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 for the exceptional one we show bounds on the space increase. Several further program transformations are shown to be space improvements or space equivalences, in particular the translation into machine expressions is a space equivalence. We also classify certain space-worsening transformations as space-leaks or as space-safe. 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 runtime. 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 function bodies, depending on the arguments. 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 perhaps could be avoided by correctly transforming the program before evaluation). Programmers may experience space leaks 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

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

H. Cirstea, D. Sabel (Eds.): Fourth International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE 2017)
EPTCS 265, 2018, pp. 98–112, doi:10.4204/EPTCS.265.8

space usage, where an example is (`head` *xs*) `eqBool` (`last` *xs*) vs. (`last` *xs*) `eqBool` (`head` *xs*) where *xs* is bound to an expression that generates a long list of Booleans (using the Haskell-conventions).

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 heap 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, and leads to general and good estimations.

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. Focusing on space usage enforces to include garbage collection into the model, i.e. into 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. A context lemma (Prop. 3.4) is proved that eases proofs of transformations being space improvements or equivalences. The main result is a classification in Sect. 4 of the rules of the calculus (but one) used as transformations, and of further transformations as max-space improvements and/or max-space equivalences, or as increasing max-space, or even as space-leaks. These results imply that the transformation into machine expressions keeps the max-space usage which also holds for the evaluations on the abstract machine. We also classify some space-worsening transformations as well-behaved (space-safe up to) or as space-leaks. We also argue that the typed calculus has more improvements than its untyped version. 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, time-optimizing transformations.

We discuss some **previous work** on time and space behavior for call-by-need functional languages. Examples of research on the correctness of program transformations are in [13, 9, 20], examples of the use of transformations in optimization in functional languages are in [14, 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]. 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], and proved in this paper as space leak. 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 Sect. 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 discusses space-safeness and space-leaks, and contains a detailed treatment of space improving transformations, and discusses specific examples. Sect. 5 contains experiments measuring space- and time-usage for an inlining transformation, which cannot be derived from the current theory. Missing explanations, arguments and proofs can be found in the technical report [15].

2 Polymorphic and Untyped Lazy Lambda Calculi

We introduce the polymorphically typed calculus *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. Technically, this shows up in the proofs when we have to argue over all contexts, which are strictly less than without types. For example, case analyses have to inspect less cases, in particular for list-processing functions (i.e. a smaller number and simpler forking diagrams).

2.1 *LRP*: The Polymorphic Variant

Let us recall the polymorphically typed and extended lazy lambda calculus (*LRP*) [18, 17, 16, 19]. We motivate and introduce several necessary extensions of *LRP* which support 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 multiset of bindings of the form $x = e$, also called

(lbeta) $((\lambda x.s)^{\mathtt{sub}}\ r) \to (\mathtt{letrec}\ x = r\ \mathtt{in}\ s)$

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

(cp-in) $(\mathtt{letrec}\ x_1 = v^{\mathtt{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \mathtt{in}\ C[x_m^{\mathtt{vis}}]) \to (\mathtt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \mathtt{in}\ C[v])$
where v is a polymorphic abstraction

(cp-e) $(\mathtt{letrec}\ x_1 = v^{\mathtt{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, E, y = C[x_m^{\mathtt{vis}}]\ \mathtt{in}\ r)$
$\to (\mathtt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, E, y = C[v]\ \mathtt{in}\ r)$
where v is a polymorphic abstraction

(llet-in) $(\mathtt{letrec}\ E_1\ \mathtt{in}\ (\mathtt{letrec}\ E_2\ \mathtt{in}\ r)^{\mathtt{sub}}) \to (\mathtt{letrec}\ E_1, E_2\ \mathtt{in}\ r)$

(llet-e) $(\mathtt{letrec}\ E_1, x = (\mathtt{letrec}\ E_2\ \mathtt{in}\ t)^{\mathtt{sub}}\ \mathtt{in}\ r) \to (\mathtt{letrec}\ E_1, E_2, x = t\ \mathtt{in}\ r)$

(lapp) $((\mathtt{letrec}\ E\ \mathtt{in}\ t)^{\mathtt{sub}}\ s) \to (\mathtt{letrec}\ E\ \mathtt{in}\ (t\ s))$

(lcase) $(\mathtt{case}_K\ (\mathtt{letrec}\ E\ \mathtt{in}\ t)^{\mathtt{sub}}\ \mathtt{of}\ alts) \to (\mathtt{letrec}\ E\ \mathtt{in}\ (\mathtt{case}_K\ t\ \mathtt{of}\ alts))$

(lseq) $(\mathtt{seq}\ (\mathtt{letrec}\ E\ \mathtt{in}\ s)^{\mathtt{sub}}\ t) \to (\mathtt{letrec}\ E\ \mathtt{in}\ (\mathtt{seq}\ s\ t))$

(seq-c) $(\mathtt{seq}\ v^{\mathtt{sub}}\ t) \to t$ if v is a value

(seq-in) $(\mathtt{letrec}\ x_1 = (c\ \vec{s})^{\mathtt{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \mathtt{in}\ C[(\mathtt{seq}\ x_m^{\mathtt{vis}}\ t)])$
$\to (\mathtt{letrec}\ x_1 = (c\ \vec{s}), \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \mathtt{in}\ C[t])$

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

(case-c) $(\mathtt{case}_K\ c^{\mathtt{sub}}\ \mathtt{of}\ \{\ldots(c\ \mathtt{->}\ t)\ldots\}) \to t$ if $ar(c) = 0$, otherwise:
$(\mathtt{case}_K\ (c\ \vec{s})^{\mathtt{sub}}\ \mathtt{of}\ \{\ldots((c\ \vec{y})\ \mathtt{->}\ t)\ldots\}) \to (\mathtt{letrec}\ \{y_i = s_i\}_{i=1}^{ar(c)}\ \mathtt{in}\ t)$

(case-in) $(\mathtt{letrec}\ x_1 = c^{\mathtt{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \mathtt{in}\ C[(\mathtt{case}_K\ x_m^{\mathtt{vis}}\ \mathtt{of}\ \{(c\ \mathtt{->}\ r)\ldots\})])$
$\to (\mathtt{letrec}\ x_1 = c, \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \mathtt{in}\ C[r])$ if $ar(c) = 0$; otherwise:
$(\mathtt{letrec}\ x_1 = (c\ \vec{t})^{\mathtt{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \mathtt{in}\ C[(\mathtt{case}_K\ x_m^{\mathtt{vis}}\ \mathtt{of}\ \{((c\ \vec{z})\ \mathtt{->}\ r)\ldots\})])$
$\to (\mathtt{letrec}\ x_1 = (c\ \vec{y}), \{y_i = t_i\}_{i=1}^{ar(c)}, \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \mathtt{in}\ C[\mathtt{letrec}\ \{z_i = y_i\}_{i=1}^{ar(c)}\ \mathtt{in}\ r])$

(case-e) $(\mathtt{letrec}\ x_1 = c^{\mathtt{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, u = C[(\mathtt{case}_K\ x_m^{\mathtt{vis}}\ \mathtt{of}\ \{(c\ \mathtt{->}\ r_1)\ldots\})],\ E\ \mathtt{in}\ r_2)$
$\to (\mathtt{letrec}\ x_1 = c, \{x_i = x_{i-1}\}_{i=2}^{m}, u = C[r_1], E\ \mathtt{in}\ r_2)$ if $ar(c) = 0$; otherwise:
$(\mathtt{letrec}\ x_1 = (c\ \vec{t})^{\mathtt{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, u = C[(\mathtt{case}_K\ x_m^{\mathtt{vis}}\ \mathtt{of}\ \{\ldots((c\ \vec{z})\ \mathtt{->}\ r)\ldots\})], E\ \mathtt{in}\ s)$
$\to (\mathtt{letrec}\ x_1 = (c\ \vec{y}), \{y_i = t_i\}_{i=1}^{ar(c)}, \{x_i = x_{i-1}\}_{i=2}^{m}, u = C[\mathtt{letrec}\ \{z_i = y_i\}_{i=1}^{ar(c)}\ \mathtt{in}\ r], E\ \mathtt{in}\ s)$

The variables y_i are fresh ones in (case-in) and (case-e).

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

letrec-environment. We also use $\{x_{g(i)} = s_{f(i)}\}_{i=j}^{m}$ 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, $LV(E)$ to denote the binding variables of a letrec-environment, and 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 nor in a case-alternative. A *reduction context* is a context where reduction may take place, and it is defined using a labeling algorithm that indicates the call-by-need reduction positions [16]. Reduction contexts are for example $[\cdot]$, $([\cdot]\ e)$, $(\mathtt{case}\ [\cdot]\ \ldots)$ and $\mathtt{letrec}\ x = [\cdot], y = x, \ldots\ \mathtt{in}\ (x\ \mathtt{True})$. 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 $c\ \vec{s}$.

We explain the rules in Fig. 2. 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 are needed when the reduction rules have to reduce an application $(f\ a)$ 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

(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*

`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); (seq) is the union of (seq-c), (seq-in), (seq-e); (case) is the union of (case-c), (case-in), (case-e).

Definition 2.1 (Normal order reduction)**.** *A* normal order reduction step $s \xrightarrow{LRP} t$ *is performed (uniquely) if the (top-down) 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. The reduction sequence* $\xrightarrow{LRP,*}$ *is the reflexive, transitive closure,* $\xrightarrow{LRP,+}$ *is the transitive closure of* $\xrightarrow{LRP}$ *and* $\xrightarrow{LRP,k}$ *denotes k* $\xrightarrow{LRP}$*-steps.*

The labeling algorithm proceeds top-down in an expression, marks the demanded subexpressions and finally detects the reduction position. It also marks the target position for a copy operation (see (cp-e) as an example), and the indirection chains used in (case)- and (seq)-reductions. In Fig. 2 we omit the types in all rules with the exception of (Tbeta) for simplicity. Note that normal-order reduction is type safe.

Definition 2.2. *A weak head normal form (WHNF) in LRP is a value v, 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* $s \xrightarrow{LRP,*} t$ *where t is a WHNF. Expression s* diverges *(*$s{\uparrow}$*) if it does not converge.*

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 all (lbeta)-, (case)- and (seq)-reduction steps in* $s{\downarrow}s_0$*, let* $\texttt{rln}_{LCSC}(s)$ *be the sum of all a-reduction steps in* $s{\downarrow}s_0$ *with* $a \in LCSC$*, where* $LCSC = \{(lbeta), (cp), (case), (seq)\}$*, and let* $\texttt{rlnall}(s)$ *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. *We define LRPgc, which employs all the rules of LRP and (gc) (see Fig. 3) as follows: Let s be an LRP-expression. 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 maximal possible set of bindings in the top letrec-environment of s is removed.*

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

Figure 4: Definition of `size`

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 are the same for LRP and LRPgc.*

3 Definitions of 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 it 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. 4.*

The `size`-measure does not count variables, it counts letrec-bindings only by the size of the bound expressions, and it ignores the type expressions and type annotations. A justification for this omission is that this corresponds to the size (number of nodes) of the sharing graph of the whole program. A technical justification for defining $\texttt{size}(x)$ as 0 is that let-reduction rules do not change the size, and that this is compatible with the size in the machine language. For example, the bindings $x = y$ do not contribute to the size. This is justified, since the abstract machine ([5]) does not create $x = y$ bindings (not even implicit ones) and instead makes an immediate substitution.

Definition 3.2. *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* $s = s_0 \xrightarrow{LRPgc} s_1 \xrightarrow{LRPgc} \cdots \xrightarrow{LRPgc} s_n$, *and* s_n *is a WHNF. If* $s\uparrow$, *then* $spmax(s)$ *is defined as* ∞.

For a (partial) reduction sequence $Red = s_1 \to \cdots \to s_n$, *we define* $spmax(Red) = \max_i\{size(s_i) \mid s_i \to s_{i+1}$ *is not a (gc), and also* s_n *is not LRPgc-reduccible with a (gc)-reduction*$\}$

Counting space only if there is no (LRPgc,gc) possible is consistent with the definition in [7]. 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 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 such that $C[s]$, $C[t]$ are closed, $spmax(C[s]) \leq spmax(C[t])$ holds. The expression s is* space-equivalent *to t, $s \sim_{spmax} t$, iff for all contexts C such that $C[s]$, $C[t]$ are closed, $spmax(C[s]) = spmax(C[t])$ holds. 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 $\mathtt{size}(s) \leq \mathtt{size}(t)$, using $C = \lambda x.[.]$.

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 such that $R[s]$, $R[t]$ are closed, we have $spmax(R[s]) \leq spmax(R[t])$. The relation $s \sim_{R,spmax} t$ holds iff $s \leq_{R,spmax} t$ and $t \leq_{R,spmax} s$.

Lemma 3.4 (Context Lemma for Maximal Space Improvement)**.** *In LRPgc the following holds: If $\mathtt{size}(s) \leq \mathtt{size}(t)$, $FV(s) \subseteq FV(t)$, and $s \leq_{R,spmax} t$, then $s \leq_{spmax} t$.*

Proof. (Sketch [15]) The proof is by generalizing the claim to multiple pairs (s_i,t_i) of expressions in multicontexts M, i.e. by comparing $M[s_1,\ldots,s_n]$ and $M[t_1,\ldots,t_n]$, where the assumptions must hold for all pairs s_i,t_i. The induction proof is (i) on the number of *LRPgc*-reduction steps of $M[t_1,\ldots,t_n]$, and (ii) on the number of holes of M. The various cases of reductions of $M[t_1,\ldots,t_n]$ are analyzed, and in all cases the claim can be shown using the induction hypothesis.

Note that the proof technique would not work for call-by-name variants of the calculus. The reason is that substitution is incompatible with the proof technique. □

Corollary 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 $s \sim_{spmax} t$.*

The context lemmas also hold if the (stronger) condition $s \leq_{X,spmax} t$, or $s \sim_{X,spmax} t$, respectively, holds where X means surface- or top-contexts.

We also 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. Transformations that applied in reduction contexts lead to a space increase of at most a fixed (additive) constant are considered as controllable and safe, whereas the case that after the transformation the space increase may exceed any constant (depending on the usage of the expressions), is considered uncontrollable, and we say it is a space leak.

Definition 3.6. *Let T be a transformation and let $s \xrightarrow{T} t$ be an instance with expressions s,t.*

1. *We say that the $s \xrightarrow{T} t$ is* space-safe up to the constant *c, if for all reduction contexts R: $spmax(R[t]) \leq c + spmax(R[s])$.*
2. *If for some c,* (1) *holds for all instances $s \xrightarrow{T} t$, then we say T is* space-safe up to the constant *c.*
3. *The transformation $s \xrightarrow{T} t$ is a* space leak*, iff for every $b \in \mathbb{R}$, there is a reduction context R, such that $spmax(R[t]) \geq b + spmax(R[s])$.*
4. *If there is one instance $s \xrightarrow{T} t$ that is a space leak, then we also say T is a* space leak.

This (simplistic) definition is a first criterion for a classification of transformations. Definition 3.6 for a classification of transformations makes sense insofar as space-improvements are not space leaks and space leaks cannot be space improvements.

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

(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 5: Extra transformation rules

(case-cx)	$(\texttt{letrec}\ x=(c_{T,j}\ x_1 \ldots x_n), E\ \texttt{in}\ C[\texttt{case}_T\ x\ ((c_{T,j}\ y_1 \ldots y_n)\ \texttt{->}\ s)\ alts]) \to\ \texttt{letrec}\ x=(c_{T,j}\ x_1 \ldots x_n), E\ \ \texttt{in}\ C[(\texttt{letrec}\ y_1=x_1, \ldots, y_n=x_n\ \texttt{in}\ s)]$
(case-cx)	$\texttt{letrec}\ x=(c_{T,j}\ x_1 \ldots x_n), E,\ \ y=C[\texttt{case}_T\ x\ ((c_{T,j}\ y_1 \ldots y_n)\ \texttt{->}\ s)\ alts]\ \texttt{in}\ r \to\ \texttt{letrec}\ x=(c\ x_1 \ldots x_n), E,\ \ y=C[(\texttt{letrec}\ y_1=x_1, \ldots, y_n=x_n\ \texttt{in}\ s)]\ \texttt{in}\ r$
(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=)	$\texttt{letrec}\ x=y, y=s, E\ \texttt{in}\ r \to\ \texttt{letrec}\ y=s, E\ \texttt{in}\ r$ where $x \notin FV(s,E,r)$, and $y=s$ cannot be garbage collected
(caseId)	$(\texttt{case}_K\ s\ (pat_1\ \texttt{->}\ pat_1) \ldots (pat_{\lvert D_K \rvert}\ \texttt{->}\ pat_{\lvert D_K \rvert})) \to s$

Figure 6: Variations of transformation rules (space improvements)

4 Space-Safe and Unsafe Transformations

More transformations are defined in Fig. 5: (cpx) is the union of (cpx-in) and (cpx-e) and copies variables, (cpcx) is the union of (cpcx-in) and (cpcx-e) and copies constructor applications with variable-only-arguments, (abs), (abse) abstracts subexpressions by putting them in a binding environment, and (ucp) is the union of (ucp1), (ucp2), and (ucp3) and is a (cp) into a unique occurrence of x, followed by a garbage collection. Further transformations are defined and mentioned in Fig. 6 and 7: (case-cx) 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[1]; (cpcxT) is a variant of (cpcx), where the target context is a T-context; (caseId) is a typed transformation that detects case-expressions 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 T-context, and similar for others.

[1] S,T-contexts are defined in Section 2.1

(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`.

Figure 7: Some special transformation rules (space-worsening)

4.1 On the Space-Safety of Transformations

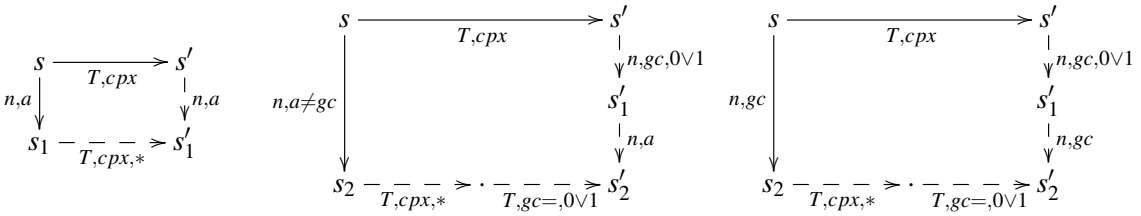
An overview of the results for max-space-improvements, -equivalences and space-worsening transformations are in the following theorem where further transformations are in Figs. 3, 5, 6 and 7. The proof technique for most of the proofs consists of computing complete sets of forking diagrams between transformation steps and the normal-order reduction steps and an appropriate induction proof on the length of reduction sequences (see [20] for more explanations), where computation of diagrams is simplified thanks to the context lemma (see [15] for details).

Theorem 4.1. *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=)*
$\not\preceq_{spmax}$	*(cpcx), (cpS)*
space-safe up to 1	*(T,(cpcxT))*
space-safe up to $\texttt{size}(v)$	*(S,(cpS))*
	where v is the copied abstraction
space-leak	*(cp), (cse), (soec)*

Proof. Complete Proofs for the space-safety can be found in [15], and sketches and remarks in the remainder of this section. As an example, we treat (cpx) in more detail:
Claim. The transformation (cpx) is a space-equivalence.
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 set of three diagrams is 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 3.5), 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. □

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

We investigate the space-properties of (cp): Used as transformation (cpS) in an *S*-context it increases max-space at most by $\texttt{size}(v)$ where v is the copied abstraction; and in general the size-increase can be bounded by $(\texttt{rln}(s)+2)*\texttt{size}(v)$ where s is the initial expression. 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 4.2. *The following estimations hold for (cp) and (cpS), where* $s \xrightarrow{cp} t$*, and where* v *is the copied abstraction:*

1. *The transformation* $s \xrightarrow{(S,cpS)} t$ *increases max-space at most by* $\texttt{size}(v)$.
2. *The transformation* $s \xrightarrow{cp} t$ *increases max-space at most by* $(\texttt{rln}(s)+2)*\texttt{size}(v)$*, i.e.* $spmax(t) \leq (\texttt{rln}(s)+2)*\texttt{size}(v) + spmax(s)$.

A consequence is that the space usage of several transformations $\xrightarrow{(S,cpS)}$ that are space-safe up to the additive constant c can be estimated:

Corollary 4.3. *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 *transformations* $\xrightarrow{(S,cpS)}$*, 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 4.2 and since $\xrightarrow{(S,cpS)}$ does not increase the size of abstractions. □

Remark 4.4. *Using (cp) as transformation with general contexts for the target, for example copying into an abstraction, may induce a space-leak, but see Proposition 4.2. More exactly, the max-space of a reduction sequence may increase linearly with the number of reduction steps, and exponentially with the number of applications of the (cp)-transformation. Examples of this behavior can be constructed as in Example 4.5.*

Note that there are instances of (cp) that behave much better, for example versions of inlining (see below in Section 5), or if the copied abstraction can be garbage collected after (cp) or transformed further, and also the special case of (ucp)-transformations.

4.2 Specific Examples and Comparison with Previous Work

Now we explain several examples and compare with related work.

Example 4.5. *We show that common subexpression elimination is a space leak. 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:* $s := \texttt{if } (\texttt{last } [1..n]) > 0 \texttt{ then } [1..n] \texttt{ else Nil}$*, where* $[1..n]$ *is the expression that lazily generates a list* $[1,\ldots,n]$*. The evaluation of* s *expands the list until the last element is generated and then evaluates the same expression to obtain* $1 : [2..n]$*. Due to eager garbage*

collection, it is not hard to see that the evaluation sequence requires constant max-space, independent of n (assuming constant space for integers). Note that this evaluation will also generate indirection chains of the form $\ldots, x_1 = x_2, x_2 = x_3, \ldots$, which are ignored by our space measure. As shown in [5] an evaluation on an abstract machine will really use constant space, if shortening indirections is performed by the abstract machine.

Now let $s' = \texttt{letrec}\ x = [1..n]\ \texttt{in if}\ (\texttt{last}\ x) > 0\ \texttt{then}\ x\ \texttt{else Nil}$. 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 analyzes for future research.

The example and arguments in [2] show 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.

Example 4.6. *An example that illustrates the definitions and may contribute to the discussion on the boundaries between space-safe and -unsafe transformation is the following: Let $s = \texttt{True}$ and $t = (id\ \texttt{True})$. Then clearly $s \leq_{spmax} t$, and the transformation $s \to t$ is space-safe. Let $s' = \lambda x.s$ and $t' = \lambda x.t$. Then $s' \to t'$ is a space leak according to our definition:*
Let $R = (\texttt{letrec}\ y = [\cdot], z = r_n\ \texttt{in}\ (\texttt{and}\ z)\ \&\&\ (\texttt{last}\ z))$, where r_n is the list $[(y\ 0), \ldots, (y\ 0)]$ of length n, `and` *is the function that computes the logical conjunction of all list entries, and && is the logical conjunction. Then the difference $spmax(R[t]) - spmax(R[s])$ is a linear function in n that exceeds all bounds, hence $s' \to t'$ is a space leak.*

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 results that are easier to obtain and to grasp 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, which exactly enforce the evaluation of the spine of the lists like $(\texttt{last}\ [\cdot])$, the *spmax*-difference is exactly 4. However, for example in a reduction context $(\texttt{seq}\ (\texttt{last}\ [\cdot])\ s)$, where the evaluation of s requires (much) more space than the expression $(\texttt{last}\ [s])$, there is no max-space difference, since we analyze the maximally used space. 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 [7], 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 $\texttt{sum}'$ with a non-strictly used accumulator and the tail-recursive $\texttt{sum}''$ with a strictly used accumulator for the result.

`sum` requires space linear in the length of the list, and the same holds for $\texttt{sum}'$. However, `sum`, $\texttt{sum}'$ and $\texttt{sum}''$ 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

Without inlining:

$\texttt{foldl} \quad = \lambda f,z,xs.\texttt{case}\ xs\ \texttt{of}\ \{([]\ \texttt{->}\ z)\ ((y:ys)\ \texttt{->}\ \texttt{foldl}\ f\ (f\ z\ y)\ ys)\}$

$\texttt{foldl}' \quad = \lambda f,z,xs.\texttt{case}\ xs\ \texttt{of}\ \{([]\ \texttt{->}\ z)\ ((y:ys)\ \texttt{->}\ \texttt{let}\ w=(f\ z\ y)\ \texttt{in}\ \texttt{seq}\ w\ (\texttt{foldl}'\ f\ w\ ys))\}$

$\texttt{foldr} \quad = \lambda f,z,xs.\texttt{case}\ xs\ \texttt{of}\ \{([]\ \texttt{->}\ z)\ ((y:ys)\ \texttt{->}\ f\ y\ (\texttt{foldr}\ f\ z\ ys))\}$

With inlining using `xor` as function:

$\texttt{xor} \quad = \lambda x,y.\texttt{case}\ x\ \texttt{of}\ \{(\texttt{True}\ \texttt{->}\ \texttt{case}\ y\ \texttt{of}\ \{(\texttt{True}\ \texttt{->}\ \texttt{False})\ (\texttt{False}\ \texttt{->}\ \texttt{True})\})\ (\texttt{False}\ \texttt{->}\ y)\}$

$\texttt{foldl} \quad = \lambda f,z,xs.\texttt{case}\ xs\ \texttt{of}\ \{([]\ \texttt{->}\ z)$
$((y:ys)\ \texttt{->}\ \texttt{foldl}\ f\ (\texttt{case}\ z\ \texttt{of}\ \{(\texttt{True}\ \texttt{->}\ \texttt{case}\ y\ \texttt{of}\ \{(\texttt{True}\ \texttt{->}\ \texttt{False})$
$(\texttt{False}\ \texttt{->}\ \texttt{True})\})$
$(\texttt{False}\ \texttt{->}\ y)\})\ ys)\}$

The inlining of `foldl'` and `foldr` is analogous to the inlining of `foldl`.

Figure 8: Definitions of `foldl`, `foldl'` and `foldr`

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 [6]: As a further comparison we check and compare our results (see Proposition 4.2) with those for weak improvement in Fig. 2 in [6]: the claim on (weak-value-beta) there appears to be practically almost useless, (at least for a special case): copying once indeed can only increase the space by a linear function in the size of the program (and as parameter the number of reductions in our formulation (see Prop. 4.2)), 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 [6] permits $\texttt{letrec}\ x=V[x]\ \texttt{in}\ C[x] \to \texttt{letrec}\ x=V[V[x]]\ \texttt{in}\ C[V[x]] \to \texttt{letrec}\ x=V[V[V[x]]]\ \texttt{in}\ C[V[V[x]]]$, 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 the two last let-shuffling rules of [6], which are (strong) space-improvements w.r.t. our measure and definitions, since we have proved that (lll) is a space equivalence.

Typed Transformations The rule (caseId) is also the heart of other type-dependent transformations, which are also only correct under typing, and is a space-improvement. Examples of more general transformations of a similar kind 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 $\psi(s\ t)$ to $\texttt{letrec}\ y=\psi(t)\ \texttt{in}\ (\psi(s)\ y)$, which is the same as an inverse (ucp). Our results, in particular the results on (ucp) (Thm. 4.1), show that the complete translation ψ is a space-equivalence. Note that the abstract machine uses extra data structures for evaluation.

5 Experimental Analysis of Inlining and `fold` Using the Tool LRPi

We use our interpreter LRPi that executes *LRPgc*-programs using an abstract machine approach and measures the runtime and space behavior (for more details see [5]) and apply it to several fold-variants.

k	100	200	300	400	500	600	700	800	900	1000
	`foldl False xor (take k lst)`									
`rln`	1214	2414	3614	4814	6014	7214	8414	9614	10814	12014
spmax	825	1625	2425	3225	4025	4825	5625	6425	7225	8025
	after inlining:									
`rln`	1012	2012	3012	4012	5012	6012	7012	8012	9012	10012
spmax	882	1682	2482	3282	4082	4882	5682	6482	7282	8082
	`foldl'` `False xor (take k lst)`									
`rln`	1315	2615	3915	5215	6515	7815	9115	10415	11715	13015
spmax	63	63	63	63	63	63	63	63	63	63
	after inlining:									
`rln`	1113	2213	3313	4413	5513	6613	7713	8813	9913	11013
spmax	75	75	75	75	75	75	75	75	75	75
	`foldr False xor (take k lst)`									
`rln`	1115	2215	3315	4415	5515	6615	7715	8815	9915	11015
spmax	66	66	66	66	66	66	66	66	66	66
	after inlining:									
`rln`	913	1813	2713	3613	4513	5413	6313	7213	8113	9013
spmax	84	84	84	84	84	84	84	84	84	84

Figure 9: Experimental Results

The correctness of the space measurement of the abstract machine is described in [18] (see Thm. 4.1).

We analyze the space behavior of `fold` using exclusive-or as function, `False` as neutral element and a list `lst` starting with a single `True` followed by $k-1$ `False`-elements generated using a take-function/list generator approach. We already compared the three fold variants concerning runtime and space consumption with each other in this scenario in [5], but now we focus on the impact of inlining, i.e. how does inlining affect the space consumption in the same scenario?

The current version of LRPi uses Peano-encoding for positive integers, but treats arbitrary Peano numbers as of size 1, which makes it is easier to analyze the results. Hence the current statistics differs from that in [5]. The `fold`-functions are defined in Fig. 8. Inlining copies the defining lambda-expression for `xor` to a call site and then applies (lbeta), (ucp), (cpx), (gc) perhaps several times to obtain the inlined definitions in Fig. 8. In order to keep the experiment simple and interpretable, we omit further obvious optimizations. Since inlining copies into an abstraction (in addition into a recursive definition), our theoretical results do not give good guarantees on the space behavior and also do not preclude that the transformation might be a space leak. Here further research is needed.

Fig. 9 uses Haskell-notation where k is the length of the input list, `rln` is the runtime measure, i.e. the number of (essential) reduction steps (see Def. 2.4), *spmax* is our space measure (see Def. 3.2).

For `foldl` the runtime decreases linearly after inlining, since this decreases the number of reduction steps by a constant for each list element. In contrast the inlining increases the space consumption by a constant. Space consumption is linear in the length of the input list, which is caused by the left-associativity of `foldl` since we get a linear number of nested `case`-expressions caused by the `xor`. The constant increase of space consumption after inlining is caused by the constant additional space that is needed by the inlined `xor`-function.

If we use the strict variant of `foldl` (i.e. `foldl'`), then the accumulator is evaluated each time and therefore no nested `case`-expressions are constructed. As expected we see that the space consumption is

constant and that the runtime again improves linearly.

For `foldr` the inlining improves the runtime linearly. Moreover the space consumption also only increases by a constant (similar to the `foldl'`-variant). The reason is the right-associativity of `foldr`: since `xor` is strict in the first argument, `foldr` runs over the whole list, but depending on the left argument, `xor` either evaluates the second argument or returns the argument. Since the list is lazily generated and contains only `False`-elements (up to one occurrence), each element gets directly generated and consumed and therefore only constant space is needed.

The example suggests that it is a good idea to invest (a bit of) space for time, since for `foldl'` and `foldr` the runtime improves linearly while the space consumption only increases by a constant. Our experiment shows a nice behavior in the considered empty context, but does not show the behavior in other contexts, or other uses of the functions.

6 Conclusion and Future Research

We successfully derived results on the space behavior of transformations in lazy functional languages, by defining a relaxed space measure and reasoning about space improvements and space equivalences. We developed and justified a criterion for classifying transformations as space safe or space leaks. An impact of our results could be a controlled runtime optimization during compile time by applying time-improving transformations, but taking into account the knowledge of their impact on the max-space usage. We contributed by detailing and refining this knowledge for call-by-need functional languages.

Future work is to extend the analysis of transformations to larger and more complex transformations in the polymorphic typed setting. A generalisation of top contexts to surface contexts is also of value for several transformations. To develop methods and justifications for space-improvements involving recursive definitions is left for future work, as well as the exploration of the use of space-ticks as in [7] to improve the computation of estimations.

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, doi:10.1145/199448.199507.

[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, doi:10.1017/CBO9781316092415.

[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.

[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, doi:10.1145/2628136.2628142.

[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, doi:10.1145/292540.292547.

[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, doi:10.1007/3-540-48919-3_8.

[13] Simon Peyton Jones & Simon Marlow (2002): *Secrets of the Glasgow Haskell Compiler inliner*. *Journal of Functional Programming* 12(4+5), pp. 393–434, doi:10.1017/S0956796802004331.

[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: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.cs.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, doi:10.4230/OASIcs.WPTE.2014.63.

[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:10.1145/2790449.27905125.

[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.cs.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. `http://www.ki.cs.uni-frankfurt.de/papers/frank/`.

[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, doi:10.1007/978-3-642-41582-1_6.

[22] Peter Sestoft (1997): *Deriving a Lazy Abstract Machine*. *J. Funct. Program.* 7(3), pp. 231–264, doi:10.1017/S0956796897002712.