Space Improvements for Total Garbage Collection

Manfred Schmidt-Schauß and Nils Dallmeyer

Goethe-University, Frankfurt, Germany

Technical Report Frank-61

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

April 15, 2019

Abstract. The focus of this paper are space-improvements of programs, which are transformations that do not worsen the space requirement during evaluations. A realistic theoretical treatment must take garbage collection method into account. We investigate space improvements under the assumption of an optimal garbage collector. Such a garbage collector is not implementable, but there is an advantage: The investigations are independent of potential changes in an implementable garbage collector and our results show that the evaluation and other similar transformations are space-improvements.

1 Introduction

Optimizing programs w.r.t. the runtime is ubiquitous in computer science and implemented in compilers. It is important to also observe the space usage of these optimizations. These may enlarge the program due to optimizations, for example by inlining, duplicating and specializing code, and also the required space during runtime may be enlarged. Our objective is to investigate the behaviour of programs w.r.t. their space usage, in particular for the optimization method that iterates the application of (small) transformations to the given program.

The target languages are call-by-need functional programming languages like Haskell [5, 2], since these permit many correct program transformations. Early work on program transformations was [7, 6], which is our paradigm for optimizations of programs. A motivation to investigate the space behavior of (runtime-optimizing) transformations is the observation [3, 4, 1] that simple correct tranformations my drastically increase the space uage. An example is (`head` xs) `eqBool` (`last` xs) vs. (`last` xs) `eqBool` (`head` xs) for a list-expression xs that generates a long list of Booleans (using the Haskell-conventions).

Work on space-improvements in call-by-need functional languages [3, 4] was based on a garbage collector that removes immediately detectable unused bindings in letrec-expressions. The same garbage collecting method was used in our investigations in [8].

The focus of this paper is the space-behaviour during evaluations. A realistic theoretical treatment must take runtime garbage collection into account. We will investigate space behaviour under the assumption of an optimal garbage collector. Such a garbage collector is clearly not implementable, but there is an advantage of this investgation: The results are independent of potential changes in an implementable garbage collector and are hepful as guidelines which transformations are safe.

The optimal garbage collection is with respect to a single evaluation and thus not correct in every context, however, we use it for measuring the space usage of correct transformations and thus obtain information on the true requirement of space.

The **contributions and results** of this paper are: A definition (Def. 3.3) of the optimum of garbage collectable positions and a space measure `size` for expressions and *sps* for complete evaluations, that do not count the garbage collectable expressions w.r.t. this optimal garbage collector.

It also does not take into account the specifics of representing lets and indirections. This makes the results robust against modifications of the realistic garbage collection and realistic representation of bindings in the store. We prove a context lemma for space improvement w.r.t. to an optimal garbage collector (Theorem 3.17), and prove that several transformations are space improvements or space equivalences w.r.t *sps* in Sect. 4.

The structure of this paper is to first define the calculus *LR* in Sect. 2. Sect. 3 defines the optimal garbage collector, space improvements and context lemmas. Sect. 4 contains analyses of the space behaviour of several transformations.

2 The Call-by-Need Lambda Calculus *LR*

We recall the calculus *LR* [9], which is an untyped call-by-need lambda calculus that extends the lambda calculus by recursive `letrec`, data constructors, case-expressions, and the `seq`-operator. We present the syntax and reduction rules. Omitted details and further information can be found in [9].

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

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

Definition 2.1. *A* context C *is an expression with a hole (denoted by* $[\cdot]$*) at expression position.* Surface contexts S *are contexts where the hole is not in an abstraction,* top contexts T *are surface contexts where the hole is not in an alternative of a* `case`*, and* weak top contexts *are top contexts where the hole is not in a* `letrec`*-expression. With* $C[s]$ *we denote the substitution of the hole in the context* C *by expression* s*. A* multicontext M *is an expression with zero or more (different) holes at expression positions.*

A *value* in *LR* is an abstraction $\lambda x.s$ or a constructor application $(c\ \overrightarrow{s})$. The reduction rules of *LR* are defined in Fig. 2. The rule (lbeta) is the sharing variant of classical β-reduction. The rules (cp-in) and (cp-e) copy abstractions. The rules (llet-in) and (llet-e) join two `letrec`-environments. The rules

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

Fig. 1. Expressions of the language *LR* where $x, x_i \in Var$ are term variables

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 2. Reduction rules

(lapp), (lcase), and (lseq) float-out a `letrec` from the first argument of an application, a `case`-, or a `seq`-expression. The rules (seq-c), (seq-in), and (seq-e) evaluate a `seq`-expression, provided that the first argument is a value (or a variable that is bound (via indirections) to a constructor application). The rules (case-c), (case-in), and (case-e) evaluate a `case`-expression provided that the first argument is (or is a variable which is bound to) a constructor application of the right type.

We also denote unions of rules in Fig. 2 as follows: (case) is the union of (case-c), (case-in), (case-e); (seq) is the union of (seq-c), (seq-in), (seq-e); (cp) is the union of (cp-in), (cp-e); (llet) is the union of (llet-in), (llet-e); and (lll) is the union of (llet), (lapp), (lcase), and (lseq).

We assume the distinct variable convention. Normal order reduction steps and notions of termination are defined as follows:

Definition 2.2 (Normal Order Reduction of LR).

1. *Let t be an expression. Then a single* normal order reduction step $t \xrightarrow{LR} t'$ *is defined by first undertaking a search for the next needed position (see [9]), and then one of the rules in Fig. 2 is applied.*
2. *We write $\xrightarrow{+}$ for the transitive closure and $\xrightarrow{*}$ for the reflexive-transitive closure of $\to$. We write $\xrightarrow{n}$ for exactly n $\to$-steps and we write $\xrightarrow{n \vee m}$ for either n or m steps.*

Definition 2.3. *A reduction context R is any context in which an immediate normal-order reduction step can be performed.*

Definition 2.4. *An expression s is a* weak head normal form (WHNF)*, if s is a value, or s is of the form* `letrec` *Env* `in` *v, where v is a value, or s is of the form* `letrec` *$x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^{m}, Env$* `in` *$x_m$.*

Definition 2.5. *An expression s* converges, *denoted as $s{\downarrow}$, iff there exists a WHNF t s.t. $s \xrightarrow{LR,*} t$. This may also be denoted as $s \downarrow t$. We write $s{\uparrow}$ iff $s{\downarrow}$ does not hold.*

We use contextual equivalence as our notion of program equivalence.

Definition 2.6. *Let s, t be two LR-expressions. We define* contextual equivalence $\sim_c$ *w.r.t. the operational semantics of LR: Let $s \sim_c t$, iff for all contexts $C[\cdot]$: $C[s]{\downarrow} \iff C[t]{\downarrow}$.*

A *program transformation* P is a binary relation on expressions. It is correct iff $P \subseteq \sim_c$. All introduced transformations are correct:

Proposition 2.7 ([9]). *The program transformations (see Figs. 2 and 4) (lbeta), (case), (seq), (cp), (lll), (gc), (cpx), (cpcx), (abs), (xch), and (ucp) are correct.*

3 Space Improvement for Total Garbage Collection

The idea of total garbage collection is to assume an optimal garbage collection, which means to remove all subexpressions that do not contribute to termination of the expression. It is clearly undecidable whether a subexpression is used or not. A motivation to consider this form of total garbage collection and the corresponding space improvement relation is the following. Practical garbage collectors may have various strength, depending on their capabilities to predict the non-usage of subexpressions. Our view is that all these garbage collections are approximations of differing strength to the maximal (total) garbage collection. Analyzing the improvement property of the total garbage collection (as maximum) is helpful and independent of the various implementations to garbage collect during evaluations, and thus of independent value.

The total garbage collection is defined by recognizing garbage-subexpression as follows: their replacement by (the nonterminating) constant *Bot* does not modify the convergence. Positions in terms are defined as usual as lists p of positive integers representing the path from the root to the tree-positions. $t[p \mapsto t']$ means the term that is constructed from t by replacing the position p with t'.

Definition 3.1 (garbage-collectable positions). *Let Bot be a constant that does not converge and has all types. A position p of an expression t is called* garbage-collectable *iff $t{\downarrow} \iff t[p \mapsto Bot]{\downarrow}$.*

The sequence of replacing garbage-collectable positions is irrelevant:

Lemma 3.2. *Let $p_1, \ldots, p_n$ be some garbage collectable positions of t. Then $t{\downarrow} \iff t[p_1 \mapsto Bot, \ldots, p_n \mapsto Bot]{\downarrow}$*

Proof. Let $t' := t[p_1 \mapsto Bot, \ldots, p_n \mapsto Bot]$. We have $t' \leq_c t$, hence $t'{\downarrow} \implies t{\downarrow}$. Now assume $t{\downarrow}$, and $t'{\uparrow}$. Then the normal-order reduction of t' must put some successor-position of p_k in the expressions of the reduction sequence in a reduction context, independent of the other positions. Then p_k cannot be a garbage-collectable position, which is a contradiction. Hence $t'{\downarrow}$.

Definition 3.3 (Total garbage collection, tgc). *Let s be an LR-expression. Then $s \xrightarrow{tgc} s'$ is defined by total garbage collection; i.e., $s' = s[p_1 \mapsto Bot, \ldots, p_n \mapsto Bot]$ where the p_i, $i = 1, \ldots, n$ are all minimal garbage collectable positions of s. If s is not changed, then it is in* tgc-normalform.

Remark 3.4. (tgc) is in general not correct. An example is the closed expression (`Cons True Nil`), where (`Cons True Nil`) $\xrightarrow{tgc}$ (`Cons` *Bot Bot*), but this transformation is not correct, since the context (`case` $[\cdot]$ (`Cons` $x\ y \to x$) $\ldots$) distinguishes the two expressions.

Definition 3.5. *The* `size`(s) *of an expression s is defined in Fig. 3.*

Note that the size does not count variable names, nor indirections, nor the nesting of letrecs. Insofar it is robust w.r.t. different implementations of bindings. The specialized size measure *sizetgc* does not even count the garbage collectable positions.

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

Fig. 3. Definition of size

Definition 3.6. *Let s be an expression. Then $sizetgc(s) := \texttt{size}(s')$ where $s \xrightarrow{tgc} s'$.*

Definition 3.7. *For a closed expressions s, we define $sps(s) := max\{sizetgc(s_i) \mid$ where s_i are the expressions in a normal-order reduction sequence of $s\}$.*

Definition 3.8. *Let s, t be two LR-expressions with $s \sim_c t$. If for all contexts C, we have $sps(C[s]) \leq sps(C[t])$, then we say s* totally-space-improves *t (or s is a total space-improvement of t) notation $s \leq_{sps} t$.*
If for all contexts C, we have $sps(C[s]) = sps(C[t])$, then we say s and t are totally-space-equivalent*. These notions are also extended and used for transformations.*

This notion applies to functions that are defined in a library and are then optimized or compiled

We also define a weaker variant that only applies in the immediate evaluation situations: Here we have in mind the optimizations of a compiler once the program is completely known.

Definition 3.9. *We consider the calculus LR. Let s, t be two expressions with $s \sim_c t$. If we have $sps(s) \leq sps(t)$, then we say s* opportunistically total-space improves *t (or s is an* opportunistic total space-improvement *of t).*

In contrast to the standard notion of space-improvement as in [4, 3, 8] $s \leq_{sps} t$ does not imply $\texttt{size}(s) \leq \texttt{size}(t)$ and also not $FV(s) \subseteq FV(t)$, which is an advantage, since this permits a more general definition and view of space consumption and space improvements, and it permits more interesting transformations. However, it implies $sizetgc(s) \leq sizetgc(t)$:

Lemma 3.10. *If s is a total space improvement of t, i.e. $s \leq_{sps} t$, then $sizetgc(s) \leq sizetgc(t)$.*

Proof. We show that if $sizetgc(s) > sizetgc(t)$ and $s \sim_c t$, then in general s cannot be a total space improvement of t.
Let us assume that s is a space-improvement of t, and let $m = sps(t) - sps(s) \geq 0$.
Let $s_0 := (\texttt{seq}^*\ s \ldots s)$ and $t_0 := (\texttt{seq}^*\ t \ldots t)$, where seq^* is an iterated seq, and the number of occurrences of $m+2$. Then $sps(s_0) = sps(s) + (m+1) * (1 + \texttt{size}(s))$, since the normal-order reduction first only modifies the leftmost s, then normal-order-reduces $(\texttt{seq}^*\ s \ldots s)$ with one s-occurrence removed. The same holds for t_0: $sps(t_0) = sps(s) + (m+1) * (1 + \texttt{size}(t))$. Since $\texttt{size}(s) > \texttt{size}(t)$, we obtain $sps(s_0) - sps(t_0) = -m + (m+1) * (\texttt{size}(s)) - \texttt{size}(t) > 0$, which is a contradiction to the assumption that s is a space-improvement of t.

Note that (tgc) is in general not a space improvement, since it is not correct (see Remark 3.4). We will show below that standard (gc) and also more general forms of garbage collection are total space improvements.

Definition 3.11. *Let (gcg) be a (garbage collecting) transformation that is an approximation of (tgc), i.e. $(gcg) : s \to s'$ holds, iff there is a set of positions $q_1, \ldots, q_n$, where every position q_i is the same or below some garbage collectable position and $s' = s[q_1 \mapsto \texttt{Bot}, \ldots, q_n \mapsto \texttt{Bot}]$.*

Theorem 3.12. *In any case (gcg) is an opportunistic total space improvement. If (gcg) is correct, then (gcg) is a total space improvement.*

Proof. We have to show the total space improving property.
Let s_0 be an expression and let C be a context, and $s_0 \xrightarrow{gcg} s_0'$. We look at the general reduction diagram, which is as follows:

$$\begin{array}{ccc} C[s_0] & \xrightarrow{gcg} & C[s_0'] \\ {\scriptstyle tgc}\downarrow & \swarrow & \nearrow {\scriptstyle tgc} \\ s_1 & & \end{array}$$

The reason is that the (gcg)-positions are removed by tgc. s_1 is a common expression in the reduction. We have $sizetgc(C[s_0]) \geq sizetgc(C[s_0'])$. Hence $sps(C[s_0]) \geq sps(C[s_0'])$.

It is easy to see that all normal-order reduction steps with the exception of (cp) are (total) space improvements, which is in accordance with the space improvement notions in [4, 3, 8].

3.1 A Context Lemma for Total Space-Improvements

We need a context lemma to ease the proofs that transformations are total space improvements. We will prove it in the calculus LR, but with the adapted measure *sizetgc*.

Definition 3.13. *We consider the calculus LR. Let s, t be two expressions with $s \sim_c t$. If for all LR-reduction contexts R, if $R[s], R[t]$ are closed and $sps(R[s]) \leq sps(R[t])$, then we write $s \leq_{R,sps} t$. If for all LR-reduction contexts R, if $R[s], R[t]$ are closed and $sps(R[s]) = sps(R[t])$, then we write $s =_{R,sps} t$.*

Lemma 3.14. *For every WHNF s, we have $sizetgc(s) = 1$.*

Proof. A WHNF is either a simple WHNF of the form $\lambda x.s$, $(c\ s_1 \dots s_n)$, or `letrec` *Env* `in` s, where s is a simple WHNF, or `letrec` *Env* `in` x, and x is bound in *Env* to a simple WHNF. Since after total garbage collection, the expressions are of the form $\lambda x.\texttt{Bot}$, $(c\ \texttt{Bot} \dots \texttt{Bot})$, or `letrec` *Env* `in` s', where s' is a simple garbage collected WHNF, or `letrec` *Env* `in` x, and x is bound in *Env* to a simple garbage collected WHNF. Hence $sizetgc(s)$ is 1.

Definition 3.15. *Let s, t be expressions. If for all contexts C, we have $sizetgc(C[s]) \leq sizetgc(C[t])$, then we denote this as $s \leq_{C,sizetgc} t$. If for all contexts C, we have $sizetgc(C[s]) = sizetgc(C[t])$, then we denote this as $s =_{C,sizetgc} t$.*

Lemma 3.16. *If M is a multicontext with n holes, and s_i, t_i are expressions with $s_i \leq_{C,sizetgc} t_i$ for all i, then also $M[s_1, \dots, s_n] \leq_{C,sizetgc} M[t_1, \dots, t_n]$.*

Proof. We show that claim by induction on the number n of holes. $M[s_1, \dots, s_n] \leq_{C,sizetgc} M[t_1, \dots, t_n]$ follows from $M[s_1, \dots, s_{n-1}, s_n] \leq_{C,sizetgc} M[s_1, \dots, s_{n-1}, t_n] \leq_{C,sizetgc} M[t_1, \dots, t_{n-1}, t_n]$. The first holds by assumption on s_n, t_n, and the second by the induction hypothesis for $n - 1$.

The following context lemma is similar, but more general than the context lemma in [8].

Theorem 3.17 (Context Lemma for Total Space Improvements). *If $s \sim_c t$, $s \leq_{R,sps} t$, and $s \leq_{C,sizetgc} t$ then $s \leq_{sps} t$.*

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

(I) If no hole of M is in a reduction context, then there are two cases:
(i) $M[t_1, \ldots, t_n]$ is a WHNF. The context M itself must be a WHNF, since otherwise there is a hole of M in a reduction context. Then also $M[s_1, \ldots, s_n]$ is a WHNF, and by the assumption, we have $1 = sps(M[s_1, \ldots, s_n]) \leq sps(M[t_1, \ldots, t_n])$.
(ii) The reduction step is $M[t_1, \ldots, t_n] \xrightarrow{LR,a} M'[t'_1, \ldots, t'_{n'}]$, and $M[s_1, \ldots, s_n] \xrightarrow{LR,a} M'[s'_1, \ldots, s'_{n'}]$ and the pairs (s'_i, t'_i) are renamed versions of pairs (s_j, t_j). This shows $sps(M'[s'_1, \ldots, s'_{n'}]) \leq sps(M'[t'_1, \ldots, t'_{n'}])$ by induction.
By Lemma 3.16 and the preconditions of this lemma, the inequation $sps(M[s_1, \ldots, s_n]) \leq sps(M[t_1, \ldots, t_n])$, holds, hence by computing the maximum, we obtain $sps(M[s_1, \ldots, s_n]) \leq sps(M[t_1, \ldots, t_n])$.

(II) Some t_j in $M[t_1, \ldots, t_n]$ is in a reduction position. Then there is one hole, say i, of M that is in a reduction position w.r.t. only M. 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 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: $sps(M[s_1, \ldots, s_{i-1}, t_i, s_{i+1}, \ldots, s_n]) \leq sps(M[t_1, \ldots, t_{i-1}, t_i, t_{i+1}, \ldots, t_n])$. Also by the assumption: $sps(M[s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n]) \leq sps(M[s_1, \ldots, s_{i-1}, t_i, s_{i+1}, \ldots, s_n])$, since $M[s_1, \ldots, s_{i-1}, \cdot, s_{i+1}, \ldots, s_n]$ is a reduction context. Hence $sps(M[s_1, \ldots, s_n]) \leq sps(M[t_1, \ldots, t_n])$. □

Corollary 3.18 (Context Lemma for Total Space Equivalence). *If $s \sim_c t$, $s =_{R,sps} t$, and $s =_{C,sizetgc} t$ then $s =_{sps} t$.*

Proof. Follows by applying Theorem 3.17 in both directions.

3.2 Criteria for Context Lemma Requirements

We show a criterion for $\leq_{C,sizetgc}$ of transformations. First we show a property that is used below several times in variants.

Lemma 3.19. *Let C_1, C_2 be multi-contexts, such that $C_1[s_1, \ldots, s_n] \sim_c C_2[s_1, \ldots, s_n]$ for all expressions $s_1, \ldots, s_n$. If $p = p_1 p_2$ is a garbage collectable position of $C_1[s_1, \ldots, s_n]$ that points into s_i, where $p_{1,1}$ is the position of the i^{th} hole of C_1 and p_2 is the position in s_i, then for the position $p_{2,1}$ of i^{th} hole of C_2, also $p_{2,1}p_2$ is a garbage collectable position in $C_2[s_1, \ldots, s_n]$.*

Proof. The simple argument is that $C_1[s_1, \ldots, s_n] \sim_c C_1[p_1, \ldots, p_i[p_2 \mapsto Bot], \ldots, s_n] \sim_c C_2[p_1, \ldots, p_i[p_2 \mapsto Bot], \ldots, s_n]$ and hence $C_2[p_1, \ldots, p_i[p_2 \mapsto Bot], \ldots, s_n] \sim_c C_2[s_1, \ldots, s_n]$. The same for the direction from C_2 to C_1 . Hence the claim holds.

Lemma 3.20. *Let s, t be expressions, such that $s \sim_c t$, $s = C_s[s_1, \ldots, s_n]$ and $t = C_t[s_1, \ldots, s_n]$, and $\texttt{size}(C_s) \leq \texttt{size}(C_t)$. Moreover the translation T: $C_s[r_1, \ldots, r_k]$ to $C_t[r_1, \ldots, r_k]$ is correct for all r_j. We also assume that all positions in C_s, C_t that are not the hole positions are reduction positions in the respective contexts. Then $s \leq_{C,sizetgc} t$.*

Proof. Since $s \sim_c t$, which implies $C[s] \sim_c C[t]$, the garbage-collectable positions in C are the same on the left and right hand side. Let p be a garbage-collectable position in $C[C_s[s_1, \ldots, s_n]]$ that is in s and goes down to s_1, w.l.o.g. Since the translation T is correct, the position p can be splitted into $p_1 p_{2,s} p_3$ where $p_{2,s}$ is the position of the first hole of C_s. Using Lemma 3.19 we see that also $p_1 p_{2,t} p_3$ is a garbage collectable position where $p_{2,t}$ is the position of the hole of C_t, and vice versa. For the position q_1 of s and t itself Lemma 3.19 also shows that q_1 in $C[s]$ is garbage collectable if q_1 in $C[t]$ is garbage collectable.
Hence there is a 1-1-correspondence between the garbage collectable positions of s in $C[s]$ and t in $C[t]$. As a summary, the garbage collectable positions in $C[s]$ and $C[t]$ are in 1-1- correspondence and either point to equal expressions, or the expression on the s-side is not greater in size than the one of $C[t]$. This means $sizetgc(C[s]) \leq sizetgc(C[t])$.

An example for the situation in Lemma 3.20 is the beta-reduction as transformation, i.e., $((\lambda x.s)\ t) \to \texttt{letrec}\ x = t\ \texttt{in}\ s$, and the two contexts are $((\lambda x.[]_1)\ []_2)$ and $(\texttt{letrec}\ x = []_2\ \texttt{in}\ []_1)$.

Note that the rules used in normal-order reduction are (lbeta), (case), (cp), (seq), (llet), (lapp), (lcase), and (lseq). We show that the claim of Lemma 3.16 holds for these rules with the exception of (cp), as a prerequisite for arguing on their total space improvement properties.

Lemma 3.21. *For the rules $a \in \{(lbeta), (case), (seq), (llet), (lapp), (lcase), (lseq)\}$, and expressions $s \xrightarrow{a} t$, we have $s \geq_{C,sizetgc} t$.*

Proof. We use Lemma 3.19 implicitly in the following, which implies that a the garbage collectable positions are transported by the reduction(s). For (lbeta) the preconditions of Lemma 3.16 hold, since (lbeta) is correct (note that the positions of the characteristic multicontexts of rule (lbeta) are switched). For the variants of (case) the conditions hold. For (llet), (lapp), (lcase) and (lseq) we have to formalize these rules by an infinite number of rule formats, since the bindings must be explicit to satisfy the conditions of Lemma 3.16 – there are no surprises. (seq) may delete a subexpression, but also all garbage collectable positions are eliminated, hence also this rule satisfies the preconditions. □

Note that the preconditions of Lemma 3.16 are in general not satisfied for (cp), since the resulting expression is larger in size, and in general this also holds after applying (*LRtgc*), see also Proposition 4.1.

Lemma 3.22. *For the rules $a \in \{(lbeta), (case), (cp), (seq), (llet), (lapp), (lcase)), (lseq)\}$ with $s \xrightarrow{a} t$, and executed at top, the inequation $sps(R[s]) \geq sps(R[t])$ is valid.*

Proof. The reduction step is the first one in the normal-order reduction, and the maximum is taken over all reduction steps, hence the inequation obviously holds.

4 Space-Safe and Unsafe Transformations

We now analyze the space-behaviur of several transformations. The correctness of the used transformations is shown in [9].

Proposition 4.1. *The rule (cp) is in general not a total space improvement.*

Proof. It is sufficient to present a counterexample: $\texttt{letrec}\ x = \lambda y.y\ \texttt{in}\ \texttt{seq}\ (x\ 0)\ r$ with $r = \texttt{seq}\ (x\ 0)\ (x\ 0)$. A normal-order reduction sequence has the subsequent expressions: $\texttt{seq}\ ((\lambda y.y)\ 0)\ r \to \texttt{seq}\ (\texttt{let}\ y = 0\ \texttt{in}\ y)\ r \to \texttt{let}\ y = 0\ \texttt{in}\ \texttt{seq}\ y\ r \to \texttt{let}\ y = 0\ \texttt{in}\ r \to \ldots$

In contrast, copying results in $\texttt{letrec}\ x = \lambda y.y\ \texttt{in}\ \texttt{seq}\ (x\ 0)\ (\texttt{seq}\ ((\lambda y.y)\ 0)\ (x\ 0))$, which has an sps that is strictly greater than before. Note that we have to apply *LRtgc* before measuring.

Proposition 4.2. *The reduction (cp) applied in normal-order is an opportunistic total space-improvement.*

Proof. This holds, since *sps* maximizes the size values along the normal-order reduction sequence, and the transformation is at the start of it and no contexts are involved.

We now consider several transformations that are not derived from reduction rules of the calculus. Let (gc) be the union of (gc1) and (gc2), that is a non-optimal garbage collection only working on the top-letrec and let (ucp) be the union of (ucp1), (ucp2), (ucp3).

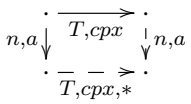
Definition 4.3. *Several transformations are defined in Fig. 4.*

Proposition 4.4. *(cpx) is a total space equivalence.*

(gc1)	$\texttt{letrec}\ \{x_i = s_i\}_{i=1}^{n}, Env\ \texttt{in}\ t \to \texttt{letrec}\ Env\ \texttt{in}\ t$ if $\forall i : x_i \notin FV(t, Env), n > 0$
(gc2)	$\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t \to t$ if for all $i : x_i \notin FV(t)$
(cpx-in)	$(\texttt{letrec}\ x = y, Env\ \texttt{in}\ C[x]) \to (\texttt{letrec}\ x = y, Env\ \texttt{in}\ C[y])$ where y is a variable and $x \neq y$
(cpx-e)	$(\texttt{letrec}\ x = y, z = C[x], Env\ \texttt{in}\ t) \to (\texttt{letrec}\ x = y, z = C[y], Env\ \texttt{in}\ t)$ (same as above)
(cpcx-in)	$(\texttt{letrec}\ x = c\ \vec{t}, Env\ \texttt{in}\ C[x]) \to (\texttt{letrec}\ x = c\ \vec{y}, \{y_i = t_i\}_{i=1}^{ar(c)}, Env\ \texttt{in}\ C[c\ \vec{y}])$
(cpcx-e)	$(\texttt{letrec}\ x = c\ \vec{t}, z = C[x], Env\ \texttt{in}\ t) \to (\texttt{letrec}\ x = c\ \vec{y}, \{y_i = t_i\}_{i=1}^{ar(c)}, z = C[c\ \vec{y}], Env\ \texttt{in}\ t)$
(abs)	$(\texttt{letrec}\ x = c\ \vec{t}, Env\ \texttt{in}\ s) \to (\texttt{letrec}\ x = c\ \vec{x}, \{x_i = t_i\}_{i=1}^{ar(c)}, Env\ \texttt{in}\ s)$ where $ar(c) \geq 1$
(xch)	$(\texttt{letrec}\ x = t, y = x, Env\ \texttt{in}\ r) \to (\texttt{letrec}\ y = t, x = y, Env\ \texttt{in}\ r)$
(ucp1)	$(\texttt{letrec}\ Env, x = t\ \texttt{in}\ S[x]) \to (\texttt{letrec}\ Env\ \texttt{in}\ S[t])$
(ucp2)	$(\texttt{letrec}\ Env, x = t, y = S[x]\ \texttt{in}\ r) \to (\texttt{letrec}\ Env, y = S[t]\ \texttt{in}\ r)$
(ucp3)	$(\texttt{letrec}\ x = t\ \texttt{in}\ S[x]) \to S[t]$ where in the three (ucp)-rules, x has at most one occurrence in $S[x]$, no occurrence in Env, t, r; and S is a surface context.

Fig. 4. Some special transformation rules

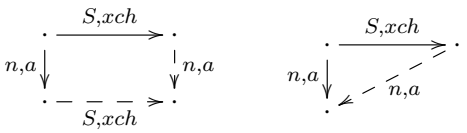
Proof. An analysis of forking overlaps between *LR*-reductions and (cpx)-transformations in top contexts shows that the following diagram is complete, where all concrete (cpx)-transformations in a diagram copy from the same binding $x = y$:



Let $s \xrightarrow{cpx} s'$. By induction on the number of LR-reductions of $T[s]$ we have $sps(T[s]) = sps(T[s'])$. We now show the requirements of the context lemma: s' might have a garbage letrec-binding from a variable to variable in contrast to s, without impact on *sizetgc* since variables are not counted by the `size`-measure. Using the diagram we see that $s =_{T,sps} s'$. Since $s \xrightarrow{cpx} s'$ does not introduce garbage that has an impact on *sizetgc* also $s =_{C,sizetgc} s'$ holds. An application of Corollary 3.18 finishes the proof.

Proposition 4.5. *(xch) is a total space equivalence.*

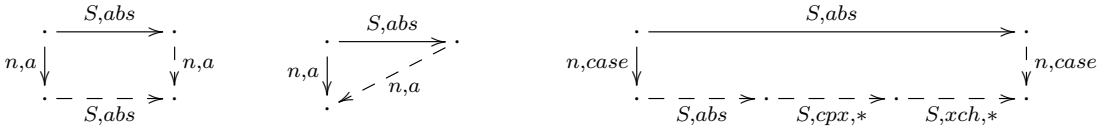
Proof. An analysis of forking overlaps between *LR*-reductions and (xch)-transformations in surface contexts shows that the following set of diagrams is complete:



Since (xch) only performs a renaming of let-variables, the garbage collection positions can be transferred directly. Also the size is unchanged, hence we can apply Corollary 3.18 for surface contexts to show that (xch) is a total space equivalence.

Proposition 4.6. *(abs) is a total space equivalence.*

Proof. An analysis of forking overlaps between *LR*-reductions and (abs)-transformations in surface contexts shows that the following set of diagrams is complete:



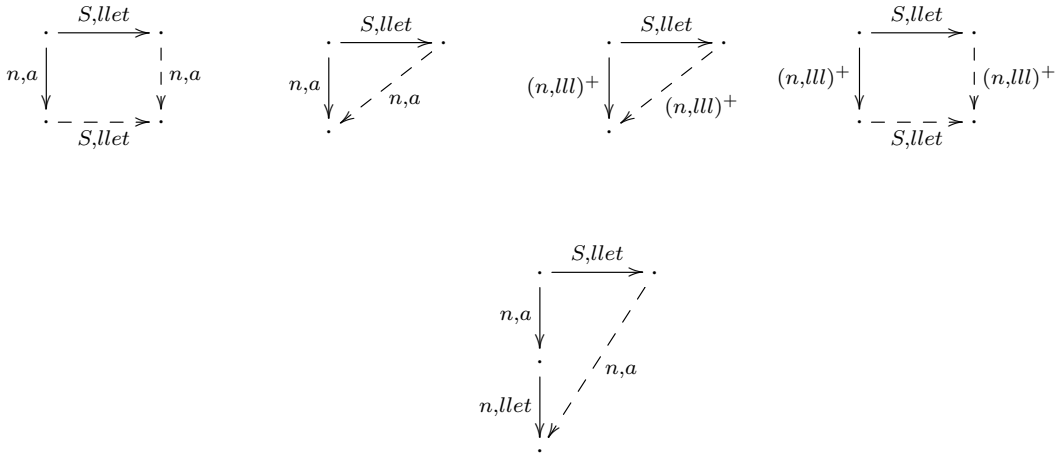
Since as above garbage positions and size remain unchanged, we use Proposition 4.4, Proposition 4.5, and Corollary 3.18 for surface contexts to show that (abs) is a total space equivalence.

Proposition 4.7. *(seq-c), (case-c), (lbeta), (lapp), (lcase), (lseq) are total space improvements.*

Proof. By considering reduction contexts we see that each of the above reductions is a normal order reduction. Also the garbage collectable positions remain unchanged for each transformation, hence we can apply Theorem 3.17 to show that these transformations are total space improvements.

Proposition 4.8. *(llet) is a total space improvement.*

Proof. The complete set of forking diagrams:



Since (llet) only moves let-bindings, (llet) does not change the size and all garbage positions can be transferred directly, hence induction using Lemma 3.21, Lemma 3.22 and Theorem 3.17 shows the claim.

Corollary 4.9. *(lll) is a total space improvement.*

Proof. Follows from Proposition 4.7 and Proposition 4.8.

Proposition 4.10. *(seq) is a total space improvement.*

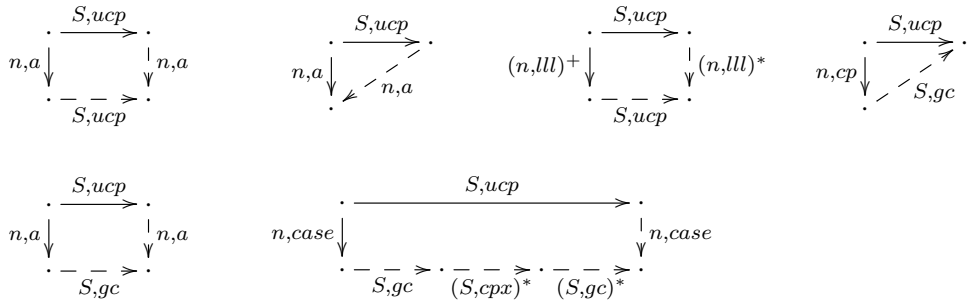
Proof. Proposition 4.7 shows that (seq-c) is a total space improvement. In the other case we use the same simulation as in [9]: If the first argument of the seq-expression is an abstraction, then we use (cp) followed by a (seq-c), where the additional space required by (cp) is directly removed by the following (seq-c) and not counted by *sps*. If the first argument of the seq-expression is a constructor application, then we use $s \xrightarrow{cpcx} \xrightarrow{seq\text{-}c} t'$ and $t \xrightarrow{abs} t'$ where again the additional space is directly removed by (seq-c) and not counted by *sps*. Hence Proposition 4.6, Lemma 3.21, Lemma 3.22 and Theorem 3.17 finish the proof.

Proposition 4.11. *(case) is a total space improvement.*

Proof. Proposition 4.7 shows that (case-c) is a total space improvement. In this proof we will only consider (case-in), since the proof for (case-e) is a copy of this proof. We use the same simulation as in [9] of (case-in): If there are no variable chains, then (cpcx) followed by a (case-c) simulates (case-in), where the additional space required by (cpcx) is directly removed by the following (case-c) and not counted by *sps*. If there are variable chains, then additionaly (cpx) is used (multiple times) to perform the needed copies for the chains. Hence Proposition 4.4, Lemma 3.21, Lemma 3.22 and Theorem 3.17 finish the proof.

Proposition 4.12. *(ucp) is a total space equivalence.*

Proof. An analysis of forking overlaps between *LR*-reductions and (ucp)-transformations in surface contexts shows that the following set of diagrams is complete:



(gc) is a total space improvement, which follows from the correctness and Theorem 3.12. Since garbage collectable positions and size remain unchanged we use Proposition 4.4 and Corollary 3.18 for surface contexts to show that (ucp) is a total space equivalence.

Theorem 4.13. *(seq), (case), (lbeta), (lll) are total space improvements, while (cpx), (xch), (abs), (ucp) are total space equivalences.*

This follows from the propositions above.

5 Conclusion and Future Work

We introduced a theoretical optimal garbage collector and analyzed the space behavior of several transformations in lazy functional languages. Because of this garbage collector the results are independent of the implementation of a specific garbage collector.
Future work is to extend the analysis to more complex transformations also taking recursion into account. The adaption of this approach for a concurrent scenario is also left for future work.

References

1. Adam Bakewell and Colin Runciman. A model for comparing the space usage of lazy evaluators. In *PPDP*, pages 151–162, 2000.
2. Haskell Community. Haskell, an advanced, purely functional programming language, 2016.
3. 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.
4. Jörgen Gustavsson and David Sands. Possibilities and limitations of call-by-need space improvement. In Benjamin C. Pierce, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001.*, pages 265–276, 2001.
5. Simon Marlow, editor. *Haskell 2010 – Language Report.* 2010.
6. 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.
7. André L. M. Santos and Simon L. Peyton Jones. On program transformation in the glasgow haskell compiler. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, UK, 6-8 July 1992*, Workshops in Computing, pages 240–251. Springer, 1992.
8. Manfred Schmidt-Schauß and Nils Dallmeyer. Space improvements and equivalences in a functional core language. In Horatiu Cirstea and David Sabel, editors, *Proceedings Fourth International Workshop on Rewriting Techniques for Program Transformations and Evaluation, WPTE@FSCD 2017, Oxford, UK, 8th September 2017.*, volume 265 of *EPTCS*, pages 98–112, 2017.
9. Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.