cp)$, then we can also use the property. Again, the induction hypothesis can be applied to show that s'_3 improves s''_3 which in turn improves s_3.
  We have to take into account that (cp) increases the size. Looking at the diagrams and the case analyses, we see that $\texttt{size}(s) \geq \texttt{size}(s'_1)$, since the (gc) of s removes at least the same bindings as the (gc) from s'. Since the cp-reductions copy the same expression, we have $\texttt{size}(s_3) \geq \texttt{size}(s'_2)$. Thus we have shown that also in this case $spmax(s) \geq spmax(s')$.
- If $a = (lll)$, then the measure of s_3, s''_3, s'_3 is strictly smaller than s, since lll does not change μ_1, but strictly decreases μ_2. By induction s''_3 space-improves s_3, and s'_3 improves s''_3. Since (lll) does not increase the $\texttt{size}$, we obtain $spmax(s) \geq spmax(s')$ also in this case.
- If the first $LRPgc$-reduction is a (gc), then the second diagram holds, and we can apply induction, since (gc) does not increase μ_1 nor μ_2, but strictly decreases the syntactical size.

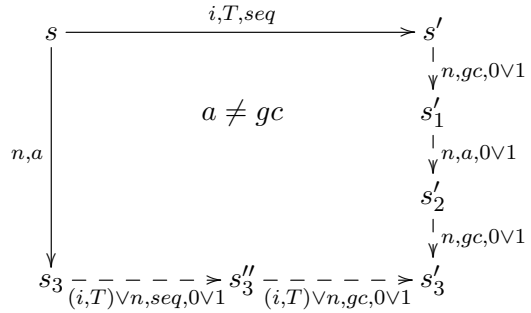
Finally, we can apply the context lemma 4.8 and obtain the claim that s' max-space-improves s.

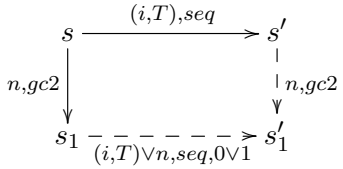
F Justification that (seq) is a space-improvement

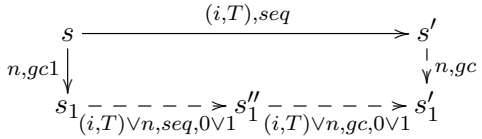
The first subsection proves a set of forking diagrams and the space improvement property, and relies on subsection F.2, where a detailed analysis of a number of cases is made.

F.1 (seq) is a Space Improvement

Lemma F.1. *The forking diagrams for* `seq` *w.r.t. space improvement can be summarized in three diagrams where we label non-LRPgc reductions with i and label the reductions in top contexts with T,*







Proof. See subsection F.2 for the detailed cases.

We prove a part of Theorem 5.4:

Proposition F.2. *The seq-reduction is a space improvement.*

Proof. This is already proved in the case of a (seq-c) in Theorem 5.3.
For the general case of a seq-reduction we will apply the context lemma for top-contexts (see Proposition 4.8 and Proposition 4.10). Therefore we will employ forking diagrams for `seq` in top-contexts from Lemma F.1. Let s be a closed expression with $s{\downarrow_{LRPgc}}$ and $s \xrightarrow{T,seq} s'$. We show by induction on the following measure that $s' \leq_{spmax} s$ by showing that for all top-contexts T, we have $spmax(T[s']) \leq spmax(T[s])$. The measure for induction is (μ_1, μ_2, μ_3), ordered lexicographically, where μ_1 is the number of $LRPgc, LCSC$-reductions of s to a WHNF, and μ_2 is μ_{lll} and μ_3 is the syntactical size.

The first two assumptions of the context lemma are satisfied, i.e. $\texttt{size}(s) \geq \texttt{size}(s')$ and $FV(s) \supseteq FV(s')$, which can be easily checked.
In the following we use the following invariances and properties:
(i) Proposition D.1 shows that (gc) is a space improvement, (ii) Lemma A.2 shows that (gc) leaves μ_1 invariant) and (iii) in [21] it is proved that (seq) does not increase μ_1.
We now go through the cases for the (summary) diagram.

- If s is a gcWHNF, then s' is also a gcWHNF and so $\texttt{size}(s) = spmax(s) \geq spmax(s') = \texttt{size}(s')$.
- $s \xrightarrow{T,seq} s'$ is an LRPgc-reduction step, then $spmax(s) = spmax(s')$ by definition.

- If $a \in \{(lbeta), (case), (seq)\}$, and the LRPgc-reduction is different from the transformation, then we can use induction and the above properties. We are in the situation of diagram 1. The induction hypothesis is applicable to s_3, since $\mu_1(s) > \mu_1(s_3) \geq \mu_1(s_3'') = \mu_1(s_3')$. This shows that s_3' improves s_3'' which in turn improves s_3. We have $\texttt{size}(s) \geq \texttt{size}(s_3)$ as well as $\texttt{size}(s) \geq \texttt{size}(s') \geq \texttt{size}(s_1') \geq \texttt{size}(s_2') \geq \texttt{size}(s_3')$. Hence also $spmax(s) \geq spmax(s')$.
- If $a = (cp)$, then we can also use the property. Again, the induction hypothesis can be applied to show that s_3' improves s_3'' which in turn improves s_3.
  We have to take into account that (cp) increases the size. Looking at the diagrams and the case analyses, we see that $\texttt{size}(s) \geq \texttt{size}(s_1')$, since the (gc) of s removes at least the same bindings as the (gc) from s'. Since the cp-reductions copy the same expression, we have $\texttt{size}(s_3) \geq \texttt{size}(s_2')$. Thus we have shown that also in this case $spmax(s) \geq spmax(s')$.
- If $a = (lll)$, then by induction s_3'' space-improves s_3, and s_3' improves s_3''. Since (lll) does not increase the `size`, we obtain $spmax(s) \geq spmax(s')$ also in this case.
- If the first *LRPgc*-reduction step is a $(gc2)$, then the second diagram holds, and we can apply induction, since (gc) does not increase μ_1 nor μ_2, but strictly decreases the syntactical size.
- If the first *LRPgc*-reduction step is a $(gc1)$, then the third diagram holds, and we can apply induction, since (gc) does not increase μ_1 nor μ_2, but strictly decreases the syntactical size. Then the `size` decreases along the reductions, and we can reason as before.
- If the first *LRPgc*-reduction step is a $(seqgc)$, then the fourth diagram is active. Since $\texttt{size}(s) \geq \texttt{size}(s')$ and $spmax(s') \geq spmax(s_1)$, the claim $spmax(s) \geq spmax(s')$ holds.

  Finally, we can apply the context lemma 4.8 and obtain the claim that s' max-space-improves s.

F.2 A Detailed Justification of Forking Diagrams of seq

We justify the complete set of forking diagrams for `seq` w.r.t. space improvement.

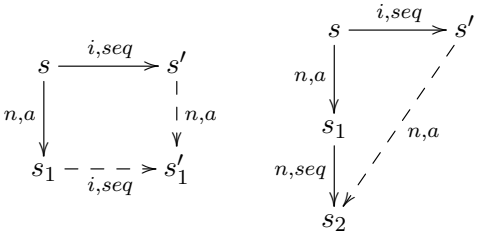
Lemma F.3. *The forking diagrams for* `seq` *w.r.t. space improvement can be summarized in the diagrams in Lemma F.1. where we write n for LRPgc and i for non-LRPgc reductions, see diagrams in Lemma F.1.*

Proof. We check all possibilities of $s_1 \xleftarrow{LRPgc,a} s \xrightarrow{T,\texttt{seq}} s'$ for closed expressions s, where the `seq`-reduction is not a *LRPgc*-reduction.

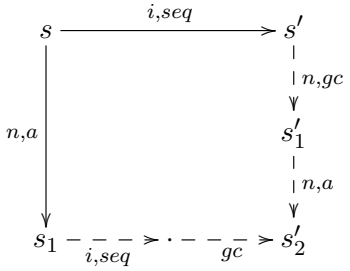
- The expression s is not a `letrec`-expression. Then closing the reduction is represented by a square diagram. The reason is that s does not admit a $\xrightarrow{LRPgc,gc}$-reduction.

$$\begin{array}{ccc} s & \xrightarrow{i,seq} & s' \\ {\scriptstyle n,a}\downarrow & & \downarrow{\scriptstyle n,a} \\ s_1 & \overset{i,seq}{\dashrightarrow} & s_1' \end{array}$$

  In the following we can assume that s is a `letrec`-expression.
- $s \xrightarrow{LRPgc,a} s_1$, i.e. the first reduction is not a garbage collection, and a is an (lbeta), (seq), (cp) or (case)-reduction.
  Then $s = \texttt{letrec}\ env_1; env_2\ \texttt{in}\ r$, where env_1 is garbage after the seq-reduction. If the `seq`-reduction does not enable a $\xrightarrow{LRPgc,gc}$-reduction, then the following two diagrams are sufficient.



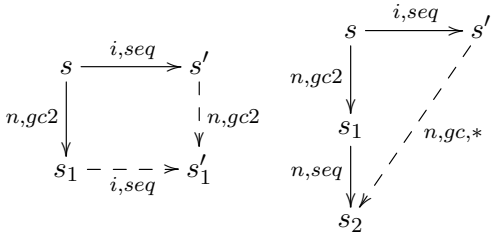
If `seq` enables a $\xrightarrow{LRPgc,gc}$-reduction, then the diagram is:



An example is

$$
\begin{array}{l}
\quad \texttt{letrec}\ x_1 = 0; x_2 = \texttt{seq}\ x_1\ 0\ \texttt{in}\ (\lambda x.x)\ x_2 \\
\xrightarrow{n,lbeta} \texttt{letrec}\ x_1 = 0; x_2 = \texttt{seq}\ x_1\ 0\ \texttt{in}\ (\texttt{letrec}\ x = x_2 \texttt{in}\ x) \\
\xrightarrow{i,seq} \texttt{letrec}\ x_1 = 0; x_2 = 0\ \texttt{in}\ (\texttt{letrec}\ x = x_2\ \texttt{in}\ x) \\
\xrightarrow{gc} \texttt{letrec}\ x_2 = 0\ \texttt{in}\ (\texttt{letrec}\ x = x_2\ \texttt{in}\ x) \\
\hline
\xrightarrow{i,seq} \texttt{letrec}\ x_1 = 0; x_2 = 0\ \texttt{in}\ (\lambda x.x)\ x_2 \\
\xrightarrow{n,gc} \texttt{letrec}\ x_2 = 0\ \texttt{in}\ (\lambda x.x)\ x_2 \\
\xrightarrow{n,lbeta} \texttt{letrec}\ x_2 = 0\ \texttt{in}\ (\texttt{letrec}\ x = x_2\ \texttt{in}\ x)
\end{array}
$$

– $s \xrightarrow{LRPgc,gc2} s'$. Then the diagram for reductions is as follows:

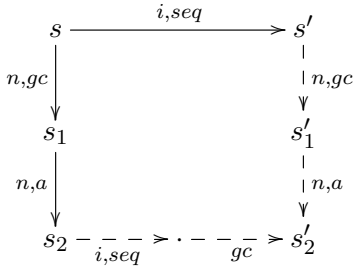


An example is

$\texttt{letrec}\ x_1 = 0\ \texttt{in}\ C[\texttt{seq}\ x_2\ 0] \xrightarrow{n,gc2} C[\texttt{seq}\ x_2\ 0] \xrightarrow{i,seq} C[0]$; and $\cdot \xrightarrow{i,seq} \texttt{letrec}\ x_1 = 0\ \texttt{in}\ C[0] \xrightarrow{n,gc2} C[0]$

!! In the following cases s is a `letrec`-expression, and the first *LRPgc*-reduction of s is not a (gc2).

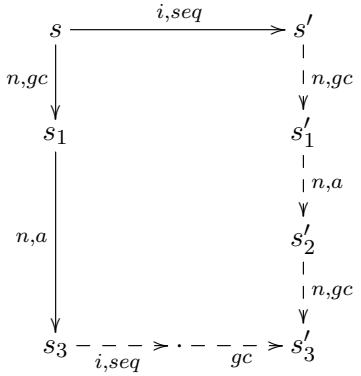
– If a is an (lbeta), (seq), (cp) or (case)-reduction,
and the first *LRPgc*-reduction is a gc, seq enables a (gc), the a-reduction does not enable a gc, then the diagram is:



An example is

$$
\begin{array}{rl}
 & \texttt{letrec } x_0 = 0; x_1 = 0; x_2 = \texttt{seq } x_1\ 0 \texttt{ in } (\lambda x.x)\ x_2 \\
\xrightarrow{n,gc} & \texttt{letrec } x_1 = 0; x_2 = \texttt{seq } x_1\ 0 \texttt{ in } (\lambda x.x)\ x_2 \\
\xrightarrow{n,lbeta} & \texttt{letrec } x_1 = 0; x_2 = \texttt{seq } x_1\ 0 \texttt{ in } \\
 & \qquad (\texttt{letrec } x = x_2 \texttt{in } x) \\
\xrightarrow{i,seq} & \texttt{letrec } x_1 = 0; x_2 = 0 \texttt{ in } (\texttt{letrec } x = x_2 \texttt{ in } x) \\
\xrightarrow{gc} & \texttt{letrec } x_2 = 0 \texttt{ in } (\texttt{letrec } x = x_2 \texttt{ in } x) \\
\hline
\xrightarrow{i,seq} & \texttt{letrec } x_0 = 0; x_1 = 0; x_2 = 0 \texttt{ in } (\lambda x.x)\ x_2 \\
\xrightarrow{n,gc} & \texttt{letrec } x_2 = 0 \texttt{ in } (\lambda x.x)\ x_2 \\
\xrightarrow{n,lbeta} & \texttt{letrec } x_2 = 0 \texttt{ in } (\texttt{letrec } x = x_2 \texttt{ in } x)
\end{array}
$$

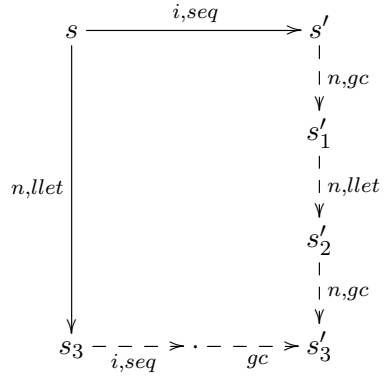
– If a is an (lbeta), (seq) or (case)-reduction.
and the first *LRPgc*-reduction is a gc, seq enables a (gc), the a-reduction enables a gc (only a seq or case are possible), then the diagram is:



An example is

$$
\begin{array}{rl}
 & \texttt{letrec } x_0 = 0; x_1 = 0; x_2 = \texttt{seq } x_1\ 0\ ; x_3 = 0 \\
 & \qquad \texttt{in } (\texttt{seq } x_3\ x_2) \\
\xrightarrow{n,gc} & \texttt{letrec } x_1 = 0; x_2 = \texttt{seq } x_1\ 0; x_3 = 0 \\
 & \qquad \texttt{in } (\texttt{seq } x_3\ x_2) \\
\xrightarrow{n,seq} & \texttt{letrec } x_1 = 0; x_2 = \texttt{seq } x_1\ 0\ ; x_3 = 0 \texttt{ in } x_2 \\
\xrightarrow{i,seq} & \texttt{letrec } x_1 = 0; x_2 = 0; x_3 = 0 \texttt{ in } x_2 \\
\xrightarrow{gc} & \texttt{letrec } x_2 = 0 \texttt{ in } x_2 \\
\hline
\xrightarrow{i,seq} & \texttt{letrec } x_0 = 0; x_1 = 0; x_2 = 0\ ; x_3 = 0 \texttt{ in } (\texttt{seq } x_3\ x_2) \\
\xrightarrow{n,gc} & \texttt{letrec } x_2 = 0\ ; x_3 = 0 \texttt{ in } (\texttt{seq } x_3\ x_2) \\
\xrightarrow{n,seq} & \texttt{letrec } x_2 = 0\ ; x_3 = 0 \texttt{ in } x_2 \\
\xrightarrow{n,gc} & \texttt{letrec } x_2 = 0\ ; \texttt{in } x_2
\end{array}
$$

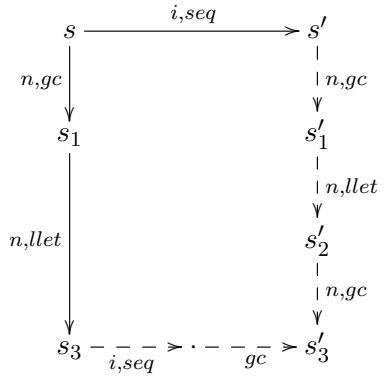
– If a is a (llet)-reduction, the first LRPgc-reduction step is (llet), and seq may enable a gc. Then the diagram is:



An example is

$$
\begin{array}{rl}
& \texttt{letrec } x_1 = 0; x_2 = \texttt{seq } x_1\ 0\ ; \texttt{in letrec } x_3 = 0; x_4 = 0 \\
& \qquad \texttt{in } x_3\ x_2 \\
\xrightarrow{n,llet} & \texttt{letrec } x_1 = 0; x_2 = \texttt{seq } x_1\ 0\ ; x_3 = 0; x_4 = 0 \texttt{ in } x_3\ x_2 \\
\xrightarrow{i,seq} & \texttt{letrec } x_1 = 0; x_2 = 0; x_3 = 0; x_4 = 0 \texttt{ in } x_3\ x_2 \\
\xrightarrow{gc} & \texttt{letrec } x_2 = 0; x_3 = 0 \texttt{ in } x_3\ x_2 \\
\hline
\xrightarrow{i,seq} & \texttt{letrec } x_1 = 0; x_2 = 0\ ; \texttt{in letrec } x_3 = 0; x_4 = 0 \texttt{ in } x_3\ x_2 \\
\xrightarrow{n,gc} & \texttt{letrec } x_2 = 0 \texttt{ in letrec } x_3 = 0; x_4 = 0 \texttt{ in } x_3\ x_2 \\
\xrightarrow{n,llet} & \texttt{letrec } x_2 = 0\ ; x_3 = 0; x_4 = 0 \texttt{ in } x_3\ x_2 \\
\xrightarrow{n,gc} & \texttt{letrec } x_2 = 0\ ; x_3 = 0 \texttt{in } x_3\ x_2
\end{array}
$$

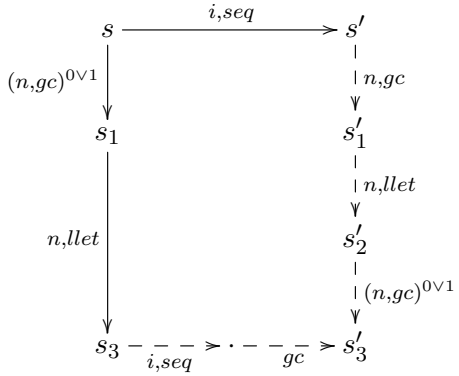
– If a is a (llet)-reduction, the first LRPgc-reduction step is (gc), and seq may enable a gc. then the diagram is:



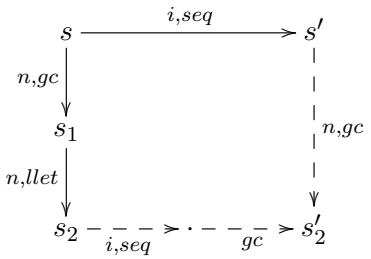
An example is

$$
\begin{array}{rl}
& \texttt{letrec } x_0 = 0; x_1 = 0; x_2 = \texttt{seq } x_1\ 0 \texttt{ in} \\
& \qquad \texttt{letrec } x_3 = 0; x_4 = 0 \texttt{ in } x_3\ x_2 \\
\xrightarrow{n,gc} & \texttt{letrec } x_1 = 0; x_2 = \texttt{seq } x_1\ 0 \texttt{ in} \\
& \qquad \texttt{letrec } x_3 = 0; x_4 = 0 \texttt{ in } x_3\ x_2 \\
\xrightarrow{n,llet} & \texttt{letrec } x_1 = 0; x_2 = \texttt{seq } x_1\ 0\ ; x_3 = 0; x_4 = 0 \texttt{ in } x_3\ x_2 \\
\xrightarrow{i,seq} & \texttt{letrec } x_1 = 0; x_2 = 0; x_3 = 0; x_4 = 0 \texttt{ in } x_3\ x_2 \\
\xrightarrow{gc} & \texttt{letrec } x_2 = 0; x_3 = 0 \texttt{ in } x_3\ x_2 \\
\hline
\xrightarrow{i,seq} & \texttt{letrec } x_0 = 0; x_1 = 0; x_2 = 0 \texttt{ in} \\
& \qquad \texttt{letrec } x_3 = 0; x_4 = 0 \texttt{ in } x_3\ x_2 \\
\xrightarrow{n,gc} & \texttt{letrec } x_2 = 0 \texttt{ in letrec } x_3 = 0; x_4 = 0 \texttt{ in } x_3\ x_2 \\
\xrightarrow{n,llet} & \texttt{letrec } x_2 = 0\ ; x_3 = 0; x_4 = 0 \texttt{ in } x_3\ x_2 \\
\xrightarrow{n,gc} & \texttt{letrec } x_2 = 0\ ; x_3 = 0 \texttt{in } x_3\ x_2
\end{array}
$$

– If in the example the binding $x_4 = 0$ is missing, then the reduction $s'_2 \xrightarrow{LRPgc,gc} s'_3$ can be omitted. Thus the summary of the last two diagrams is



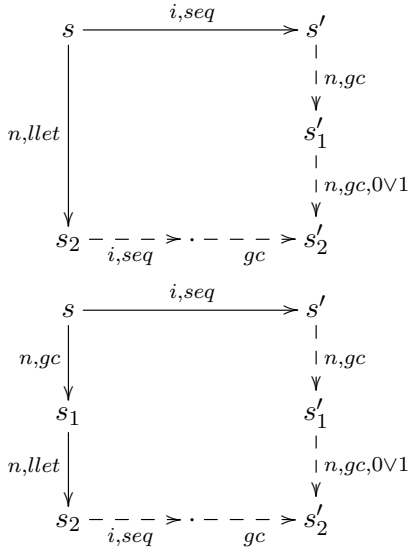
– If a is a (llet)-reduction, the first LRPgc-reduction step is (gc), and seq may enable a gc, then the diagram is:



$$
\begin{array}{l}
\quad \texttt{letrec}\ x_0 = 0;\ x_1 = 0\ \texttt{in}\ \texttt{letrec}\ x_2 = \texttt{seq}\ x_1\ 0\ \texttt{in}\ \texttt{letrec}\ x_3 = 0; x_4 = 0\ \texttt{in}\ x_3\ x_2 \\
\xrightarrow{n,gc} \texttt{letrec}\ x_1 = 0\ \texttt{in}\ \texttt{letrec}\ x_2 = \texttt{seq}\ x_1\ 0\ \texttt{in}\ \texttt{letrec}\ x_3 = 0; x_4 = 0\ \texttt{in}\ x_3\ x_2 \\
\xrightarrow{n,llet} \texttt{letrec}\ x_1 = 0;\ x_2 = \texttt{seq}\ x_1\ 0\ ;\texttt{in}\ \texttt{letrec}\ x_3 = 0; x_4 = 0\ \texttt{in}\ x_3\ x_2 \\
\xrightarrow{i,seq} \texttt{letrec}\ x_1 = 0;\ x_2 = 0\ \texttt{in}\ \texttt{letrec}\ x_3 = 0; x_4 = 0\ \texttt{in}\ x_3\ x_2 \\
\xrightarrow{gc} \texttt{letrec}\ x_2 = 0\ \texttt{in}\ \texttt{letrec}\ x_3 = 0; x_4 = 0\ \texttt{in}\ x_3\ x_2 \\
\hline
\xrightarrow{i,seq} \texttt{letrec}\ x_0 = 0;\ x_1 = 0\ \texttt{in}\ \texttt{letrec}\ x_2 = 0\ \texttt{in}\ \texttt{letrec}\ x_3 = 0; x_4 = 0\ \texttt{in}\ x_3\ x_2 \\
\xrightarrow{n,gc2} \texttt{letrec}\ x_2 = 0\ \texttt{in}\ \texttt{letrec}\ x_3 = 0; x_4 = 0\ \texttt{in}\ x_3\ x_2
\end{array}
$$

$$
\begin{array}{l}
\quad \texttt{letrec}\ x = 0\ \texttt{in}\ (\texttt{letrec}\ y = \texttt{seq}\ x\ 0\ \texttt{in}\ (\texttt{letrec}\ v = 0\ \texttt{in}\ ((\lambda z.z)\ 0))) \\
\xrightarrow{n,llet-in} \texttt{letrec}\ x = 0,\ y = \texttt{seq}\ x\ 0\ \texttt{in}\ (\texttt{letrec}\ v = 0\ \texttt{in}\ ((\lambda z.z)\ 0)) \\
\xrightarrow{i,seq} \texttt{letrec}\ x = 0,\ y = 0\ \texttt{in}\ (\texttt{letrec}\ v = 0\ \texttt{in}\ ((\lambda z.z)\ 0)) \\
\xrightarrow{gc2} \texttt{letrec}\ v = 0\ \texttt{in}\ ((\lambda z.z)\ 0) \\
\hline
\xrightarrow{i,seq} \texttt{letrec}\ x = 0\ \texttt{in}\ (\texttt{letrec}\ y = 0\ \texttt{in}\ (\texttt{letrec}\ v = 0\ \texttt{in}\ ((\lambda z.z)\ 0))) \\
\xrightarrow{n,gc2} \texttt{letrec}\ y = 0\ \texttt{in}\ (\texttt{letrec}\ v = 0\ \texttt{in}\ ((\lambda z.z)\ 0)) \\
\xrightarrow{n,gc2} \texttt{letrec}\ v = 0\ \texttt{in}\ ((\lambda z.z)\ 0)
\end{array}
$$

Fig. 10. Examples for seq-forking diagrams



Examples for these diagrams are in Fig. 10.

G Space Improvement Lemmas for case

In order to show that the case-rule as a transformation is also a space improvement, we use the variant (case*) (see Fig. 5),

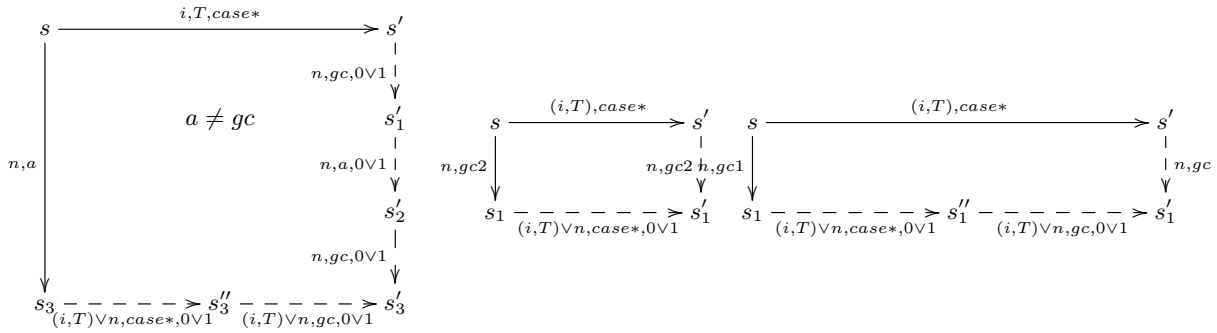


Fig. 11. A complete set forking diagrams for $\xrightarrow{T,case*}$ with LRPgc-reductions

Remark G.1. It follows from [21] that case*-transformation are correct program transformations, since the difference to a (case)-transformation consists of applications of (gc) and (cpx). I.e. it is easy to see that $s \xrightarrow{case} s_1$, and $s \xrightarrow{case*} s_1^*$, then $s_1 \xrightarrow{(gc)\vee(cpx),*} s_1^*$.

Lemma G.2. *A complete set forking diagrams for $\xrightarrow{T,case*}$ with LRPgc-reductions is in Fig. 11.*

Proof. By inspecting all the cases. Note that the diagrams for (case*) are very similar to the ones for (seq), since the effects of the overlaps are comparable.

Proposition G.3. *The transformation (case*) is a space improvement.*

Proof. The proof is almost the same as the proof for (seq) in the proof of Theorem 5.4, where seq has to be replaced by case*. The only new argument is that (case*) also decreases the `size` of expressions.

H Proof of a Proposition for gc=

Proposition H.1 (Proof of Propostiton 5.9). *The transformation (gc=) is a space-equivalence.*

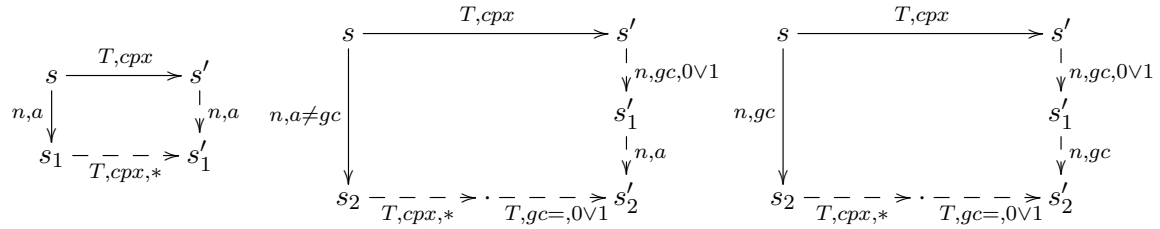
Proof. An analysis of forking overlaps between LRPgc-reductions and *gc*=-transformations, which are different, shows that they (almost) commute, i.e. i.e., $s_1 \xleftarrow{n,a} s \xrightarrow{T,gc=} s'$ can be joined by $s_1 \xrightarrow{T,gc=,0\vee1} s'_1 \xleftarrow{n,a} s'$. We will apply the context lemma for space equivalence (Proposition 4.12), which also holds for T-contexts.
Let $s_0 \xrightarrow{gc=} t_0$, and let $s = R[s_0]$ and $s' = R[t_0]$. Then $\texttt{size}(s) = \texttt{size}(s')$ as well as $FV(s) = FV(s')$. The equality $spmax(s) = spmax(s')$ can easily be shown by induction on the number of LRPgc-reductions. Then an application of the context lemma (Proposition 4.12) for space equivalences shows the claim.

I Space-Equivalence of (cpx)

Proposition I.1. *The transformation cpx is a space-equivalence.*

Proof. Due to the context lemma it is sufficient to check forking diagrams in top contexts, however, we permit that (cpx) may copy into arbitrary contexts.

An analysis of forking overlaps between LRPgc-reductions and (*cpx*)-transformations in top contexts shows that the following sets of diagrams are complete, where all concrete (cpx)-transformations in a diagram copy from the same binding $x = y$:

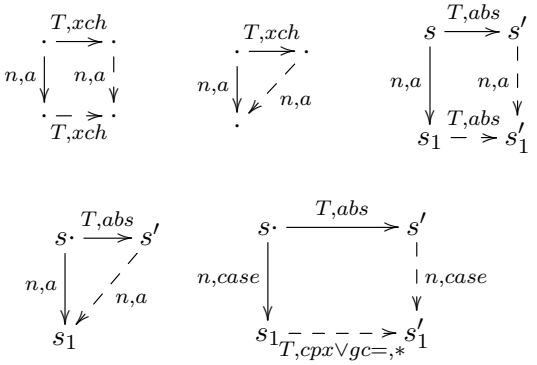


We also need the diagram-property that $s_1 \xleftarrow{n,a} s \xrightarrow{T,gc=} s'$ can be joined by $s_1 \xrightarrow{T,gc=,0\vee1} s'_1 \xleftarrow{n,a} s'$.
We will apply the context lemma for space equivalence (Proposition 4.12), which also holds for T-contexts.
Let $s_0 \xrightarrow{cpx} t_0$, and let $s = T[s_0]$ and $s' = T[t_0]$. Then $\texttt{size}(s) = \texttt{size}(s')$ as well as $FV(s) = FV(s')$.
We have to show $spmax(s) = spmax(s')$, which can be shown by an induction on the number of LRPgc-reductions of $T[s_0]$. The claim to be proved by induction is sharpened: in addition the number of LRPgc-reductions of $T[s_0]$ is not greater than for $T[t_0]$.
Since (cpx) as well as $(gc=)$ do not change the size, we have the same maximal space usage for s and s'. An application of the context lemma for top contexts and for space equivalence finishes the proof.

J Space Equivalences (xch) and (abs)

Proposition J.1 (Proof of Proposition 6.1). *(abs) and (xch) are space equivalences*

Proof. The transformations (abs) and (xch) do not change the LRPgc-WHNF property.
A complete set of forking diagrams for (xch) and (abs), respectively, in top contexts is:



These diagrams can be derived for example from the more general diagrams in [21]. These transformations keep the number of LCSC-reductions.

The same proof technique as in Proposition 5.6 will be used, i.e. the context lemma for space equivalence and induction proofs with the same measure. First the space equivalence property of (xch) is proved using the same methods as above and the context lemma for max-space improvement. The next part is the space equivalence property of (abs), which follows from space equivalence of (abs) and (cpx) by Proposition 5.6 and 5.9.

K Proof for the Transformation (caseId)

Lemma K.1. *A complete set of forking diagrams for the correct instances of the transformation (caseId) with LRPgc-reductions is in Fig. 12. This implies that whenever the starting (caseId) transformation is correct, then the other (caseId)-transformation in the diagrams is also correct.*

Proof. These are adaptations from [18].

Proof of Proposition 6.6:
Correct instances of the (caseId)-transformation are space-improvements.

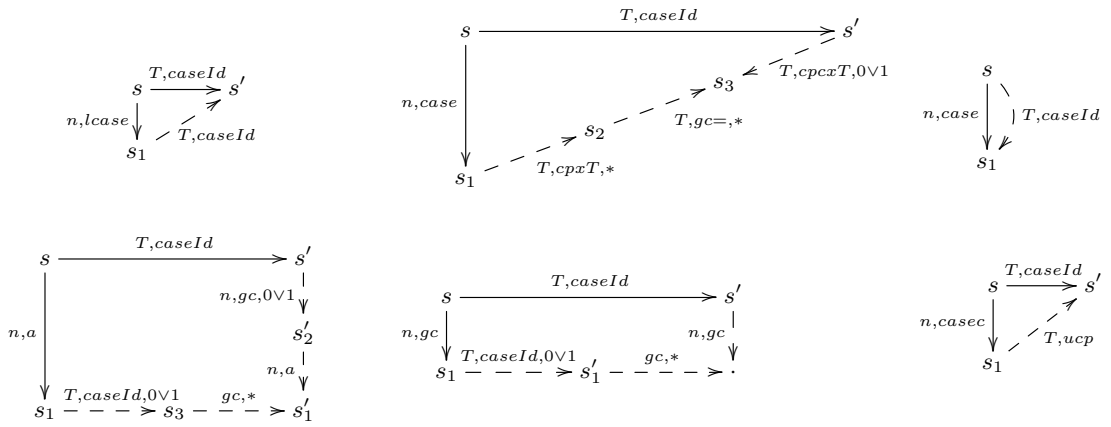


Fig. 12. Forking diagrams for (caseId)

Proof. We apply the context lemma for max-space-improvements and the diagrams in Lemma K.1. Let s, s' be expression with $s \xrightarrow{T,caseId} s'$. We show that the prerequisites for the context lemma for max-space-improvements hold: The conditions $FV(s) \supseteq FV(s')$ and $\texttt{size}(s) \geq \texttt{size}(s')$ obviously hold. We show the third condition $s' \leq_{R,spmax} s$ by induction on the following measure of s: (i) the number of $LCSC$-reductions $\mu_1(s)$, (ii) the measure $\mu_{lll}(s)$, and (iii) the measure $\texttt{synsize}(s)$. See Theorem 2.8 and Lemma 3.3 for the modification and invariances of the measures μ_1, μ_2 under reductions. If s is a LRPgc-WHNF, then s' is also a LRPgc-WHNF, and the claim holds, since $\texttt{size}(s) \geq \texttt{size}(s')$. Now s has a LRPgc-reduction, and we check each applicable diagram in turn. If diagram 1 is applicable, then the induction hypothesis can be applied to s_1, and we obtain $spmax(s_1) \geq spmax(s')$. Since $\texttt{size}(s) > \texttt{size}(s')$, this implies $spmax(s) = \max(\texttt{size}(s), spmax(s_1)) \geq spmax(s')$. If diagram 2 is applicable, then $spmax(s_1) = spmax(s_2) \geq spmax(s_3)$ by Propositions 5.6 and Theorem 5.4. Proposition 6.2 shows $spmax(s_3) \geq spmax(s')$. Since also $spmax(s) \geq spmax(s_1)$, we obtain $spmax(s) \geq spmax(s')$. In the case of diagram 3, $s \xrightarrow{n,a} s_1$ is the same reduction as $s \xrightarrow{T,caseId} s_1$, then the claim holds obviously. For diagram 4, the induction hypothesis can be applied to s_1, which shows $spmax(s_1) \geq spmax(s_3)$. Since gc is a max-space improvement by Theorem 5.4 we derive $spmax(s_3) \geq spmax(s'_1)$.

Since along LRPgc-reduction sequences, $spmax$ is decreasing, we obtain $spmax(s') \geq spmax(s'_2) \geq spmax(s'_1)$. Since also $\texttt{size}(s) \geq \texttt{size}(s')$, we obtain $spmax(s) = \max(\texttt{size}(s), spmax(s_1)) \geq \max(\texttt{size}(s'), \texttt{size}(s'_2), spmax(s'_1)) = spmax(s')$. For diagram 5, the induction hypothesis can be applied to s_1, and the rest of the computations is similar, since (gc) is a max-space improvement. For diagram 6, Theorem 5.11 and Theorem 6.3 show $spmax(s) \geq spmax(s_1) \geq spmax(s')$ and the claim is proved. Finally we apply the context lemma 4.8 for max-space improvements which proves the claim.

L Space Properties of cpcx

Lemma L.1. *The transformation (cpcx) does not change the LRPgc-WHNF property. A complete set of forking and commuting diagrams for $(T, (cpcxT))$ is in Fig. 13.*

Proof. The diagrams can be derived using the cases and examples in [21], and omitting the diagram, where the copy target is within an abstraction. Observe that (cpcxT) may trigger a new garbage collection, which have to be covered by the diagrams. The diagrams are explicit, and have more arrows for more information on the the cases, since we want to prove space properties.
Diagram 1 is the non-interfering case, also with (n,gc) and the case where the $x = c\ \vec{s}$-binding is removed.
Diagram 2 covers the case where the $x = c\ \vec{s}$-binding will become garbage after the copy.
Diagram 3 covers the case where the target position of (cpcxT) is removed by the (no,a)-reduction.

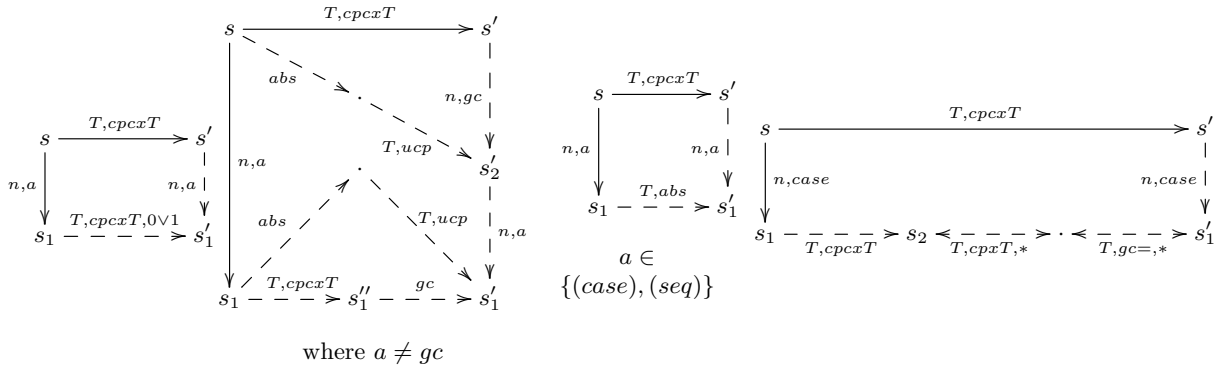


Fig. 13. Forking diagrams for (cpcxT)

Diagram 4 covers the case where (cpcxT) copies into the variable-chain of the normal-order case-reduction, or somewhere else.

Clearly, (cpcxT) is not a max-space improvement, but for the improvement property of (caseId), we need that the reverse of (cpcxT) has this property in top contexts.

Proposition L.2. *[This is Lemma 6.2]*
If $s \xrightarrow{T,cpcxT} s'$*, then* $spmax(s) \leq spmax(s') \leq spmax(s) + 1$.

Proof. The inequality is proved by an induction argument using the diagrams as commuting ones (see Fig. 13). We show the claim by induction on the following measure of s: (i) the number of $LCSC$-reductions of s', (ii) the measure $\mu_{lll}(s')$, and (iii) the measure $\texttt{synsize}(s')$.

If s, s' are LRPgc-WHNFs, then the claim holds, since $\texttt{size}(s) + 1 = \texttt{size}(s')$. If s is a LRPgc-WHNF, and s' is not an LRPgc-WHNF, then only one binding of size 1 may be garbage collected, hence $spmax(s) + 1 = spmax(s')$.
Otherwise, s is LRPgc-reducible, and we have to check all diagrams.
Assume the first diagram is applicable. If $s_1 = s'_1$, then there are two cases: (i) $spmax(s_1) \geq \texttt{size}(s)+1$. Then $spmax(s_1) = spmax(s) = spmax(s')$. (ii) $spmax(s_1) \leq \texttt{size}(s)$. Then $spmax(s) = \texttt{size}(s)$ and $spmax(s') = \texttt{size}(s) + 1$.
If $s_1 \neq s'_1$, then the induction hypothesis is applicable. We obtain $spmax(s) \leq spmax(s') \leq spmax(s) + 1$ from the maximum computation.
In the case of the second diagram Theorem 6.3 and Proposition 6.1 (J.1) show that $spmax(s) = spmax(s'_2)$ and $spmax(s_1) = spmax(s'_1)$. Since $\texttt{size}(s') = \texttt{size}(s) + 1$, we obtain $spmax(s) \leq spmax(s') \leq spmax(s) + 1$.
In the case of the third diagram we obtain $spmax(s_1) = spmax(s'_1)$ by Proposition 6.1 (J.1), and since $\texttt{size}(s') = \texttt{size}(s) + 1$, this implies the claim.
In the fourth diagram, the induction hypothesis can be applied. This together with Propositions 5.6 and 5.9 show $spmax(s_1) \leq spmax(s_2) \leq spmax(s_1) + 1$ and $spmax(s_2) = spmax(s'_1)$. Maximum computations show the claim. □

M Space Improvements for Append as a Recursive Function

We show as an example that for the append-function, written `++`, the improvement relation $((xs \texttt{ ++ } ys) \texttt{ ++ } zs) \geq_{spmax} (xs \texttt{ ++ } (ys \texttt{ ++ } zs))$ holds, where xs, ys, zs are variables. These expressions are contextually equivalent.

More exactly, we can show that in reduction-contexts and under the further restriction that only the lists are evaluated, the spmax-difference is a constant: 4.

The definition of append (denoted as `++`) is:
$\texttt{++} = \lambda xs.\lambda ys.\texttt{case } xs \texttt{ of Nil -> } ys; x : xxs \texttt{ -> } x : (\texttt{++ } xss\ ys)$

The context lemma and the already known space equivalences will be used to show the improvement relation. It is sufficient to show the claim in reduction contexts.

Let us assume that we start with the same reduction context R.

Then $R[((xs \texttt{ ++ } ys) \texttt{ ++ } zs)]$ has to be compared with $R[(xs \texttt{ ++ } (ys \texttt{ ++ } zs))]$. We use induction on the number of recursive expansions of `++`.
We will tacitly shift the let-environments to the top using the space-equivalence property of (lll) and (ucp).
First we compute the left-hand side:
(1L) The body of append is copied: $R[((xs \texttt{ ++ } ys) \texttt{ ++}_{body} zs)]$.
(2L) 2 lbeta-reduction steps: $R \cup \{xs_1 = (xs \texttt{ ++ } ys), ys_1 = zs\}$ `in` $(\texttt{case } xs_1 \ldots)$.
(2L2) copying append: $R \cup \{xs_1 = (xs \texttt{ ++}_{body} ys), ys_1 = zs\}$ `in` $(\texttt{case } xs_1 \ldots)$.
(2L3) 2 lbeta: $R \cup \{xs_1 = (\texttt{case } xs_2 \ \ldots), xs_2 = xs, ys_2 = ys, ys_1 = zs\}$ `in` $(\texttt{case } xs_1 \ldots)$.
(1R) The body of append is copied: $R[(xs \texttt{ ++}_{body} (ys \texttt{ ++ } zs))]$.
(2R) 2 lbeta-reduction steps: $R \cup \{xs_1 = xs, ys_1 = (ys \texttt{ ++ } zs)\}$ `in case` $xs_1 \ldots$).
(3NT) If the local evaluation of xs does not terminate, then the improvement relation holds, irrespective of the size.
(3NilL) If xs locally evaluates to the empty list, then the reduction result is $R \cup \{xs_1 = ys_2, xs_2 = xs, ys_2 = ys, ys_1 = zs\}$ `in` $(\texttt{case } xs_1 \ldots)$.
(3NilR) If xs locally evaluates to the empty list, then the reduction result is $R \cup \{xs_1 = xs, ys_1 = (ys \texttt{ ++ } zs)\}$ `in` ys_1). The next step is: $R \cup \{xs_1 = xs, ys_1 = (ys \texttt{ ++}_{body} zs)\}$ `in` ys_1), and then $R \cup \{xs_1 = xs, ys_1 = (\texttt{case } ys \texttt{ Nil -> } zs; \ldots\}$ `in` ys_1), which is the same as for the left hand side.
(3NilRL) Comparison: the L-expression has a space maximum $r + 14$, and the right one a space maximum of $r + 10$. The latter steps do not contribute to the space maximum, thus the claim holds in this case.
(3CL) If xs locally evaluates to $a : as$, then $R \cup \{xs_1 = a : (as \texttt{++} ys), ys_1 = zs\}$ `in` $(\texttt{case } xs_1 \ldots)$. The next step is $R[(a : ((as \texttt{ ++ } ys) \texttt{ ++ } zs))]$. Note that xs_1 is garbage collected.
(3CR) If xs locally evaluates to $a : as$, then $R[a : (as \texttt{ ++ } (ys \texttt{ ++ } zs))]$.
Now we can use induction on the number of steps.

Finally, the context lemma shows that claim $((xs \texttt{ ++ } ys) \texttt{ ++ } zs) \geq_{spmax} (xs \texttt{ ++ } (ys \texttt{ ++ } zs)$.
Note that the example in [8] where the maxspace-difference is linear, the hole of the context is within an abstraction.

N Space Usage of Two Sum Variants

We reconsider the definitions of three variants of `sum` of a list, and the space analysis in [8]. If the list is unevaluated, then the different sum-versions below would change the evaluation order of parts of the expressions, which has the potential to introduce space leaks. Hence, we avoid this complication for our analysis and use a fully evaluated list $[1, \ldots, n]$ of integers as argument for the sum-functions.

For the analysis, we make further simplifications: We assume that positive integers are available (for example Peano-integers) and assume that an integer occupies a space unit of 1. We also assume that addition $(+)$ is a strict function in two arguments and that $n + m$ immediately returns the result without using extra space. These simplification are justified, since we are only interested in analyzing the recursive variant in comparison with the tail recursive variants, and since we could also use Boolean values or constants for numbers.

$$
\begin{aligned}
\texttt{sum} &= \lambda xs.\texttt{case } xs \texttt{ of Nil -> } 0; y : ys \texttt{ -> } y + (\texttt{sum } ys) \\
\texttt{sum}' &= \lambda xs.\texttt{asum } 0 \ xs \\
\texttt{asum} &= \lambda a.\lambda \ xs.\texttt{case } xs \texttt{ of Nil -> } a; \\
&\qquad y : ys \texttt{ -> asum } (a + y) \ ys \\
\texttt{sum}'' &= \lambda xs.\texttt{asum}' \ 0 \ xs \\
\texttt{asum}' &= \lambda a.\lambda \ xs.\texttt{case } xs \texttt{ of Nil -> } a; \\
&\qquad y : ys \texttt{ -> let } a' = a + y \texttt{ in seq } a' \ (\texttt{asum}' \ a' \ ys)
\end{aligned}
$$

Let us assume that the definitions of the functions are in the letrec environment, and then we compare `sum` $[1,\ldots,n]$, `sum'` $[1,\ldots,n]$ and `sum''` $[1,\ldots,n]$. We also assume that the input environment including the list is not garbage collected during the evaluation.

We analyze the space usage for an empty list first:

- (sum): `sum Nil` $\longrightarrow$ $\texttt{sum}_{body}$ `Nil`:
  $\longrightarrow$ `letrec` $xs = $ `Nil in case` xs `of Nil ->` $0; y : ys$ `->` $y +$ `sum` $ys \longrightarrow 0$.
  The maximal space, but only for the expressions without the top- let and without the input list: $8 + 1 = 9$.

- (sum'): `sum' Nil` $\longrightarrow$ $\texttt{sum'}_{body}$ `Nil` $\longrightarrow$
  `letrec` $xs = $ `Nil in asum` $0\ xs \longrightarrow$
  `letrec` $xs = $ `Nil in` $\texttt{asum}_{body}\ 0\ xs$
  `letrec` $xs = $ `Nil`$; a = 0; xs' = xs$ `in case` xs' `of Nil ->` $a; y : ys$ `->` `asum` $(y + a)\ ys$
  The result is 0. The maximal space, without the top- let and without the input list: is 9 for the body of `asum`, and 1 for the application, i.e. $9 + 1 = 10$.
- (sum"): The analysis is the same, the size is $10 + 1 = 11$.

If the lists are not empty, then the analysis results in intermediate steps:

- `sum` $[i,\ldots,n]$:
  `letrec` $xs = [i,\ldots,n]$ `in` $1 + (2 + \ldots$
  $+($`case` xs `of Nil ->` $0; y : ys$ `->` $y +$ `sum` $ys))$
  The maximum sized expression without the input list is $1 + (2 + \ldots + (n$
  $+(\lambda xs.($`case` xs `of Nil ->` $0; y : ys$ `->` $y +$ `sum` $ys))$ `Nil`
  The size is $2 * n + 2 + (size(\texttt{sum}_{body})) = 2 * (n + 1) + 8$.
- $\texttt{sum}'$ $[i,\ldots,n]$:
  `letrec` $xs = [i,\ldots,n]$ `in asum` $0\ xs$
  `letrec` $xs = [i,\ldots,n]; a = 0$ `in case` xs `of Nil ->` $a; y : ys$ `->` `asum` $(y + a)\ ys$
  The maximum sized expression (without the input list) is:
  `letrec` $\ldots; xs_n = $ `Nil`
  `in` $\texttt{asum}_{body}\ (\ldots(0 + 1) + 2 \ldots + n)\ xs_n$.
  The size is $2 * n + 2 + 9 = 2 * (n + 1) + 9$
- $\texttt{sum}''$ $[i,\ldots,n]$:
  The maximum occurs in the expression:
  `letrec` $ys = $ `Nil`$; a' = m$ `in seq` a' $(\texttt{asum}'_{body}\ a'\ ys)$, counted without the `Nil`:
  The size is $1 + 1 + 2 + 9 = 13$.
  Thus the maximal size in this case is a constant.

O Computations for Examples

We sketch the argumentation for space improvement property of two example transformations.

O.1 `map` $\lambda x.x$ vs. `id`

First we have to check contextual equality of the expressions:

This could be done by applicative bisimulation in the call-by-name variant of the calculus LRP (see [16]). We are sure that the applicative bisimulation also in the Kleene-restriction is a sufficient criteron for contextual equivalence. The proof is by standard methods. Since LRP and *LRPgc* are equivalent, this also holds in *LRPgc* .

We have to check the cases:

1. If xs is $\bot$, then the equivalence holds.

2. If $xs = \texttt{Nil}$, then again equivalence holds by simple computation.
3. If $xs = a : ss$, then $\texttt{map}\ (\lambda x.x)\ (a : as)$ reduces to $a : \texttt{map}\ (\lambda x.x)\ as$, and the right hand side to $a : as$, and we can go on with the applicative bisimulation.

In order to check whether the space improvement property holds, we use the context lemma for space improvement. Let R be a reduction context.
We compare $R[\texttt{map}\ (\lambda x.x)\ xs]$ and $R[(\lambda x.x)\ xs]$.
If xs is not convergent, then $\geq_{R,spmax}$ holds.
If the left hand side reduces as $R[\texttt{map}_{body}\ (\lambda x.x)\ xs]$, then this reduces to $R[\texttt{letrec}\ f = (\lambda x.x); xs' = xs\ \texttt{in case}\ xs' \ldots]$.
If xs reduces to the empty list, then the spmax of the right hand side is smaller than spmax of the left hand side.
If xs reduces to $a : as$, then we also see that the space of the left hand side is greater than that of the right hand side, and by induction, spmax of the lhs is greater then that of the rhs.

O.2 `foldr (:) []` vs. `id`

Correctness can be proved using applicative simulation in the call-by-name variant of LRP

We have to check the cases:

1. If xs is $\bot$, then the equivalence holds, since the functions are strict in the argument.
2. If $xs = \texttt{Nil}$, then again equivalence holds by simple computation.
3. If $xs = a : ss$, then $\texttt{foldr}\ (:)\ []\ (a : as)$ reduces to $a : \texttt{foldr}(:)\ []\ as$, and the right hand side to $a : as$. This is sufficient for the applicative bisimulation.

In order to check whether the space improvement property holds, we use the context lemma for space improvement. Let R be a reduction context.
We compare $R[\texttt{foldr}\ (:)\ []\ xs]$ and $R[(\lambda x.x)\ xs]$.
If xs is not convergent, then $\geq_{R,spmax}$ holds.
If the left hand side reduces as $R[\texttt{foldr}_{body}\ (:)\ []\ xs]$; then this reduces to $R[\texttt{letrec}\ f = (:); e = []; xs' = xs\ \texttt{in case}\ xs' \ldots]$. If xs reduces to the empty list, then the spmax of the right hand side is smaller than spmax of the left hand side.
If xs reduces to $a : as$, then we also see that the space of the left hand side is greater than that of the right hand side, and by induction, spmax of the lhs is greater then that of the rhs.