An Environment for Analyzing Space Optimizations in Call-by-Need Functional Languages

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

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

We present an implementation of an interpreter LRPi for the call-by-need calculus LRP, based on a variant of Sestoft's abstract machine Mark 1, extended with an eager garbage collector. It is used as a tool for exact space usage analyses as a support for our investigations into space improvements of call-by-need calculi.

1 Introduction

Lazy functional languages like Haskell use call-by-need as evaluation strategy. This leads to a more declarative way of programming where a specification of the result is emphasized instead of specifying the sequence of evaluations. This approach allows a lot of correct program transformations that can potentially be used by a compiler for optimization purposes. It would be a very helpful information to know, whether a program transformation decreases time/space usage or in which situations this may occur. We will capture this using the notion of improvements. In this paper we emphasize space improvements, pursuing our long-term research goal to analyze time- and space-improvements for Haskell-like languages. The goal of this paper is to put forward further studies on improvements, with a main focus on providing a test environment to support and speed up the analysis of improvements.

Previous work on improvements w.r.t. time usage (the number of reduction steps), is e.g. [9, 10, 11] for call-by-name and [8, 15, 13, 14] for call-by-need. There seem to be only a few studies on space improvements, by Gustavsson and Sands [4, 5, 3]. Their notion of (strong) space improvement is mainly the same as ours, however, they use an untyped (restricted) language. We will investigate a typed language since typing enables more transformations to be improvements, for example `map id xs` is equivalent to `xs` under typing, but not in untyped calculi. The reason is that also contexts must be typed and thus only tests that are (type-)compatible with the intention of the program are used for characterizing improvements.

We will use the lazy typed functional core language LRP [12] for defining and analyzing space improvements. LRP has a rich syntax including `letrec`, data constructors and `case`-expressions, Haskell's `seq`-operator, and polymorphic typing, modeling Haskell's core language. Evaluation in LRP is defined by a rewriting semantics. An improvement w.r.t. a measure is a locally applicable (and correct) transformation that transforms an expression e_1 to e_2, such that e_2 is at least as good as e_1 w.r.t. the chosen measure in all contexts. Our correctness notion is contextual equivalence, which means that e_1, e_2 behave identically w.r.t. termination in all contexts.

Our approach is to use an abstract Sestoft-machine (see also [4]) as interpreter in order to have a realistic model for the resource consumption at runtime. Since space is an issue, in particular the maximally used space during an evaluation, a detection of dynamically generated garbage is required, which leads to the implementation of an (eager) garbage collector. This is nontrivial, since letrec permits cyclic

H. Cirstea, S. Escobar (Eds.): Third International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE'16).
EPTCS 235, 2017, pp. 78–92, doi:10.4204/EPTCS.235.6

references. Furthermore, indirections $x = y$ in `letrec`-environments turned out to lead to more space consumption in the Sestoft machine than in the calculus, which is defeated by removing indirections at compile time as well as adapting the abstract machine. Removing indirections can be done efficiently (see Section 5.1.1). The interpreter implementation is also shown to exactly count the maximal space usage for machine expressions (see Theorem 5.3). Our specific space analyses can exhibit in examples the reason for unexpected space increases, can refute transformations being space improvements, and can also give hints on the complexity of evaluations.

Outline In Section 2 LRP is introduced. In Section 3 LRP is extended with garbage collection. Space improvements are explained in Section 4. In Section 5 the LRP-interpreter is described. The analyses and some results are in Section 6. We conclude in Section 7.

2 Polymorphically Typed Lazy Lambda Calculus

We recall the Polymorphically Typed Lazy Lambda Calculus (LRP) [13] as language. We also motivate and introduce several necessary extensions for supporting realistic space analyses.

LRP [12] is LR (e.g. see [16]) extended with types. I.e., LRP is an extension of the lambda calculus by polymorphic types, recursive `letrec`-expressions, `case`-expressions, `seq`-expressions, data constructors, type abstractions $\Lambda a.s$ to express polymorphic functions and type applications $(s\ \tau)$ for type instantiations. The syntax of expressions and types of LRP is defined in Fig. 1.

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:

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

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

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

Figure 1: Syntax of expressions and types

An expression is *well-typed* if it can be typed using typing rules that are defined in [12]. LRP is a core language of Haskell and is simplified compared to Haskell, because it does not have type classes and is only polymorphic in the bindings of `letrec` variables. But LRP is strong enough to express polymorphically typed lists, and functions working on such data structures.

From now on we use Env as abbreviation for a `letrec`-environment, $\{x_{g(i)} = s_{f(i)}\}_{i=j}^{m}$ for $x_{g(j)} = s_{f(j)}, \ldots, x_{g(m)} = s_{f(m)}$ and $alts$ for `case`-alternatives. We use $FV(s)$ and $BV(s)$ to denote free and bound variables of an expression s and $LV(Env)$ to denote the binding variables of a `letrec`-environment. Furthermore we abbreviate $(c_{K,i}\ s_1\ \ldots\ s_{ar(c_{K,i})})$ with $c\ \overrightarrow{s}$ 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 hole $[\cdot]$ at expression position. A *value* is an abstraction $\lambda x.s$, a type abstraction u or a constructor application $c\ \overrightarrow{s}$.

After the reduction position is determined using the labeling algorithm of [12], a unique reduction rule of Fig. 2 is applied at this position which constitutes a normal-order reduction step.

(lbeta)	$C[((\lambda x.s)^{\mathrm{sub}}\ r)] \to C[(\mathtt{letrec}\ x = r\ \mathtt{in}\ s)]$
(Tbeta)	$((\Lambda a.u)^{\mathrm{sub}}\ \tau) \to u[\tau/a]$
(cp-in)	$(\mathtt{letrec}\ x_1 = v^{\mathrm{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ C[x_m^{\mathrm{vis}}])$
	$\to (\mathtt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ C[v])$
	where v is a polymorphic abstraction
(cp-e)	$(\mathtt{letrec}\ x_1 = v^{\mathrm{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, Env, y = C[x_m^{\mathrm{vis}}]\ \mathtt{in}\ r)$
	$\to (\mathtt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^m, Env, y = C[v]\ \mathtt{in}\ r)$
	where v is a polymorphic abstraction
(llet-in)	$(\mathtt{letrec}\ Env_1\ \mathtt{in}\ (\mathtt{letrec}\ Env_2\ \mathtt{in}\ r)^{\mathrm{sub}}) \to (\mathtt{letrec}\ Env_1, Env_2\ \mathtt{in}\ r)$
(llet-e)	$(\mathtt{letrec}\ Env_1, x = (\mathtt{letrec}\ Env_2\ \mathtt{in}\ t)^{\mathrm{sub}}\ \mathtt{in}\ r) \to (\mathtt{letrec}\ Env_1, Env_2, x = t\ \mathtt{in}\ r)$
(lapp)	$C[((\mathtt{letrec}\ Env\ \mathtt{in}\ t)^{\mathrm{sub}}\ s)] \to C[(\mathtt{letrec}\ Env\ \mathtt{in}\ (t\ s))]$
(lcase)	$C[(\mathtt{case}_K\ (\mathtt{letrec}\ Env\ \mathtt{in}\ t)^{\mathrm{sub}}\ \mathtt{of}\ alts)] \to C[(\mathtt{letrec}\ Env\ \mathtt{in}\ (\mathtt{case}_K\ t\ \mathtt{of}\ alts))]$
(seq-c)	$C[(\mathtt{seq}\ v^{\mathrm{sub}}\ t)] \to C[t]$ if v is a value
(seq-in)	$(\mathtt{letrec}\ x_1 = (c\ \overrightarrow{s})^{\mathrm{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ C[(\mathtt{seq}\ x_m^{\mathrm{vis}}\ t)])$
	$\to (\mathtt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ C[t])$ if v is a value
(seq-e)	$(\mathtt{letrec}\ x_1 = (c\ \overrightarrow{s})^{\mathrm{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, Env, y = C[(\mathtt{seq}\ x_m^{\mathrm{vis}}\ t)]\ \mathtt{in}\ r)$
	$\to (\mathtt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^m, Env, y = C[t]\ \mathtt{in}\ r)$ if v is a value
(lseq)	$C[(\mathtt{seq}\ (\mathtt{letrec}\ Env\ \mathtt{in}\ s)^{\mathrm{sub}}\ t)] \to C[(\mathtt{letrec}\ Env\ \mathtt{in}\ (\mathtt{seq}\ s\ t))]$
(case-c)	$C[(\mathtt{case}_K\ c^{\mathrm{sub}}\ \mathtt{of}\ \{\ldots(c \to t)\ldots\})] \to C[t]$ if $ar(c) = 0$, otherwise:
	$C[(\mathtt{case}_K\ (c\ \overrightarrow{x})^{\mathrm{sub}}\ \mathtt{of}\ \{\ldots((c\ \overrightarrow{y}) \to t)\ldots\})] \to C[(\mathtt{letrec}\ \{y_i = x_i\}_{i=1}^{ar(c)}\ \mathtt{in}\ t)]$
(case-in)	$(\mathtt{letrec}\ x_1 = c^{\mathrm{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ C[(\mathtt{case}_K\ x_m^{\mathrm{vis}}\ \mathtt{of}\ \{\ldots(c \to r)\ldots\})])$
	$\to (\mathtt{letrec}\ x_1 = c, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ C[r])$ if $ar(c) = 0$; otherwise:
	$(\mathtt{letrec}\ x_1 = (c\ \overrightarrow{t})^{\mathrm{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ C[(\mathtt{case}_K\ x_m^{\mathrm{vis}}\ \mathtt{of}\ \{\ldots((c\ \overrightarrow{z}) \to r)\ldots\})])$
	$\to (\mathtt{letrec}\ x_1 = (c\ \overrightarrow{y}), \{y_i = t_i\}_{i=1}^{ar(c)}, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ C[\mathtt{letrec}\ \{z_i = y_i\}_{i=1}^{ar(c)}\ \mathtt{in}\ r])$
(case-e)	$(\mathtt{letrec}\ x_1 = c^{\mathrm{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, u = C[(\mathtt{case}_K\ x_m^{\mathrm{vis}}\ \mathtt{of}\ \{\ldots(c \to r_1)\ldots\})],\ Env\ \mathtt{in}\ r_2)$
	$\to (\mathtt{letrec}\ x_1 = c, \{x_i = x_{i-1}\}_{i=2}^m, u = C[r_1], Env\ \mathtt{in}\ r_2)$ if $ar(c) = 0$; otherwise:
	$(\mathtt{letrec}\ x_1 = (c\ \overrightarrow{t})^{\mathrm{sub}}, \{x_i = x_{i-1}\}_{i=2}^m,$
	$u = C[(\mathtt{case}_K\ x_m^{\mathrm{vis}}\ \mathtt{of}\ \{\ldots((c\ \overrightarrow{z}) \to r)\ldots\})], Env\ \mathtt{in}\ s)$
	$\to (\mathtt{letrec}\ x_1 = (c\ \overrightarrow{y}), \{y_i = t_i\}_{i=1}^{ar(c)}, \{x_i = x_{i-1}\}_{i=2}^m,$
	$u = C[\mathtt{letrec}\ \{z_i = y_i\}_{i=1}^{ar(c)}\ \mathtt{in}\ r], Env\ \mathtt{in}\ s)$

Figure 2: Basic reduction rules. The variables y_i are fresh.

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 notions for 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 [12] terminates on s, inserting sub (subexpression) and vis (visited by the labeling), and the applicable rule of Fig. 2 produces t. The notation* $\xrightarrow{LRP,*}$ *is the reflexive, transitive closure,* $\xrightarrow{LRP,+}$ *is the transitive closure of* $s \xrightarrow{LRP} t$*; and* $\xrightarrow{LRP,k}$ *denotes k normal order steps.*

Definition 2.2.

1. *A weak head normal form (WHNF) is a value, or an expression* $\texttt{letrec}\ Env\ \texttt{in}\ v$*, where v is a value, or an expression* $\texttt{letrec}\ x_1 = c\ \overrightarrow{t}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ x_m$.
2. *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.*
3. *The symbol* $\bot$ *represents a closed diverging expression, e.g.* $\texttt{letrec}\ x = x\ \texttt{in}\ x$.

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

The following notion of reduction length is used for measuring the time behavior in LRP.

Definition 2.4. *For a closed LRP-expression s with* $s{\downarrow}s_0$*, let* $rln(s)$ *be the sum of all (lbeta)-, (case)- and (seq)-reduction steps in* $s{\downarrow}s_0$*, and let* $rlnall(s)$ *be the number of all reductions, but not (TBeta), in* $s{\downarrow}s_0$*.*

3 LRP with Eager Garbage Collection

The calculus LRP does not remove garbage itself. However, for measuring the space-behavior, garbage should be ignored (and removed). Thus in this section we add reduction rules for removing garbage, and show that an evaluation strategy with garbage collection does not change the semantics of the calculus. In Fig. 3 the rules for garbage collection are defined. We use (gc) for the union of (gc1) and (gc2).

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

Figure 3: Garbage collection rules

Since we focus on space improvements, it is useful to model eager garbage collection also in the calculus, which leads to the calculus LRPgc. It collects (dynamic) garbage only in the top `letrec`, which is sufficient to remove all (reference-) garbage, if the starting program does not contain garbage.

Definition 3.1. *LRPgc is LRP where the normal-order reduction is modified as follows:*
Let s be an LRP-expression. A normal-order-gc (nogc) reduction step *is defined by two cases:*

1. *If a (gc)-transformation is applicable to s in the top* `letrec`*, then this transformation is applied to s, where the maximal number of bindings is removed.*
2. *If 1. is not applicable and an LRP-normal-order reduction step is applicable to s, then this normal-order reduction is applied to s.*

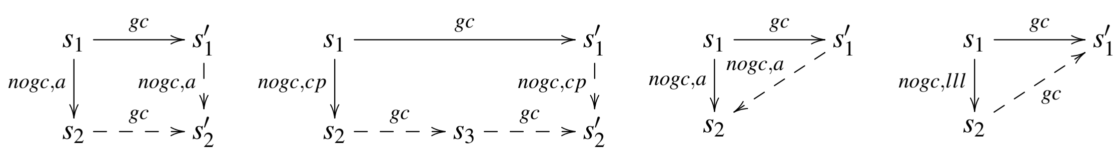
A sequence of nogc-reduction steps is called an nogc-reduction sequence*. An LRPgc-WHNF s is either an LRP-WHNF which is not a* `letrec` *expression, or it is an LRP-WHNF that is a* `letrec`*-expression which does not permit (gc)-transformation in the top* `letrec`*. If for s, there is an nogc-reduction sequence that leads to an LRPgc-WHNF, then we say s* converges w.r.t. LRPgc *and write* $s\downarrow_{nogc}$*.*
In LRPgc, the equivalence $s \sim_{c,nogc} t$ *is defined as for LRP, but w.r.t.* $\downarrow_{nogc}$*.*

Several subsequent (gc)-reductions are possible in an *LRPgc*-normal-order reduction sequence, for example a (gc2)-reduction followed by a (gc1)-reduction.

We will show in the following that the calculi *LRP* and *LRPgc* are equivalent w.r.t. convergences as well as w.r.t. the rln-measure.

In the following we will use complete sets of forking (and commuting) diagrams (more information on this technique is in [16]). A forking is an overlapping between a normal-order transformation and a non-normal-order transformation (also called internal transformation). A complete set of forking diagrams for transformation b contains a forking diagram for each possible forking of the form $s_2 \xleftarrow{nogc} s_1 \xrightarrow{b} s_1'$. The treatment is similar for commuting diagrams and commuting situations $s_1 \xrightarrow{b} s_1' \xrightarrow{nogc} s_2'$. We will use the notation $(nogc,a)$ which is an arbitrary nogc-reduction if not otherwise stated. If the label a is used twice, then all occurrences of a represent the same rule. Let LCSC := $\{$(lbeta), (case), (seq), (cp)$\}$.

Lemma 3.2. *The forking diagrams between a nogc-reduction and a non-normal-order (gc)-transformation in LRPgc in any context are the following:*



The commuting diagrams can be immediately derived from the forking diagrams.

Proof. The first diagram occurs if the nogc-reduction and the transformation can be commuted. The second diagrams happens if the gc-transformation was done in the copied abstraction. The third diagram occurs, if the effect of the gc-transformation was also done by the nogc-reduction, where we assume that $\xrightarrow{gc}$ and $\xrightarrow{nogc,a}$ are different. Finally the fourth diagram occurs for example in $(\texttt{letrec}\ Env_1\ \texttt{in}\ (s_1\ s_2)) \xleftarrow{nogc} ((\texttt{letrec}\ Env_1\ \texttt{in}\ s_1)\ s_2) \xrightarrow{gc} (s_1\ s_2)$ and where $(\texttt{letrec}\ Env_1\ \texttt{in}\ (s_1\ s_2)) \xrightarrow{gc} (s_1\ s_2)$. □

Theorem 3.3. *LRP and LRPgc are convergence-equivalent, i.e. for all expressions s:* $s\downarrow \iff s\downarrow_{nogc}$*.*

Proof. If $s\downarrow_{nogc}$ then $s\downarrow$ holds, since (gc) and all reductions of the calculus are correct w.r.t. LRP-normal-order reduction, which follows from their untyped correctness (see [16]).
Under the assumption that (gc) is correct in LRPgc, it is straightforward to show that $s\downarrow$ implies $s\downarrow_{nogc}$.
It remains to show that (gc) is correct in LRPgc: Therefore we have to use the diagrams in Lemma 3.2 for (gc). We consider the situation $s_0 \xleftarrow{nogc,*} s_1 \xrightarrow{gc} s_1'$ where s_0 is an *LRPgc*-WHNF. For the induction proof we consider the smaller diagram $s_2 \xleftarrow{nogc} s_1 \xrightarrow{gc} s_1'$ and show that there is a nogc-reduction of s_1' such that $\text{rlnall}(s_1') \leq \text{rlnall}(s_1)$. First we observe that *LRPgc*-WHNFs remain *LRPgc*-WHNFs under (gc).
The induction measure is $\text{rlnall}(s_1)$. For the situation $s_2 = s_1'$ or if any of the four diagrams applies to the

situation, the induction hypothesis applies, where in case of diagrams 2, we have to apply it twice. This shows that there is a nogc-reduction of s_1 to a *LRPgc*-WHNF.

The second part is to consider the situation $s_1 \xrightarrow{gc} s_1' \xrightarrow{nogc,*} s_0'$, where s_0' is an *LRPgc*-WHNF. Here we show more: that there is an nogc-reduction of s_1 with $\mathrm{rln}_{LCSC}(s_1) \leq \mathrm{rln}_{LCSC}(s_1')$, where rln_{LCSC} counts the normal-order reductions from LCSC until a WHNF is reached. The induction is on the lexicographic combination of the measures $(\mathrm{rln}_{LCSC}(s_1'), \mu_{lll}(s_1), |s_1'|, \mathrm{rlnall}(s_1'))$, where μ_{lll} is the measure from [16] that is strictly decreased by every $\xrightarrow{lll}$ and $\xrightarrow{gc}$-reduction, and $|s_1'|$ is the size of s_1' as an expression. If s_1' is an LRP-WHNF, then either $s_1 \xrightarrow{gc} s_1'$ is a normal-order reduction, and we are done, or it is not a normal-order reduction, and s_1 is also an LRP-WHNF.

If $s_1 \xrightarrow{gc} s_1'$ is an nogc-reduction, then the claim holds. In the case of the first diagram, the induction hypothesis can be applied by the following reasoning: if the s_1'-reduction is a LCSC-reduction, then the measure is decreased; if it is an (lll) or (gc), then the first component is the same but pair of the second and third component is strictly smaller. In the case of the second diagram, $\mathrm{rln}_{LCSC}(s_2')$ is strictly smaller, and hence also, by the induction hypothesis, $\mathrm{rln}_{LCSC}(s_3)$ and we can again apply the induction hypothesis. In the case of the third diagram, reasoning is obvious. Finally, in the case of the fourth diagram, $\mu_{lll}(s_2) < \mu_{lll}(s_1')$, hence the induction hypothesis can be applied. □

Corollary 3.4. *The contextual equivalences of LRP and LRPgc are identical.*

The proof of Theorem 3.3 also shows that the rln-measure of expressions is the same for *LRP* and *LRPgc*. Hence we can drop the distinction between LRP and LRPgc w.r.t. rln as well as for $\sim_c$.

4 Time- and Space-Improvements

For space analyses, we first define the size of expressions:

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

$\mathtt{size}(x)$	$= 0$	$\mathtt{size}(s\ t)$	$= 1 + \mathtt{size}(s) + \mathtt{size}(t)$
$\mathtt{size}(c\,\overrightarrow{s})$	$= 1 + \sum_{i=1}^{n} \mathtt{size}(s_i)$	$\mathtt{size}(\mathtt{seq}\ s_1\ s_2)$	$= 1 + \mathtt{size}(s_1) + \mathtt{size}(s_2)$
$\mathtt{size}(\lambda x.s)$	$= 1 + \mathtt{size}(s)$	$\mathtt{size}(\mathtt{letrec}\ \{x_i = s_i\}_{i=1}^{n}\ \mathtt{in}\ s)$	$= \mathtt{size}(s) + \sum_{i=1}^{n} \mathtt{size}(s_i)$
$\mathtt{size}(c\,\overrightarrow{x}\ \text{->}e)$	$= 1 + \mathtt{size}(e)$	$\mathtt{size}(\mathtt{case}\ e\ \mathtt{of}\ \{alt_1\ \dots\ alt_n\})$	$= 1 + \mathtt{size}(e) + \sum_{i=1}^{n} \mathtt{size}(alt_i)$

Type annotations are not counted by the size measure and thus they are also not shown in the definition of `size`. Note that our chosen size measure also does not count variables, the number of `letrec`-bindings, nor the `letrec`-label itself. This can be justified, since these constructs are usually represented more efficiently (or do not occur) in realistic implementations, for example in the abstract machine.

For measuring the space-behavior of s, we use the maximum size occurring in an *nogc*-reduction sequence to a WHNF:

Definition 4.2. *Let s be a closed LRP-expression. If* $s = s_0 \xrightarrow{nogc} s_1 \xrightarrow{nogc} \dots \xrightarrow{nogc} s_n$ *where* s_n *is a WHNF, then spmax(s) is the maximum of* $\mathtt{size}(s_i)$. *If* $s\!\uparrow$ *then* $spmax(s) = \infty$.

This measure is very strict and especially appropriate if the available space is limited. A transformation is a time improvement [14, 13] if it never increases the rln-reduction length, and a transformation is a space improvement if it never increases the space consumption.

Definition 4.3. *Let* s,t *be two expressions with* $s \sim_c t$. *Then s is a* maxspace-improvement *of t,* $s \leq_{maxspace} t$, *if for all contexts C: If* $C[s]$, $C[t]$ *are closed then* $spmax(C[s]) \leq spmax(C[t])$.
We say s (time-)improves *t,* $s \preceq t$, *if for all contexts C: If* $C[s]$, $C[t]$ *are closed, then* $rln(C[s]) \leq rln(C[t])$.

These relations are precongruences. Note that we use $n < \infty$, and $\infty \leq \infty$.

5 An Abstract Machine for LRP

In this section we present the abstract machine (a variant of the Sestoft-machine) to evaluate LRP-programs and measure their time and space usage. However, the conceptually simple abstract machine has to be extended and adapted to obtain a good behavior w.r.t. space measuring: it must be able to remove unused bindings in `letrec`s, and it has to prevent superfluous duplications of expressions in the input as well as their dynamic creation. A first step is to transform the LRP-expressions into so-called machine expressions on which the Sestoft-machine can be applied. These are LRP-expressions with the restriction that arguments of applications, constructor applications, and the second argument of `seq` must be variables. We also remove all type information.

Definition 5.1. *The translation ψ from arbitrary LRP-expressions into machine expressions is defined as follows, where y, y_i are fresh variables:*

$$\begin{array}{llll}
\psi(x :: \rho) & := x & \psi(s\ t) & := \texttt{letrec}\ y = \psi(t)\ \texttt{in}\ (\psi(s)\ y) \\
\psi(s\ \tau) & := \psi(s) & \psi(\texttt{seq}\ s\ t) & := \texttt{letrec}\ y = \psi(t)\ \texttt{in}\ (\texttt{seq}\ \psi(s)\ y) \\
\psi(\Lambda a_1.\Lambda a_2.\ldots.\Lambda a_k.\lambda x{::}\tau.s) & := \lambda x.\psi(s) & \psi(c\ \overrightarrow{s}) & := \texttt{letrec}\ \{y_i = \psi(s_i)\}_{i=1}^{n}\ \texttt{in}\ (c\ \overrightarrow{y_i}) \\
\psi(\texttt{letrec}\ \{x_i = s_i\}_{i=1}^{n}\ \texttt{in}\ t) & := \texttt{letrec}\ \{x_i = \psi(s_i)\}_{i=1}^{n}\ \texttt{in}\ \psi(t) & & \\
\multicolumn{4}{l}{\psi(\texttt{case}_K\ e\ \texttt{of}\ \{(Pat_{K,1}\ \texttt{->} t_1)\ \ldots\ (Pat_{K,|D_K|}\ \texttt{->} t_{|D_K|})\})} \\
 & \multicolumn{3}{l}{:= \texttt{case}_K\ \psi(e)\ \texttt{of}\ \{(Pat_{K,1}\ \texttt{->}\ \psi(t_1))\ldots(Pat_{K,|D_K|}\ \texttt{->}\ \psi(t_{|D_K|}))\}}
\end{array}$$

The transformation adds `letrec`-expressions and removes type annotations. This transformation does not change the reduction length, i.e. $\mathrm{rln}(s) = \mathrm{rln}(\psi(s))$ (see [13]). It is easy to see that $\texttt{size}(s) = \texttt{size}(\psi(s))$ holds. Below we will show that for machine expressions s the value $spmax(s)$ is correctly computed. Unfortunately, this does not hold in general: for example $((\texttt{seq True}\ (\lambda x.a))\ \texttt{True})$ and $(\texttt{letrec}\ x_1 = \texttt{True}, x_2 = \lambda x.a\ \texttt{in}\ (\texttt{seq True}\ x_2)\ x_1)$ have different *spmax*-values for $\texttt{size}(a) \geq 1$: $5 + \texttt{size}(a)$ and $4 + 2\texttt{size}(a)$, respectively, since the latter has a space peak at $(\texttt{letrec}\ x_1 = \texttt{True}, x_2 = \lambda x.a\ \texttt{in}\ (\lambda x.a)\ x_1)$.

The used abstract machine is defined in [14, 13] and is based on the abstract machine Mark 1 by Peter Sestoft (see [17]), which was designed for call-by-need evaluation. The machine is extended in a straightforward way to handle `seq`-expressions, where a `seq`-expression evaluates the first argument to a value and then returns the second argument. A state Q is a triple $\langle \Gamma \mid s \mid S \rangle$, where Γ is an environment of variable-to-expression bindings (sometimes called heap), s is a machine expression (often called control expression) and S is a stack with entries #app(x), #seq(x), #case($alts$) and #upd(x) where x is a variable and $alts$ is a list of case alternatives. Because the stack is implemented as a list we sometimes use the usual list notation for the stack. The control expression is the expression which has to be evaluated next, together with the stack it controls the control flow of the program. The stack is also responsible to trigger updates on the heap. Note that the WHNFs of the abstract machine are machine expressions that are WHNFs.

The abstract machine is defined in Fig. 4. The execution of a program starts with the whole program as control expression and an empty heap and stack. The transition rules define the transition from one state to the next, where at most one rule is applicable in each step.

The rules (Unwind1), (Unwind2), and (Unwind3) perform the search for the redex (according to the labeling in *LRP*), by storing arguments of applications, `seq`-expressions, or `case`-alternatives on the stack. The rule (Lookup) moves heap bindings into the scope of evaluation (if they are demanded). If evaluation of a binding is finished, the rule (Update) restores the result in the heap. (Letrec) moves `letrec`-bindings into the heap, by creating new heap bindings. (Subst) is applicable if the first argument of an application is evaluated to an abstraction and the stack contains the argument. It then performs a β-reduction (with a variable as argument). (Branch) analogously performs a (case)-reduction on the abstract machine. (Seq) evaluates a `seq`-expression. Rule (Blackhole) results in an infinite loop, i.e. an error.

The abstract machine iteratively applies these rules until a final state is reached. Note that the control expression of a state is a Mark 1 value if no rule is applicable.

The (optional) rule (GC) performs garbage collection of bindings. The (optional) rule (SCRem) performs a specific form of saving space: it prevents unnecessary copying of values by avoiding the intermediate construction of indirections $y = x$ and applying the replacement instead. For a correct space measurement, these rules have to be applied whenever possible.

Initial state: $\langle \emptyset \mid e \mid [] \rangle$ where e is a machine expression.
Transition rules:
(Unwind1) $\langle \Gamma \mid (s\ x) \mid S \rangle \to \langle \Gamma \mid s \mid \#\text{app}(x) : S \rangle$
(Unwind2) $\langle \Gamma \mid (\texttt{seq}\ s\ x) \mid S \rangle \to \langle \Gamma \mid s \mid \#\text{seq}(x) : S \rangle$
(Unwind3) $\langle \Gamma \mid \texttt{case}_K\ s\ \texttt{of}\ alts \mid S \rangle \to \langle \Gamma \mid s \mid \#\text{case}(alts) : S \rangle$
(Lookup) $\langle \Gamma, x = s \mid x \mid S \rangle \to \langle \Gamma \mid s \mid \#\text{upd}(x) : S \rangle$
(Letrec) $\langle \Gamma \mid \texttt{letrec}\ Env\ \texttt{in}\ s \mid S \rangle \to \langle \Gamma, Env \mid s \mid S \rangle$
(Subst) $\langle \Gamma \mid \lambda x.s \mid \#\text{app}(y) : S \rangle \to \langle \Gamma \mid s[y/x] \mid S \rangle$
(Branch) $\langle \Gamma \mid c_{K,i}\ \overrightarrow{x} \mid \#\text{case}(\ldots\ ((c_{K,i}\ \overrightarrow{y})\texttt{->}t)\ \ldots) : S \rangle \to \langle \Gamma \mid t[\overrightarrow{x}/\overrightarrow{y}] \mid S \rangle$
(Seq) $\langle \Gamma \mid v \mid \#\text{seq}(y) : S \rangle \to \langle \Gamma \mid y \mid S \rangle$ if v is a Mark 1 value
(Update) $\langle \Gamma \mid v \mid \#\text{upd}(x) : S \rangle \to \langle \Gamma, x = v \mid v \mid S \rangle$ if v is a Mark 1 value
(Blackhole) $\langle \Gamma \mid y \mid S \rangle \to \langle \Gamma \mid y \mid S \rangle$ if no binding for y exists on the heap
Garbage Collection and Stack Chain Removal (both optional):
(GC) $\langle \Gamma, \{x_i = s_i\} \mid s \mid S \rangle \to \langle \Gamma \mid s \mid S \rangle$ where $\{x_i = s_i\}$ is the maximal set such that for all i:
$x_i \notin FV(\Gamma), x_i \notin FV(s), \#\text{app}(x_i) \notin S, \#\text{seq}(x_i) \notin S$, and if $x_i \in FV(alts)$ then $\#\text{case}(alts) \notin S$
(SCRem) $\langle \Gamma \mid s \mid \#\text{upd}(x) : \#\text{upd}(y) : S \rangle \to \langle \Gamma[x/y] \mid s[x/y] \mid \#\text{upd}(x) : S[x/y] \rangle$
Value: A machine expression is a *Mark 1 value* if it is an abstraction or constructor application.
WHNF: Let v be a Mark 1 value. Then a machine expression is a *Mark 1-WHNF* if it is a Mark 1 value or of the form $\texttt{letrec}\ x_1 = e_1,\ \ldots,\ x_n = e_n\ \texttt{in}\ v$.
Final State: Let v be a Mark 1 value, then a final state is: $\langle \Gamma \mid v \mid [] \rangle$

Figure 4: Mark1: Initial state, transition rules, value, WHNF and final state

The rule (Update) is only applicable if (Lookup) was used before, hence (Letrec) is the only rule which is able to add completely new bindings to the heap.

Moreover every (Lookup) triggers an (Update). There are situations where a variable as control expression leads to another variable as control expression (e.g. variable chains in `letrec`-environments). For example the state $\langle \Gamma \mid \texttt{True} \mid \#\text{upd}(x) : \#\text{upd}(y) : \#\text{upd}(z) : S \rangle$ leads to three (Update) in sequence. Seen as a `letrec`-environment, $\texttt{letrec}\ x = y, y = z, z = \texttt{True}$ leads to $\texttt{letrec}\ x = \texttt{True}, y = \texttt{True}, z = \texttt{True}$. But if we consider the rules in Fig. 2, we see that LRP does copy such values right to the needed position, without copying it to each position of the corresponding chain. The following example even shows that the difference in space consumption is at least $c \cdot n$, where c is the size of the value v:

$$\texttt{letrec}\ id = (\lambda x.x), x_1 = (id\ x_2), \ldots, x_{n-1} = (id\ x_n), x_n = v\ \texttt{in}\ \texttt{seq}\ x_1\ (\texttt{T}\ x_1\ x_2\ \ldots\ x_n)$$

The tuple $(\texttt{T}\ x_1\ x_2\ \ldots\ x_n)$ ensures that none of the bindings can be removed by the garbage collector.

Machine execution leads to a sequence of n (Update)-transitions, where the value v gets copied to each binding of the chain. To avoid this effect, the rule (SCRem) has to be applied whenever possible. If we consider the example above, then we have:

$$\langle \Gamma \mid \texttt{True} \mid \#\text{upd}(x) : \#\text{upd}(y) : \#\text{upd}(z) : S\rangle \xrightarrow{\text{(SCRem)},2} \langle \Gamma[x/y,x/z] \mid \texttt{True} \mid \#\text{upd}(x) : S[x/y,x/z]\rangle$$

The rule (SCRem) is correct, since $\langle \Gamma \mid v \mid \#\text{upd}(x) : \#\text{upd}(y)\rangle$ corresponds to $\texttt{letrec}\ \Gamma, x = v, y = x\ \texttt{in}\ y$ with $x \neq y$ before application, and after the application it is $\langle \Gamma[x/y] \mid v[y/x] \mid \#\text{upd}(x)\rangle$ corresponding to $\texttt{letrec}\ \Gamma[x/y], x = v[y/x]\ \texttt{in}\ y[y/x]$ and replacing variables by variables is shown to be correct in [16].

Now we compare LRP with the abstract machine:

Definition 5.2. *Let s be a closed machine expression such that $\langle \emptyset \mid s \mid []\rangle \xrightarrow{n} Q$ where Q is a final state.*

1. *$mln(s)$ is the number of all (Subst)-, (Branch)- and (Seq)-steps in the sequence.*
2. *$mlnall(s)$ is the number of all machine steps in the sequence, thus $mlnall(s) = n$.*
3. *$mspmax(s)$ is $\max\{\texttt{size}(St_i) \mid 1 \leq i \leq n, \neg(St_{i-1} = \langle \Gamma, c\,\vec{x}, S\rangle \wedge St_{i-1} \xrightarrow{\text{Update}} St_i)\}$, (i.e. states after (Update) are ignored for constructor applications), where $\langle \emptyset \mid s \mid []\rangle = St_1 \to St_2 \to \ldots \to St_n = Q$.*

If s diverges then $mln(s) = mlnall(s) = mspmax(s) := \infty$.

The size of a machine state is the sum of the heap sizes seen as outer `letrec`, the size of the control expression and the expressions on the stack, where the #-labels are not counted. (Update) might increase the size of the current state in contrast to LRP, where variables can be processed directly without looking up and then updating them (e.g. compare (case-in) of LRP with (Branch) of the Mark 1).

We show that the abstract machine can be used for computing reduction lengths and space measures as needed for reasoning on time- and space-improvements (restricted to machine expressions in the case of space-improvements):

Theorem 5.3 (Adequacy of the abstract machine w.r.t. resource consumption)**.** *Let s be an LRP expression with $s\downarrow$.*

1. *On input $\psi(s)$, the measure $mln(\psi(s))$ coincides with $rln(s)$.*
2. *If s is a machine expression and if the abstract machine eagerly applies (GC) and (SCRem), then $mspmax(s)$ coincides with $spmax(s)$.*

Proof. Since in [13] it was shown that $\text{rln}(s) = \text{mln}(\psi(s))$ holds, LRP, restricted to machine expressions, and Mark-1 provide equal results concerning reduction lengths. Note that this does not hold for rlnall and mlnall, since the abstract machine moves `letrec`-environments directly on top, while LRP needs additional (lll)-reduction steps.

Because bindings $x = y$ are eliminated by (SCRem) the only difference between evaluating the machine expressions s in LRPgc and the evaluation of s on the abstract machine with eagerly applying rules (GC) and (SCRem) concerning space is the following: The abstract machine copies constructor applications in contrast to LRP. The constructor applications are either directly processed by a (Seq) or (Branch), or the copying is a final (Update)-transition. The claim holds, since we do not count the sizes of exactly these intermediate states between (Update) and (Seq) as well as (Update) and (Branch), and a final (Update) in the computation of $mspmax(s)$. □

5.1 Implementation

The LRP interpreter (LRPi) is implemented in Haskell and can be downloaded here:

`http://www.ki.informatik.uni-frankfurt.de/research/lrpi`

All details concerning compilation can be found on this page. The interpreter is able to execute LRP-programs and to generate statistics concerning reduction lengths and different space measures. Various size and space measures can be defined easily, thus the interpreter can be used to compare different size and space measurements or to explore other resource usages apart from time and space analyses.
The interpreter is user friendly and is able to calculate TikZ-pictures showing the `size`-values during runtime (for use in LaTeX).

The rule (GC) is implemented as a stop-and-copy garbage collector that is called by the abstract machine depending on the garbage collection mode. If we set the garbage collector to run after each state transition, then the reduction length and *spmax*-results (restricted to LRP-machine-expressions in the case of space measurement) are correctly counted for LRPgc, since the interpreter automatically applies (SCRem) whenever possible.

5.1.1 Removing Indirection Chains

We support the interpretation by two initial operations: There is a complete garbage collection before starting the interpretation, and an efficient algorithm to remove chains of indirections (variable-variable binding chains) in the input expression, which avoids unnecessary space consumption in the Sestoft machine. The algorithm is only applied once at compile time, since none of the rules in Fig. 4 create variable-to-variable bindings that cannot be removed by (SCRem). Since we often configure the garbage collector to run whenever possible, this can reduce the runtime of garbage collection runs for large programs. This is implemented efficiently and runs in time $\mathcal{O}(n \log n)$ where n is the number of variables. For more information see [1].

6 Analyses for Examples

This section contains analyses illustrating the performance and output of the interpreter and tool LRPi. In particular it shows several experiments: a simple program transformation, various fold-applications and a fusion, comparing two list-reverse variants, and sharing vs. non-sharing. The latter is illustrated by an example that can be seen as a variant of common subexpression elimination which shows that saving space may increase the runtime and that a transformation, which for a large class of tests reduces the space, might fail to be a space improvement in some cases. All analyses are done after translating the input to machine expression format.

One of the aims of LRPi is to support conjectures of space improvements by affirmative tests, or to refute the space improvement property of a specific transformation by finding a counter example. Since LRPi only tests in the empty environment, a complete test would require to perform the test also within contexts, which, however, cannot be done completely, since there are infinitely many, even using context lemmas to minimize the set of necessary contexts. Using a simulation mode, the contexts could be restricted to testing the functions on arguments. For these tests typing makes a big difference, since certain transformations are correct only if typing is respected and also the space improvement property may depend on the restriction to typed arguments or type-correct insertion into contexts.
Examples for conjectured space improvements are the reductions of the calculus (see Fig.2) used as transformations, with the exception of the (cp)-reductions.

The used function definitions can be found in Fig. 5. The `fold`-function definitions are taken from [15], `concat` and `concatMap` are inlined versions of the definitions in [6]. We first consider `fold`-functions of Haskell. `foldr` is the usual right-fold, `foldl` the usual left-fold and `foldl'` a more strict variant of `foldl`, which is not completely strict, since the used `seq` only evaluates w until a value is achieved. Following [15], we use the LRPi to find an example in which `foldl` is worse than `foldr` if

```
comp      = λf,g.(λx.f (g x))
foldr     = λf,z,xs.case xs of {([] -> z)
                ((y : ys) -> f y (foldr f z ys))}
foldl     = λf,z,xs.case xs of {([] -> z)
                ((y : ys) -> foldl f (f z y) ys)}
foldl'    = λf,z,xs.case xs of {([] -> z)
                ((y : ys) ->
                    letrec w = (f z y)
                      in seq w (foldl' f w ys))}
map       = λf,lst.case lst of {([] -> [])
                ((x : xs) -> ((f x) : (map f xs)))}
tail      = λlst.case lst of {
                    ([] -> ⊥) ((x : xs) -> xs)}
replicate = λn,x.case n of {(Zero -> [])
              ((Succ m) -> x : (replicate m x))}
last      = λlst.case lst of {(x : xs) ->
                case xs of {([] -> x)
                          ((y : ys) -> last xs)}}
```

```
reverse   = λxs.case xs of {([] -> [])
                ((y : ys) -> reverse ys ++ [y])}
reverse'  = λxs.reversew [] xs
reversew  = λxs,ys.case ys of {([] -> xs)
                ((z : zs) -> reversew (z : xs) zs)}
(++)      = λxs,ys.case xs of {([] -> ys)
                ((z : zs) -> z : (zs ++ ys))}
concat    = λxs.(foldr
                (λx,y.foldr (λz,zs.(z : zs)) y x)
                    [] xs)
concatMap = λf,xs.(foldr
                (λx,b.foldr
                    (λz,zs.(z : zs)) b (f x))
                [] xs)
xor       = λx,y.case x of {
                (True -> case y of {
                        (True -> False)
                        (False -> True)})
                (False -> y)}
```

Figure 5: Several function definitions

the preconditions on arguments are not fulfilled. Choosing `xor` for f and `False` as e, the requirement $f\ e\ s \preceq f\ s\ e$ holds, but the requirement $(f\ (f\ s_1\ s_2)\ s_3) \preceq (f\ s_1\ (f\ s_2\ s_3))$ is not fulfilled for $s_1 =$ `True`, $s_2 =$ `False`, $s_3 =$ `False`. A list starting with a single `True` element followed by $k-1$ `False`-elements generated using a take-function/list generator approach (using a Peano encoding to represent the numbers) is used as input list.

We configure LRPi to collect garbage whenever possible. As we will see, `foldr` indeed has a better runtime behavior than `foldl` and the space consumption of `foldr` and `foldl'` are almost equal. Moreover, we see that `foldl` has a much worse space behavior than `foldl'`. This difference is caused by the known stack problems of `foldl` that can be solved in the case of `xor` by using `foldl'` instead.

We can identify the stack overflow problem (of fold-expressions) in the space diagram in Fig. 7 using $k = 250$, directly calculated by LRPi. Let s_i be the i-th expression during execution. Because of lazy evaluation, the `foldl`-expression is expanded step by step without calculating any intermediate results until `foldl` itself is no longer required and is removed by the garbage collector. This leaves a long chain of nested `(++)`-function calls that lead to the big rise of the curve, because this causes a long chain of (lbeta)- and (case)-transformations. The small decrease before the rise of the curve is caused by the removal of `foldl` by the garbage collector, because the definition of `foldl` is not needed anymore after the expansion is completed. Note that (gc)-reductions are not counted by mlnall, but counted in the following diagrams in Fig. 7.

k	25	50	75	100	125	150	175	200	225	250
	foldl using xor									
mln	302	602	902	1202	1502	1802	2102	2402	2702	3002
mlnall	1085	2160	3235	4310	5385	6460	7535	8610	9685	10760
spmax	217	417	617	817	1017	1217	1417	1617	1817	2017
	foldl′ using xor									
mln	327	652	977	1302	1627	1952	2277	2602	2927	3252
mlnall	1235	2460	3685	4910	6135	7360	8585	9810	11035	12260
spmax	87	112	137	162	187	212	237	262	287	312
	foldr using xor									
mln	279	554	829	1104	1379	1654	1929	2204	2479	2754
mlnall	1016	2016	3016	4016	5016	6016	7016	8016	9016	10016
spmax	90	115	140	165	190	215	240	265	290	315

Figure 6: Table of analysis results for different `fold`-variants

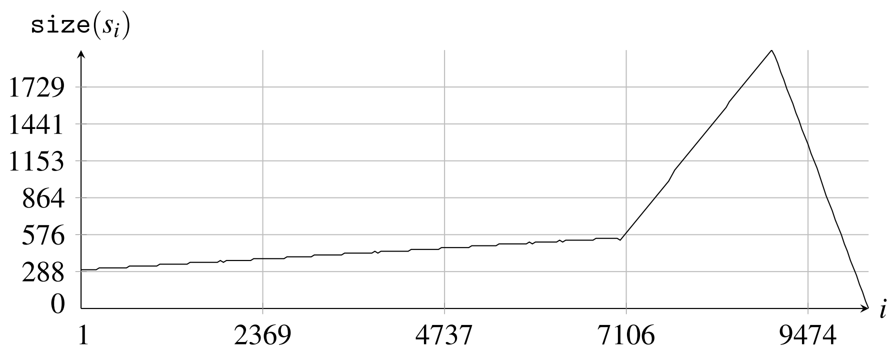


Figure 7: Size diagram for `foldl` using `xor` and input size $k = 250$

We now want to compare `reverse` with `reverse′` in Fig 8. We use `last` to force the evaluation and moreover we create a list containing k times the element `True` using `replicate` k `True`. This supports the following conjectures on complexities: `reverse` requires quadratic runtime, caused by the left-associativity of `(++)` while `reverse′` requires linear runtime. Because `(++)` only goes through each intermediate list, `reverse` appears to not need asymptotically more space than `reverse′`. Both `reverse` and `reverse′` appear to have a linear space complexity, perhaps `reverse′` has smaller constants in the asymptotic complexity formula.

Now we want to have a short look on fusion. The composition of functions can lead to well readable programs, because recursions are hidden and the main steps of the calculation are clearly visible. But this leads to intermediate structures and to an increase of the reduction length and especially space consumption, if we use a realistic (non-eager) garbage collector. The Glasgow Haskell Compiler (GHC) uses the so called short cut fusion as introduced in [2]. This approach eliminates such intermediate tree and list structures to gain a better runtime and to reduce the needed space.

As shown in [7], short cut fusion might be unsafe if `seq` is used, but in the majority of cases this

k	50	100	150	200	250	300	350	400
	`last (reverse (replicate` k `True))`							
mln	4230	15955	35180	61905	96130	137855	187080	243805
mlnall	15799	59074	129849	228124	353899	507174	687949	896224
spmax	462	862	1262	1662	2062	2462	2862	3262
	`last (reverse' (replicate` k `True))`							
mln	457	907	1357	1807	2257	2707	3157	3607
mlnall	1782	3532	5282	7032	8782	10532	12282	14032
spmax	100	150	200	250	300	350	400	450

Figure 8: Comparing two reverse variants

k	100	200	300	400	500	600	700	800	900	1000
	Difference of reduction lengths between fused and unfused									
Δ mln	206	406	606	806	1006	1206	1406	1606	1806	2006
Δ mlnall	623	1223	1823	2423	3023	3623	4223	4823	5423	6023
	Difference of *spmax* between fused and unfused									
Δ Eager	14	14	14	14	14	14	14	14	14	14
Δ Every 1000th	47	47	47	47	47	47	47	47	47	47
Δ Every 2000th	60	60	60	60	60	60	60	60	60	60
Δ Never	132	232	332	432	532	632	732	832	932	1032

Figure 9: Differences in time and space between fused and unfused `concatMap`

approach works and is used by the GHC. Moreover [18] shows that this approach might increase sharing and therefore a part of the memory is longer used. Thus it may increase the space consumption.

We now want to compare `(comp concat map) tail` with `concatMap tail`. As input we use a list containing k inner lists of the form `[True,True]`, again generated by a list-generator/take-function approach. The differences in the table are the unfused version minus the fused version. The results are in Fig. 9. As expected the reduction length and space consumption behaves linearly in all cases. We also see that the frequency of the garbage collector directly affects the space consumption, if we compare each garbage collection mode of the fused with the unfused version. The rarer the garbage collector runs the higher is the difference in space consumption: If we turn off the garbage collector and use the fused version instead of the unfused version, then the decrease of space consumption is linear in the length of the list.

With regard to *nogc* the advantage concerning space consumption of the fused versions over the unfused versions of the above examples is only constant, but the advantage is even linear if we turn off garbage collection. Thus the above examples for fusion are space improvements in a weak sense. Practically, the weak space improvements above are very useful because they are also time improvements.

The final example is a case where the decrease of space consumption behaves inverse to time consumption. The example experiments in Fig. 10 reports on comparing $(list \text{ ++ } list)$ `++` $(list \text{ ++ } list)$ with `let` $xs = list$ `in` $(xs \text{ ++ } xs)$ `++` $(xs \text{ ++ } xs)$ (written here in Haskell notation), driving evaluation

k	12	13	14	200	400	600	800	1000
	Shared append							
mln	297	321	345	4809	9609	14409	19209	24009
mlnall	1152	1245	1338	18636	37236	55836	74436	93036
spmax	77	79	81	453	853	1253	1653	2053
	Unshared append							
mln	453	489	525	7221	14421	21621	28821	36021
mlnall	1730	1867	2004	27486	54886	82286	109686	137086
spmax	78	79	80	266	466	666	866	1066

Figure 10: Shared versus unshared append

using the `last` function, and where `++` is the the `append` function. The first expression has four separate occurrences of a (long) $list$, whereas the second expression shares the $list$s, where $list$ varies in length in the experiments. The results are consistent with the claim that common subexpression elimination (cse) is a time improvement [14], and show that (cse) and an increase of sharing in general may increase the (maximal) space usage. In neither direction the example is a space improvement, which shows that (cse) is not a space improvement.

7 Conclusion and Future Work

We demonstrated that the interpreter LRPi is a useful tool for exploring improvements. The conceptual work on it also had an influence on constructing appropriate models of resource consumption. Among the influences are: the calculus must incorporate (gc), and the Sestoft machine turned out to have a non-optimal space behavior, which had to be improved. We expect that in the future there will be more influences and feedback in both directions between measuring tool with its experiments and the theory.

Future research into the relations between calculus, machine translations and abstract machine is justified. Further work is to extend LRPi also taking contexts (according to Def. 4.3) into account, or automating the inspection of series of arguments, in order to improve the affirmative power for space improvements. Moreover, a more practical integer representation would be helpful, since Peano encodings affect and pollute the space measurement. Also refining the garbage collection (for example locally generated garbage) is an issue.

Acknowledgments

We thank David Sabel for discussions and hints which were very helpful in improving the paper. We also thank the reviewers of WPTE for the numerous helpful remarks.

References

[1] Nils Dallmeyer (2016): *Design and implementation of a test suite for exploring space improvements in a call-by-need functional language with polymorphic types*. Msc. thesis, Institut für Informatik, J.W.Goethe-University Frankfurt.

[2] Andrew Gill, John Launchbury & Simon L. Peyton Jones (1993): *A Short Cut to Deforestation*. In: *Proc. Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, ACM, New York, NY, USA, pp. 223–232, doi:10.1145/165180.165214.

[3] Jörgen Gustavsson (2001): *Space-Safe Transformations and Usage Analysis for Call-by-Need Languages*. PhD thesis, Department of Computing Science, Chalmers University of Technology and Göteborg University.

[4] Jörgen Gustavsson & David Sands (1999): *A Foundation for Space-Safe Transformations of Call-by-Need Programs*. *Electronic Notes in Theoretical Computer Science* 26, pp. 69 – 86. HOOTS '99, Higher Order Operational Techniques in Semantics, doi:10.1016/S1571-0661(05)80284-1.

[5] Jörgen Gustavsson & David Sands (2001): *Possibilities and Limitations of Call-by-need Space Improvement*. In: *Proc. Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, ACM, New York, NY, USA, pp. 265–276, doi:10.1145/507635.507667.

[6] Hackage (2016): *Hackage, the base package*. Available at `http://hackage.haskell.org/package/base`.

[7] Patricia Johann & Janis Voigtländer (2006): *The Impact of Seq on Free Theorems-Based Program Transformations*. *Fundam. Inf.* 69(1-2), pp. 63–102.

[8] Andrew Moran & David Sands (1999): *Improvement in a Lazy Context: An Operational Theory for Call-by-need*. In: *Proc. 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, ACM, New York, NY, USA, pp. 43–56, doi:10.1145/292540.292547.

[9] David Sands (1991): *Operational Theories of Improvement in Functional Languages (Extended Abstract)*. In: *Proc. Fourth Glasgow Workshop on Functional Programming*, Springer-Verlag, pp. 298–311, doi:10.1007/978-1-4471-3196-0_24.

[10] David Sands (1995): *A Naïve Time Analysis and its Theory of Cost Equivalence*. *Journal of Logic and Computation* 5, pp. 495–541, doi:10.1093/logcom/5.4.495.

[11] David Sands (1995): *Total Correctness by Local Improvement in Program Transformation*. In: *Proc. 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, ACM, New York, NY, USA, pp. 221–232, doi:10.1145/199448.199485.

[12] 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.

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

[14] Manfred Schmidt-Schauß & David Sabel (2015): *Improvements in a functional core language with call-by-need operational semantics*. In Moreno Falaschi & Elvira Albert, editors: *Proc. 17th International Symposium on Principles and Practice of Declarative Programming*, ACM, pp. 220–231, doi:10.1145/2790449.2790512.

[15] Manfred Schmidt-Schauß & David Sabel (2015): *Sharing-Aware Improvements in a Call-by-Need Functional Core Language*. In Ralf Lämmel, editor: *Proceedings of IFL*, IFL '15, ACM, New York, NY, USA, pp. 6:1–6:12, doi:10.1145/2897336.2897343.

[16] Manfred Schmidt-Schauß, David Sabel & Marko Schütz (2008): *Safety of Nöcker's Strictness Analysis*. *J. Funct. Program.* 18(4), pp. 503–551, doi:10.1017/S0956796807006624.

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

[18] Josef Svenningsson (2002): *Shortcut Fusion for Accumulating Parameters & Zip-like Functions*. In: *Proc. Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, ACM, New York, NY, USA, pp. 124–132, doi:10.1145/581478.581491.