Space Improvements and Equivalences in a Functional Core Language

Manfred Schmidt-Schauß*
Goethe-University
Frankfurt am Main
schauss@ki.cs.uni-frankfurt.de

Nils Dallmeyer*
Goethe-University
Frankfurt am Main
dallmeyer@ki.cs.uni-frankfurt.de

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. 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 [10, 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, 12] 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 (high space usage during evaluation, which could be avoided by correctly transforming the program before evaluation). Programmers may experience this space leak 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. [6, 7, 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 $(\texttt{head}\ xs)\ \texttt{eqBool}\ (\texttt{last}\ xs)$ vs. $(\texttt{last}\ xs)\ \texttt{eqBool}\ (\texttt{head}\ xs)$ where xs is bound to an expression that generates a long list of Booleans (using the Haskell-conventions).

*supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SCHM 986/11-1.

Submitted to:
WPTE17

Early work on space improvements by Gustavsson and Sands [6, 7] 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 [6, 7] 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.

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-abstraction, 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 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. 3.3) 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 in every context. Two context lemmas (Props. 3.4 and 3.5) are proved that ease proofs of transformations being space improvements or equivalences. The main result is a classification in Section 4 of the rules of the calculus used as transformations, and of further transformations as max-space improvements and/or max-space equivalences, or as increasing the space. These results, in particular the space-equivalence (ucp) then imply that the transformation into machine expressions keeps the max-space usage which also holds for the evaluations on the abstract machine. In addition, for the (cp)-rule, which is not a max-space improvement, we show upper bounds for the max-space increase of a version of (cp) not copying into abstractions. The type-dependent rule (caseId) is shown to be a max-space improvement, which is a minimal example for the space behavior of larger class of type-dependent transformations and which are not space-improvements if types are omitted. 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.

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. [13, 9, 20], and examples for the use of transformations in optimization in functional languages are [14] and the work on Hermit [21]. A theory of (time) optimizations of call-by-need functional languages was started in [11] for a call-by-need higher order language, also based on a variant of Sestoft's abstract machine [22]. 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 [11], but not proved to be a time improvement (but it is conjectured), and which is proved correct in [17]. Hackett and Hutton [8] applied the improvement theory of [11] 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 [8] uses the same call-by-need abstract machine as [11] 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* in Section 2.1 and a variant *LRPgc* with

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:

$$\begin{array}{lll} \tau \in Typ & ::= & a \mid (\tau_1 \to \tau_2) \mid (K\ \tau_1\ \ldots\ \tau_{ar(K)}) \\ \rho \in PTyp & ::= & \tau \mid \forall a.\rho \end{array}$$

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

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

Figure 1: Syntax of expressions and types of LRP

garbage collection in Sect.2.2. Sect. 3 defines space improvements and contains the context lemmata. Sect. 4 contains a detailed treatment of space improving transformations. Finally in Sect. 5, we comment on the impact of some time-improvements on space usage, and argue on the the translation into machine language and its space-usage. Missing explanations, arguments and proofs can be found in the technical report [15].

2 Lazy Lambda Calculi, Polymorphic and Untyped

We introduce the polymorphically typed *LRP*, and the variant *LRPgc* with garbage collection, since numerous complex transformations have their nice space improving property under all circumstances (in all contexts) only in a typed language.

2.1 *LRP*: The Polymorphic Variant

We recall the polymorphically typed and extended lazy lambda calculus (*LRP*) [18, 17, 16] as 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 [20]) 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, polymorphic 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.

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 sufficiently expressive for polymorphically typed lists and functions working on such data structures.

From now on we use E 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. Bindings in `letrec`-environments can be commuted. We use $FV(s)$ and $BV(s)$ to denote free and bound variables of an expression s and $LV(E)$ to

(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) $(\texttt{seq}\ v^{\text{sub}}\ t) \to t$ 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) $(\texttt{case}_K\ c^{\text{sub}}\ \texttt{of}\ \{\ldots(c\ \texttt{->}\ t)\ldots\}) \to t$ 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) $(\texttt{letrec}\ x_1 = c^{\text{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \texttt{in}\ C[(\texttt{case}_K\ x_m^{\text{vis}}\ \texttt{of}\ \{(c\ \texttt{->}\ r)\ldots\})])$
$\to (\texttt{letrec}\ x_1 = c, \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \texttt{in}\ C[r])$ 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 in (case-in) and (case-e).

Figure 2: Basic *LRP*-reduction rules [16]

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 $[\cdot]$, $([\cdot]\ e)$, $(\texttt{case}\ [\cdot]\ \ldots)$ and $\texttt{letrec}\ x = [\cdot], y = x, \ldots\ \texttt{in}\ (x\ \texttt{True})$. Note that reduction contexts are surface as well as top-contexts. A *value* is an abstraction $\lambda x.s$, a polymorphic 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 is needed 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. (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 $s \xrightarrow{LRP} t$ *is performed (uniquely) if the labeling algorithm in [16] terminates on s inserting the (superscript) labels sub (subexpression) and vis (visited) and the applicable rule (i.e. matching also the labels) of Fig. 2 produces t.* $\xrightarrow{LRP,*}$ *is the reflexive, transitive closure,* $\xrightarrow{LRP,+}$ *is the transitive closure of* $\xrightarrow{LRP}$ *and* $\xrightarrow{LRP,k}$ *denotes k* $\xrightarrow{LRP}$*-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* `letrec` E `in` v*, where v is a value, or an expression* `letrec` $x_1 = c\ \vec{t}, \{x_i = x_{i-1}\}_{i=2}^{m}, E$ `in` x_m*. An expression s* converges *to an expression t (*$s{\downarrow}t$ *or* $s{\downarrow}$ *if we do not need t) if* $s \xrightarrow{LRP,*} t$ *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.* `letrec` $x = x$ `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* $\mathtt{rln}(s)$ *be the sum of all (lbeta)-, (case)- and (seq)-reduction steps in* $s{\downarrow}s_0$*, let* $\mathtt{rln}_{LCSC}(s)$ *be the sum of all a-reduction steps in* $s{\downarrow}s_0$ *with* $a \in LCSC$*, where* $LCSC = \{(lbeta),(case),(seq),(cp)\}$*, and let* $\mathtt{rlnall}(s)$ *be the total number of reduction steps, but not (TBeta), in* $s{\downarrow}s_0$*.*

2.2 LRPgc: LRP with Garbage Collection

As extra reduction rule in the normal order reduction we add garbage collection (gc), which is the union of (gc1) and (gc2), but restricted to the top letrec (see Fig. 3).

Definition 2.5 (*LRPgc*). *We define LRPgc, which employs all the rules of LRP and (gc) (see Fig. 3) as follows: 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 in the top letrec-environment 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 extension of *LRP*-normal-order reduction by garbage collection steps does not change the convergence and correctness:

Theorem 2.6. *The calculus LRP is convergence-equivalent to LRPgc. I.e. for all expressions s:* $s{\downarrow} \iff s{\downarrow_{LRPgc}}$*. Contextual equivalence and preorder for LRP coincide with the corresponding notions in LRPgc.*

(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 $i : x_i \notin FV(t)$

Figure 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.

Figure 4: Extra transformation rules

(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) like (case) in all other cases

(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)

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

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

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

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

Figure 5: Some special transformation rules

3 Space Improvements

From now on we use the calculus *LRPgc* as defined in Definition 2.5. We define an adapted (weaker) size measure than the size of the syntax tree, 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, since this enables proofs for the exact behavior w.r.t. our space measure for a lot of transformations.

Definition 3.1. *The size* $\texttt{size}(s)$ *of an expression* s *is defined in Fig. 6. It 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.

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

Figure 6: Definition of `size`

Definition 3.2. *The space measure $spmax(s)$ of the reduction of a closed expression s is the maximum of those $\mathtt{size}(s_i)$, 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 possible is consistent with the definition in [7].

Definition 3.3. *Let s,t be two expressions with $s \sim_c t$ and $s\downarrow$. Then s is a* space-improvement *of t, $s \leq_{spmax} t$, iff for all contexts C: if $C[s]$, $C[t]$ are closed then $spmax(C[s]) \leq spmax(C[t])$. The expression s is* space-equivalent *to t, $s \sim_{spmax} t$, iff for all contexts C: if $C[s]$, $C[t]$ are closed then $spmax(C[s]) = spmax(C[t])$. 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. Note also that $s \leq_{spmax} t$ implies $\mathtt{size}(s) \leq \mathtt{size}(t)$.

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 $spmax(R[s]) \leq spmax(R[t])$. 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 3.4 (Context Lemma for Maximal Space Improvements)**.** *If $\mathtt{size}(s) \leq \mathtt{size}(t)$, $FV(s) \subseteq FV(t)$, and $s \leq_{R,spmax} t$, then $s \leq_{spmax} t$.*

Lemma 3.5 (Context Lemma for Maximal Space Equivalence)**.** *If $\mathtt{size}(s) = \mathtt{size}(t)$, $FV(s) = FV(t)$, and $s \sim_{R,spmax} t$, then $s \sim_{spmax} t$.*

Proofs of the contetx lemmata are in [15] and are induction proofs using the *LRPgc*-normal-order reduction sequence and the syntax of contexts.

The context lemmas for max-space improvement and max-space equivalence also hold if the condition $s \leq_{X,spmax} t$, or $s \sim_{X,spmax} t$, respectively, holds where X means surface-contexts, and also if X means top-contexts.

4 Space Improving Transformations

For most of the proofs will use complete sets of forking and commuting diagrams between transformation steps and the normal-order reduction steps (see [20] 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 (see [15]).

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

Improvement	rules
$\succeq_{spmax}$	(lbeta), (case), (seq), (lll), (gc), (case*), (caseId)
$\sim_{spmax}$	(cpx), (abs), (abse), (xch), (ucp), (case-cx), (cpxT), (gc=)
$\not\preceq_{spmax}$	(cpcx), (cpS)
space-increase $O(1)$	(T,(cpcxT))
space-increase $O(A)$	(T,(cpT))
space-increase high	(cp), (cse)

Two further translations are mentioned: (cse) means common subexpression elimination; and $s \xrightarrow{gc=} s'$ 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 $x = y$.
The translation $\xrightarrow{(T,(cpcxT))}$ means (cpcxT) applied in a top-context, $\xrightarrow{(T,(cpT))}$ is (cp) applied in a top context, and A is the size of the copied abstraction by cp. The mention of "space increase high" means that in the worst case it is exponential or worse.

Note that a majority of the reasoning and proofs is done in the untyped calculi $LR, LRgc$ (see [15]).

5 Optimizations Respecting Space Usage

Our proposal for 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 filtering criterion: use only 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), as proved in [15]. Second, there are further transformation rules in Figs. fig:extra-transformation-rules-LRPgc, 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, as for example $(T,(cpT))$, and $(T,(cpcxT)$.
5. Common subexpression elimination is a time improvement as shown in [17], but it comes with a higher risk of size explosion, hence further information on the max-space-effect is required for safe optimizations.

Some program transformations that are used for optimizing running time may increase the size of the programs but also the max-space used during evaluation. The symmetric disadvantage of max-space improving program translation is that these may increase the running time. An example is common subexpression elimination: If the list generating expression $[1..n]$ occurs twice in a program and is only used for scanning, then the required space may be constant, whereas after applying common subexpression elimination, the complete list is generated in the first run and kept until the second use is finished. The required additional space is now linear in n.

Associativity of append: In [7], 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 exact analysis uses the context lemma for space improvement and in particular the space-equivalence of (ucp) which allows to inline uniquely used bindings. The proof itself uses an induction argument. The exact analysis shows that within reduction contexts, the *spmax*-difference is exactly 4.

Typed Transformations 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.

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. We consider a translation into a variant of the Sestoft machine [22, 11] extended by (eager) garbage collection. The translation ψ into machine expressions (see also [17]) in particular translates $\psi(s\,t)$ as $\texttt{letrec } y = \psi(t) \texttt{ in } (\psi(s)\ y)$, which is the same as an inverse (ucp). Our results, in particular the results on (ucp), show that the complete translation ψ is a space-equivalence.

6 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 lots of discussions and helpful hints, and reviewers for their constructive comments.

References

[1] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky & P. Wadler (1995): *A call-by-need lambda calculus*. In: *POPL'95*, ACM Press, San Francisco, California, pp. 233–246.

[2] Adam Bakewell & Colin Runciman (2000): *A model for comparing the space usage of lazy evaluators*. In: *PPDP*, pp. 151–162, doi:10.1145/351268.351287.

[3] Richard Bird (2014): *Thinking functionally with Haskell*. Cambridge University Press, Cambridge, UK.

[4] Haskell Community (2016): *Haskell, an advanced, purely functional programming language*. Available at `www.haskell.org`.

[5] Nils Dallmeyer & Manfred Schmidt-Schauß (2016): *An Environment for Analyzing Space Optimizations in Call-by-Need Functional Languages*. In Horatiu Cirstea & Santiago Escobar, editors: *Proc. 3rd*

WPTE@FSCD, *EPTCS* 235, pp. 78–92, doi:10.4204/EPTCS.235.6. Available at `http://dx.doi.org/10.4204/EPTCS.235.6`.

[6] Jörgen Gustavsson & David Sands (1999): *A Foundation for Space-Safe Transformations of Call-by-Need Programs*. *Electr. Notes Theor. Comput. Sci.* 26, pp. 69–86, doi:10.1016/S1571-0661(05)80284-1.

[7] Jörgen Gustavsson & David Sands (2001): *Possibilities and Limitations of Call-by-Need Space Improvement*. pp. 265–276, doi:10.1145/507635.507667.

[8] Jennifer Hackett & Graham Hutton (2014): *Worker/wrapper/makes it/faster*. In: *ICFP '14*, pp. 95–107.

[9] Patricia Johann & Janis Voigtländer (2006): *The Impact of seq on Free Theorems-Based Program Transformations*. *Fundamenta Informaticae* 69(1–2), pp. 63–102.

[10] Simon Marlow, editor (2010): *Haskell 2010 – Language Report*. Available at `www.haskell.org/onlinereport/haskell2010/`.

[11] A. K. D. Moran & D. Sands (1999): *Improvement in a Lazy Context: An operational theory for call-by-need*. In: *POPL 1999*, ACM Press, pp. 43–56.

[12] Andrew K. D. Moran, David Sands & Magnus Carlsson (1999): *Erratic Fudgets: A semantic theory for an embedded coordination language*. In: *Coordination '99*, *Lecture Notes in Comput. Sci.* 1594, Springer-Verlag, pp. 85–102.

[13] Simon Peyton Jones & Simon Marlow (2002): *Secrets of the Glasgow Haskell Compiler inliner*. *Journal of Functional Programming* 12(4+5), pp. 393–434.

[14] Simon L. Peyton Jones & André L. M. Santos (1998): *A Transformation-Based Optimiser for Haskell*. *Science of Computer Programming* 32(1–3), pp. 3–47, doi:http://dx.doi.org/10.1016/S0167-6423(97)00029-4.

[15] Manfred Schmidt-Schauß & Nils Dallmeyer (2017): *Space Improvements and Equivalences in a Polymorphically Typed Functional Core Language: Context Lemmas and Proofs*. Frank report 57, Institut für Informatik, Goethe-Universität Frankfurt am Main. `http://www.ki.informatik.uni-frankfurt.de/papers/frank/`.

[16] Manfred Schmidt-Schauß & David Sabel (2014): *Contextual Equivalences in Call-by-Need and Call-By-Name Polymorphically Typed Calculi (Preliminary Report)*. In M. Schmidt-Schauß, M.Sakai, D. Sabel & Y. Chiba, editors: *WPTE 2014*, *OASICS* 40, Schloss Dagstuhl, pp. 63–74.

[17] Manfred Schmidt-Schauß & David Sabel (2015): *Improvements in a Functional Core Language with Call-By-Need Operational Semantics*. In Elvira Albert, editor: *Proc. PPDP '15*, ACM, New York, NY, USA, pp. 220–231, doi:2790449.27905125.

[18] Manfred Schmidt-Schauß & David Sabel (2015): *Improvements in a Functional Core Language with Call-By-Need Operational Semantics*. Frank report 55, Institut für Informatik, Goethe-Universität Frankfurt am Main. `http://www.ki.informatik.uni-frankfurt.de/papers/frank/`.

[19] Manfred Schmidt-Schauß & David Sabel (2015): *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.

[20] Manfred Schmidt-Schauß, Marko Schütz & David Sabel (2008): *Safety of Nöcker's Strictness Analysis*. *J. Funct. Programming* 18(04), pp. 503–551, doi:10.1017/S0956796807006624.

[21] Neil Sculthorpe, Andrew Farmer & Andy Gill (2013): *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*, *Lecture Notes in Computer Science* 8241, pp. 86–103.

[22] Peter Sestoft (1997): *Deriving a Lazy Abstract Machine*. *J. Funct. Program.* 7(3), pp. 231–264.