An efficient composition of bidirectional programs by tupling and lazy updates

First Author[1][0000−1111−2222−3333], Second Author[2,3][1111−2222−3333−4444], and Third Author[3][2222−−3333−4444−5555]

[1] Princeton University, Princeton NJ 08544, USA
[2] Springer Heidelberg, Tiergartenstr. 17, 69121 Heidelberg, Germany
`lncs@springer.com`
http://www.springer.com/gp/computer-science/lncs
[3] ABC Institute, Rupert-Karls-University Heidelberg, Heidelberg, Germany
{`abc,lncs`}`@uni-heidelberg.de`

Abstract. Bidirectional transformation (BX) is a solution of view update problems and widely used for synchronizing data. Its semantics and correctness are researched well, but the efficiency and optimization are not considered well. In this work we focus on the evaluation of composition, because it includes some problems of evaluation time efficiency and memory allocation. To solve these problems we make *put* and *get* more tight, based on the idea of tupling. Because simple tupled result includes some redundancies for left associative compositions, we apply two optimization techniques: lazy update and lazy evaluation. We show some experimental results that our optimized approach is faster than the original approach that is used in an actual BX language.

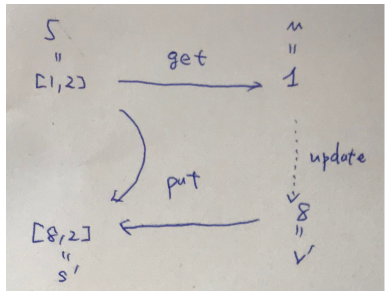
Keywords: Bidirectional transformation · Tupling · Optimization · ??.

- FLOPS 2020, 15 November 2019 (AoE): Abstract submission
- 22 November 2019 (AoE): Submission deadline
- keywords, ORCID id, introduction, contribution, future work, related work

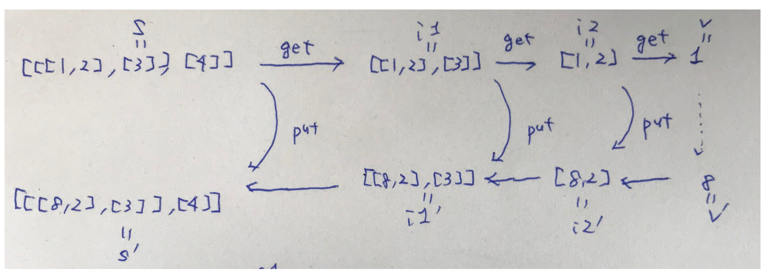
1 Introduction

In software, there are strong demands for synchronizing data. In database community this is known as "view update problem" and researched for a long time [1]. As a solution for this problem, bidirectional transformation (BX) is introduced []. As an example, let us consider a small BX program of $pHead$[4]. BX provides two functions: *get* and *put*. The following is one example of $pHead$'s evaluation.

[4] The actual program is shown in the next section.



Here, the original source `s`, `[1,2]`, is given by a user. *get* is a projection: *get* of *pHead* picks the first element of the given original source as a view. After the view `v` is obtained, the user can modify the view. In this case, he or she modified the view to `8` from `1`. From this updated view, how can we obtain new updated source? This is "view update problem." For this, BX provides *put*, update on the original source. In this example, *put* of *pHead* constructs a new list `[8,2]` from the updated view (v') and the original source (s).
For more various evaluations, we can use composition of programs. Let us consider another BX example program, three composition of *pHead*: $pHead \circ pHead \circ pHead$. The following is one example of its evaluation.

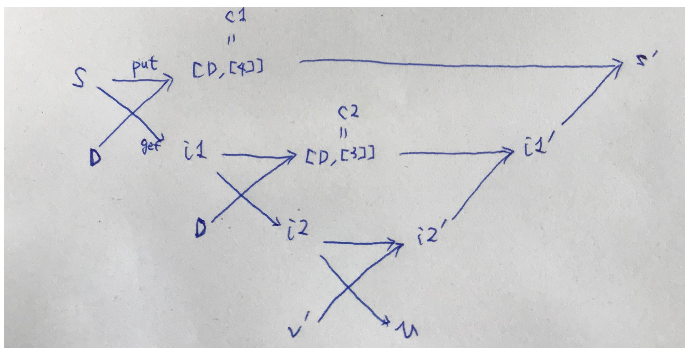


In this example, the original source `s` is `[[[1,2],[3]],[4]]`. By repetitious evaluation of *get*s of *pHead*, we can obtain the view v, `1`. To obtain the final updated source `s'`, we need to evaluate *put* three times. The first put evaluation is from `i2` and `v'` and we can obtain `i2'`.
For evaluating put-direction, there are two strategies. One is "not keeping any intermediate states and obtaining them by evaluation when they are needed." In this example, "intermediate states" are `i1` and `i2`. If we choose this strategy, the number of *get*s will be quadratic. This is a problem, because the evaluation will be slow if the BX programs include many compositions. In this example, two *get*s are required for obtaining `i2` and one *get* is required for obtaining `i1`. In total, three *get*s are required.
The other strategy is "keeping all intermediate states and using them when they are needed." This strategy causes a storage problem. If views includes most information, for evaluating *put* the needed parts of the original source will be small. It is redundant to keep all intermediate states.

In this work, we choose another strategy "keeping complements and using them when they are needed." Complements are smaller program fragments than the original intermediate states. Therefore this strategy solves the previous two problems. Readers might already know that it is a hard problem to obtain complements []. In our work, thanks to the following finding it is not hard:

In very-well behaved BX programs, *put* can be a complement function for *get*, *get* can be a complement function for *put*.

For using complement, we combine *put* and *get*, and produce new function *pg* in Section 3. This is a kind of tupling. Because parts of *put* and *get* are doing the same computation, we can make efficient by tupling them.
As an example, let us consider the previous example again: $pHead \circ pHead \circ pHead$.



In the figure, `c1` and `c2` are complements. The points in this figure are two. After evaluation of the first *pg*, we do not need to keep the original source `s`, because all its information is in `c1` and `i1`. This evaluation looks better than previous two strategies: this does not require repeated evaluation and require smaller storage than the original sources. Actually, the simple combined *pg* is not effective for left associative compositions. To achieve efficient evaluation, we introduce *cpg*, lazy update version of *pg*, in Section 4 and *kpg*, lazy evaluation version of *cpg*, in Section 5. In Section 6 we combine *pg* and *kpg* to achieve efficiency for both associative compositions. In Section 7 we compare all approaches including the original BX approach by the evaluation time and memory allocation.

Contribution of this paper We can summarize contribution of this paper as follows.

- Improvement of evaluation efficiency
  - This is both in evaluation time and memory allocation
- Optimization by tupling
  - In BX, as far as authors know optimization by *get* and *put* more tight is the first attempt

2 Bidirectional Programming Language: minBiGUL

MinBiGUL, our target language in this paper, is a subset of BiGUL [?] which is a simple yet powerful putback-based bidirectional language. BiGUL supports two transformations: a forward transformation *get* producing a view from a source and a backward transformation *put* taking a source and a modified view to produce an updated source. Intuitively, if we have a BiGUL program bx, these two transformations are following functions:

$get\,[\![bx]\!] : s \to v, \quad put\,[\![bx]\!] : s * v \to v$

BiGUL is well-behaved since two functions $put\,[\![bx]\!]$ and $get\,[\![bx]\!]$ satisfy the round-trip laws as follows:

$put\,[\![bx]\!]\ s\ (get\,[\![bx]\!]\ s) = s \qquad$ [GetPut]

$get\,[\![bx]\!]\ (put\,[\![bx]\!]\ s\ v) = v \qquad$ [PutGet]

The GetPut law means that if there is no change to the view, there should be no change to the source. The PutGet law means that we can recover the modified view by applying the forward transformation to the updated source.

MinBiGUL inherited from BiGUL also supports transformations *put* and *get* which are satify two above laws. In addition, we restrict adaptive cases of BiGUL on minBiGUL. Then *put* and *get* satify one more law, PutPut, like the following:

$put\,[\![bx]\!]\ (put\,[\![bx]\!]\ s\ v')\ v = put\,[\![bx]\!]\ s\ v \qquad$ [PutPut]

The PutPut law means that a source update should overwrite the effect of previous source updates. Due to the satisfaction of three laws, GetPut, PutGet and PutPut, minBiGUL is very well-behaved.

2.1 Syntax

The syntax of minBiGUL is briefly written as follows:

$$bx := Skip\ h \mid Replace \mid Prod\ bx_1\ bx_2 \mid RearrS\ f_1\ f_2\ bx \mid RearrV\ g_1\ g_2\ bx$$
$$\mid Case\ cond_{sv}\ cond_s\ bx_1\ bx_2 \mid Compose\ bx_1\ bx_2$$

A minBiGUL program may be either a skip of a function or a replacement or a product of two minBiGUL programs or a source/view rearrangement or a case combinatator without adaptive cases or a composition of some minBiGUL programs.

For source/view rearrangement, BiGUL use just one lambda expression to express how to deconstruct as well as reconstruct data. It is a kind of bijection. However, to be able to implement it in OCaml, the environment used for developing minBiGUL and solutions in the paper, we need give two functions which one is the inverse of the other. In the above syntax, $f_2 = f_1^{-1}$ and $g_2 = g_1^{-1}$.

To help make demonstration more direct, we provide the following alternatives representation:

$Prod\ bx_1\ bx_2 \equiv bx_1 \times bx_2, \quad Compose\ bx_1\ bx_2 \equiv bx_1 \circ bx_2$

In general, $\circ$ has a higher priority than $\times$. Since only consider a product of two minBiGUL programs is considered, it is unnecessary to define the associativity precedence of $\times$. The one of $\circ$ may be left or right, however we do not declare the default.

2.2 Semantics

Definition 1. $put\,[\![bx]\!]\ s\ v$
$put\,[\![Skip\ h]\!]\ s\ v =$
  $if\ h\ s = v\ then\ s\ else\ fail$
$put\,[\![Replace]\!]\ s\ v = v$
$put\,[\![bx_1 \times bx_2]\!]\ (s_1, s_2)\ (v_1, v_2) =$
  $((put\,[\![bx_1]\!]\ s_1\ v_1), (put\,[\![bx_2]\!]\ s_2\ v_2))$
$put\,[\![RearrS\ f_1\ f_2\ bx]\!]\ s\ v =$
  $f_2\ (put\,[\![bx]\!]\ (f_1\ s)\ v)$
$put\,[\![RearrV\ g_1\ g_2\ bx]\!]\ s\ v =$
  $put\,[\![bx]\!]\ s\ (g_1\ v)$
$put\,[\![Case\ cond_{sv}\ cond_s\ bx_1\ bx_2]\!]\ s\ v =$
  $if\ cond_{sv}\ s\ v$
  $then\ s' \Leftarrow put\,[\![bx_1]\!]\ s\ v$
  $else\ s' \Leftarrow put\,[\![bx_2]\!]\ s\ v$
  $fi\ cond_s\ s';\ return\ s'$
$put\,[\![bx_1 \circ bx_2]\!]\ s\ v =$
  $put\,[\![bx_1]\!]\ s\ (put\,[\![bx_2]\!]\ (get\,[\![bx_1]\!]\ s)\ v)$

Definition 2. $get\,[\![bx]\!]\ s$
$get\,[\![Skip\ h]\!]\ s =$
  $h\ s$
$get\,[\![Replace]\!]\ s = s$
$get\,[\![bx_1 \times bx_2]\!]\ (s_1, s_2) =$
  $((get\,[\![bx_1]\!]\ s_1), (get\,[\![bx_2]\!]\ s_2))$
$get\,[\![RearrS\ f_1\ f_2\ bx]\!]\ s =$
  $get\,[\![bx]\!]\ (f_1\ s)$
$get\,[\![RearrV\ g_1\ g_2\ bx]\!]\ s =$
  $g_2\ (get\,[\![bx]\!]\ s)$
$get\,[\![Case\ cond_{sv}\ cond_s\ bx_1\ bx_2]\!]\ s =$
  $if\ cond_s\ s$
  $then\ v' \Leftarrow get\,[\![bx_1]\!]\ s$
  $else\ v' \Leftarrow get\,[\![bx_2]\!]\ s$
  $fi\ cond_{sv}\ s\ v';\ return\ v'$
$get\,[\![bx_1 \circ bx_2]\!]\ s =$
  $get\,[\![bx_2]\!]\ (get\,[\![bx_1]\!]\ s)$

In defintions 1 and 2, we use if-then-else-fi statements to express semantics of $put\,[\![Case]\!]$ and $get\,[\![Case]\!]$. This statement is useful to describe many functions related to $Case$ in this paper. Statement (if E_1 then X_1 else X_2 fi E_2) means if the test E_1 is true, the statement X_1 is executed and the assertion E_2 must be true, otherwise, i.e. E_2 is false, the statement X_2 is executed and the assertion E_2 must be false. If the values of E_1 and E_2 are distinct, the if-then-else-fi structure is undefined. We can write the equivalent if-then-else statement as follows:

if E_1 then X_1 else X_2 fi $E_2; S$
$\equiv$ if $E_1 = true$ then $\{X_1;$ if $E_2 = true$ then S else assert $false;\}$
  else $\{X_2;$ if $E_2 = false$ then S else assert $false;\}$

Next, let's take a look at some examples to better understand about minBiGUL. We start with quite obivious things as follows:

$put\,[\![Skip\ (\lambda x.(x * x))]\!]\ 10\ 100 = 10$ $get\,[\![Skip\ (\lambda x.(x * x))]\!]\ 10 = 100$
$put\,[\![Skip\ (\lambda\_.())]\!]\ 1\ () = 1$ $get\,[\![Skip\ (\lambda\_.())]\!]\ 1 = ()$
$put\,[\![Replace]\!]\ 1\ 100 = 100$ $get\,[\![Replace]\!]\ 1 = 1$

Now, let's consider the definition of $phead$ in minBiGUL:

$phead = RearrS\ f_1\ f_2\ bx_s$ where: $f_1 = \lambda(s :: ss).(s, ss)$, $f_2 = \lambda(s, ss).(s :: ss)$,
  $bx_s = RearrV\ g_1\ g_2\ bx_v$ where: $g_1 = \lambda v.(v, ())$, $g_2 = \lambda(v, ()).v$,
    $bx_v = Replace \times (Skip\ (\lambda\_.())$

The above program rearranges the source, a non-empty list, to a pair of its head element s and its tail ss, and the view to a pair $(v, ())$, then we can use v to replace s and $()$ to keep ss. Intuitively, $put\,[\![phead]\!]\ s_0\ v_0$ returns a list whose head is v_0 and tail is the tail of s_0, and $get\,[\![phead]\!]\ s_0$ returns the head of the list s_0. For instance, $put\,[\![phead]\!]\ [1, 2, 3]\ 100 = [100, 2, 3]$ and $get\,[\![phead]\!]\ [1, 2, 3] = 1$. If we wanna update the head element of the head element of a list of lists by using the view, we can define a composition like $phead \circ phead$. For example:

$$put\,[\![phead \circ phead]\!]\ [[1,2,3],[\,],[4,5]]\ 100 = [[100,2,3],[\,],[4,5]]$$
$$get\,[\![phead \circ phead]\!]\ [[1,2,3],[\,],[4,5]] = 1$$

3 pg

3.1 Self-inverse function: pg

In minBiGUL or even BiGUL, when evaluating a composition of many programs, *get*s are re-evaluated so many times since no intermediate state is stored during the evaluation. This is a kind of information loss. One question is whether it is possible to calculate such programs without losing information. And the answer is yes. It comes from the the idea of reversible computation where all information during the evaluation need to be kept. In minBiGUL, we can do that by tupling *put* and *get*. A pair of a source and a view is accepted as the input of a function named *pg* to produce a new pair that contains the actual result of the corresponding minBiGUL program.

Definition 3. $pg\,[\![bx]\!](s,v) = (put\,[\![bx]\!]\ s\ v, get\,[\![bx]\!]\ s)$

pg is an involution that is self-inverse. An involution is a function f that satifies $f(f(x)) = x$ for all x in the domain of f.

Proof. $pg[\![bx]\!]\ (pg\,[\![bx]\!](s,v))$
$= pg\,[\![bx]\!]((put\,[\![bx]\!]\ s\ v),(get\,[\![bx]\!]\ s))$ [*pg* definition]
$= (put\ [\![bx]\!]\ (put\,[\![bx]\!]\ s\ v)\ (get\,[\![bx]\!]\ s), get\,[\![bx]\!]\ (put\,[\![bx]\!]\ s\ v))$ [*pg* definition]
$= (put\ [\![bx]\!]\ (put\,[\![bx]\!]\ s\ v, get\,[\![bx]\!]\ s), v)$ [PutGet]
$= (put\ [\![bx]\!]\ (s, get\,[\![bx]\!]\ s), v)$ [PutPut]
$= (s,v)$ [GetPut]

3.2 Construction of pg

$$pg\,[\![Skip\ h]\!](s,v) \overset{1}{=} (\text{if } h\ s = v \text{ then } s \text{ else fail}, h\ s)$$
$$\overset{2}{=} \text{if } h\ s = v \text{ then } (s, h\ s) \text{ else fail}$$
$$\overset{3}{=} \text{if } h\ s = v \text{ then } (s, v) \text{ else fail}$$

There is a trick in the construction of $pg\,[\![Skip\ h]\!]$. The first equality is simply based on the definitions of *pg*, $put\,[\![Skip\ h]\!]$ and $get\,[\![Skip\ h]\!]$. The second one tuples two results of *put* and *get* in the body of the if-expression. And the last one is quite obivious. What we call the trick here is in the second equality where in some cases, the result of *pg* may be fail although there is no fail when evaluating $get\,[\![Skip\ h]\!]$.

$$pg\,[\![Replace]\!](s,v) = (v,s)$$
$$pg\,[\![bx_1 \times bx_2]\!]((s_1,s_2),(v_1,v_2))$$
$$\overset{1}{=} ((put\,[\![bx_1]\!]\ s_1\ v_1),(put\,[\![bx_2]\!]\ s_2\ v_2),(get\,[\![bx_1]\!]\ s_1),(get\,[\![bx_2]\!]\ s_2))$$
$$\overset{2}{=} (s_1,v_1) \Leftarrow pg\,[\![bx_1]\!](s_1,v_1);$$
$$(s_2,v_2) \Leftarrow pg\,[\![bx_2]\!](s_2,v_2);$$

$$
\begin{aligned}
&\qquad ((s_1, s_2), (v_1, v_2))\\
&pg\,[\![RearrS\ f_1\ f_2\ bx]\!](s,v) \overset{1}{=} (f_2\ (put\,[\![bx]\!]\ (f_1\ s)\ v), get\,[\![bx]\!]\ (f_1\ s))\\
&\quad \overset{2}{=} (s,v) \Leftarrow pg\,[\![bx]\!](f_1\ s, v);\\
&\qquad (f_2\ s, v)\\
&pg\,[\![RearrV\ g_1\ g_2\ bx]\!](s,v) \overset{1}{=} (put\,[\![bx]\!]\ s\ (g_1\ v), g_2\ (get\,[\![bx]\!]\ s))\\
&\quad \overset{2}{=} (s,v) \Leftarrow pg\,[\![bx]\!](s, g_1\ v);\\
&\qquad (s, g_2\ v)
\end{aligned}
$$

Constructions of pg for the replacement, the product and the source/view rearrangements are very clear when just paring put and get respectively, then doing basic changes.

$$
\begin{aligned}
&pg\,[\![Case\ cond_{sv}\ cond_s\ bx_1\ bx_2]\!](s,v)\\
&\quad \overset{1}{=} (\text{if } cond_{sv}\ s\ v \\
&\qquad \text{then } s' \Leftarrow put\,[\![bx_1]\!]\ s\ v\\
&\qquad \text{else } s' \Leftarrow put\,[\![bx_2]\!]\ s\ v\\
&\qquad \text{fi } cond_s\ s';\ \text{return } s',\\
&\qquad\quad \text{if } cond_s\ s\\
&\qquad\quad \text{then } v' \Leftarrow get\,[\![bx_1]\!]\ s\\
&\qquad\quad \text{else } v' \Leftarrow get\,[\![bx_2]\!]\ s\\
&\qquad\quad \text{fi } cond_{sv}\ s\ v';\ \text{return } v')\\
&\quad \overset{2}{=} \text{if } cond_{sv}\ s\ v\ \&\&\ cond_s\ s\\
&\qquad \text{then } (s', v') \Leftarrow pg\,[\![bx_1]\!](s,v)\\
&\qquad \text{else } (s', v') \Leftarrow pg\,[\![bx_2]\!](s,v)\\
&\qquad \text{fi } cond_s\ s'\ \&\&\ cond_{sv}\ s\ v';\ \text{return } (s', v')
\end{aligned}
$$

A restriction for $pg\,[\![Case]\!]$ needs to be introduced here. We know that there is one entering condition and one exit condition when evaluating $put\,[\![Case]\!]$ as well as $get\,[\![Case]\!]$. If a tupling put and get occurs, there will be 4 combinations of these conditions. This means that two entering conditions of $put\,[\![Case]\!]$ and $get\,[\![Case]\!]$ are not always simultaneously satified. The evaluated branches are distinct in the put and get directions for combinations $((cond_{sv}\ s\ v)\ \&\&\ (not(cond_s\ s)))$ and $((not(cond_{sv}\ s\ v)\ \&\&\ (cond_s\ s))$, which are restricted in this paper. This does not happen for the others which is used in the construction of $pg\,[\![Case]\!]$.

$$
\begin{aligned}
&pg\,[\![bx_1 \circ bx_2]\!](s,v)\\
&\quad \overset{1}{=} (put\,[\![bx_1]\!]\ s\ (put\,[\![bx_2]\!]\ (get\,[\![bx_1]\!]\ s)\ v), get\,[\![bx_2]\!]\ (get\,[\![bx_1]\!]\ s))\\
&\quad \overset{2}{=} v_1 \Leftarrow get\,[\![bx_1]\!]\ s;\\
&\qquad (s_2, v_2) \Leftarrow pg\,[\![bx_2]\!](v_1, v);\\
&\qquad (s_3, v_3) \Leftarrow pg\,[\![bx_1]\!](s, s_2);\\
&\qquad\quad (s_3, v_2)\\
&\quad \overset{3}{=} (s_1, v_1) \Leftarrow pg\,[\![bx_1]\!](s, dummy);\\
&\qquad (s_2, v_2) \Leftarrow pg\,[\![bx_2]\!](v_1, v);\\
&\qquad (s_3, v_3) \Leftarrow pg\,[\![bx_1]\!](s, s_2);\\
&\qquad\quad (s_3, v_2)\\
&\quad \overset{4}{=} (s_1, v_1) \Leftarrow pg\,[\![bx_1]\!](s, construct\_dummy\ s);\\
&\qquad (s_2, v_2) \Leftarrow pg\,[\![bx_2]\!](v_1, v);\\
&\qquad (s_3, v_3) \Leftarrow pg\,[\![bx_1]\!](s_1, s_2);\\
&\qquad\quad (s_3, v_2)
\end{aligned}
$$

The construction of $pg\,[\![bx_1 \circ bx_2]\!]$ can be considered as the soul of the pg function. The first two equalities comes from mentioned definitions and some basic transformations. The third one rewrites $v_1 \Leftarrow get\,[\![bx_1]\!]\ s$ into $(s_1, v_1) \Leftarrow pg\,[\![bx_1]\!](s, dummy)$. It is possible if we consider $get\,[\![bx_1]\!]\ s$ as the second element of $pg\,[\![bx_1]\!](s, dummy)$ where $dummy$ is a special value that makes the $put\,[\![bx_1]\!]$ valid. Due to GetPut law, we can use $get\ s$ to update the source s in the put direction. Then it is feasible to construct a $dummy$ from the source.

The last equality changes *dummy* by an application *construct_dummy s*, and also substitutes s in $(s_3, v_3) \Leftarrow pg\,[\![bx_1]\!](s, s_2)$ by s_1 which is the evaluated result of $put\,[\![bx_1]\!]\ s\ (construct\_dummy\ s)$. This transformation is possible because of the PutPut law. Then, v_3 in the result pair (s_3, v_3) equals *dummy*. So we can realize that there is no loss information when computing a composition.

4 cpg

When evaluating $pg\,[\![bx_1 \circ bx_2]\!]$, we realize that there are three *pg* calls, of which twice for $pg\,[\![bx_1]\!]$ and once for $pg\,[\![bx_2]\!]$. If a given program is a left-associative composition, the number of *pg* calls will be exponential. Therefore, the runtime inefficiency is inevitable. In this section, we introduce a new function, *cpg*, accumulates changes in the source and the view to solve that problem. $cpg[\![bx]\!](ks, kv, s, v)$ is an extension of $pg\,[\![bx]\!](s, v)$ where ks and kv are continuations used to hold the modification information, and s and v are used to keep evaluated values. The output of this function is a 4-tuple (ks, kv, s, v).
To be more convenient for presenting the definition of *cpg* as well as the other functions later, we provide some following utility functions:

$fst = \lambda(x_1, x_2).x_1 \qquad snd = \lambda(x_1, x_2).x_2$
$con = \lambda ks_1.\lambda ks_2.\lambda x.((ks_1\ x), (ks_2\ x))$

Definition 4. $cpg[\![bx]\!](ks, kv, s, v)$

$cpg[\![Skip\ h]\!](ks, kv, s, v) =\ if\ h\ s = v\ then\ (ks, kv, s, v)\ else\ fail$
$cpg[\![Replace]\!](ks, kv, s, v) = (kv, ks, v, s)$
$cpg[\![bx_1 \times bx_2]\!](ks, kv, s, v) =$
$\quad (ks_1, kv_1, s_1, v_1) \Leftarrow cpg[\![bx_1]\!](fst \circ ks, fst \circ kv, fst\ s, fst\ v);$
$\quad (ks_2, kv_2, s_2, v_2) \Leftarrow cpg[\![bx_2]\!](snd \circ ks, snd \circ kv, snd\ s, snd\ v);$
$\qquad (con\ ks_1\ ks_2, con\ kv_1\ kv_2, (s_1, s_2), (v_1, v_2))$
$cpg[\![RearrS\ f_1\ f_2\ bx]\!](ks, kv, s, v) =$
$\quad (ks, kv, s, v) \Leftarrow cpg[\![bx]\!](f_1 \circ ks, kv, f_1\ s, v);$
$\qquad (f_2 \circ ks, kv, s, v)$
$cpg[\![RearrV\ g_1\ g_2\ bx]\!](ks, kv, s, v) =$
$\quad (ks, kv, s, v) \Leftarrow cpg[\![bx]\!](ks, g_1 \circ kv, s, g_1\ v);$
$\qquad (ks, g_2 \circ kv, ks, g_2\ v)$
$cpg[\![Case\ cond_{sv}\ cond_s\ bx_1\ bx_2]\!](ks, kv, s, v) =$
$\quad if\ cond_{sv}\ s\ v\ \&\&\ cond_s\ s$
$\quad then\ (ks, kv, s', v') \Leftarrow cpg[\![bx_1]\!](ks, kv, s, v)$
$\quad else\ (ks, kv, s', v') \Leftarrow cpg[\![bx_2]\!](ks, kv, s, v)$
$\quad fi\ cond_s\ s'\ \&\&\ cond_{sv} sv';\ return\ (ks, kv, s', v')$
$cpg[\![bx_1 \circ bx_2]\!](ks, kv, s, v) =$
$\quad (ks_1, kv_1, s_1, v_1) \Leftarrow cpg[\![bx_1]\!](ks, id, s, construct\_dummy\ s);$
$\quad (ks_2, kv_2, s_2, v_2) \Leftarrow cpg[\![bx_2]\!](kv_1, kv, v_1, v);$
$\qquad (ks_1 \circ ks_2, kv_2, ks_1\ s_2, v_2)$

Apart from the last construction, the others are quite similar to the corresponding ones of *pg*, but have some updates over the source and the view on accumulative functions *ks* and *kv* respectively.

For $cpg\,[\![bx_1 \circ bx_2]\!]$, there are only two cpg calls. The first call $cpg\,[\![bx_1]\!]$ requires parameter $(ks, id, s, construct\_dummy\ s)$ where s and ks are corresponding to the source and the update over source. Because there is no real view here, we need to construct a dummy from the source. Then the continuation updating on this dummy should be initiated as an identity function. The first cpg call is assigned to a 4-tuple (ks_1, kv_1, s_1, v_1) where s_1 is redundant because it is not used in the later evaluation process. Some computations to get s_1 are seen as redundant. Such values and computations have negative impacts on the runtime as well as the memory allocation in the system. In the next assigment, a 4-tuple (ks_2, kv_2, s_2, v_2) is assigned by the second cpg call which uses the input as (kv_1, kv, v_1, v) where kv_1 and v_1 are obtained from the result of the first assignment, and kv and v come from the input. It is relatively similar to the second pg call assignment in $pg\,[\![bx_1 \circ bx_2]\!]$. After two cpg calls, a function application, $ks_1\ s_2$, is used to produce the updated source instead of calling $cpg\,[\![bx_1]\!]$ one more time like in $pg\,[\![bx_1 \circ bx_2]\!]$.
Suppose that we have a source s_0 and a view v_0. The pair of the updated source and view (s, v) where $s = put\,[\![bx]\!]\ s_0\ v_0$ and $v = get\,[\![bx]\!]\ s_0$ can be obtained using cpg as follows:

$$s \Leftarrow s_0; v \Leftarrow v_0; (ks, kv, s, v) \Leftarrow cpg[\![bx]\!](\lambda\_.s, id, s, v);$$
$$(s; v)$$

In general, the begining of a continuation should be an identity function. However, to be able to use the function application to get the result of $cpg\,[\![bx_1 \circ bx_2]\!]$, the accumulative function on the source s needs to be initiated as a constant function $\lambda\_.s$.
Suppose the begining of continuations ks and kv are ks_0 and id respectively. Let's consider $cpg[\![phead_1 \circ phead_2]\!](ks_0, id, s, v)$ where $s = [[1, 2, 3], [\,], [4, 5]]$ and $v = 100$. After the first two assignments in the definition of cpg for the composition, we have: $ks_1 = f_2 \circ (con\ (fst \circ g_1 \circ id)\ (snd \circ f_1 \circ ks_0))$ and $s_2 = [100, 2, 3]$ where $f_1 = \lambda(s :: ss).(s, ss)$, $f_2 = \lambda(s, ss).(s :: ss)$, $g_1 = \lambda v.(v, ())$.
Then: $ks_1\ s_2 = (f_2 \circ (con\ (fst \circ g_1 \circ id)\ (snd \circ f_1 \circ ks_0)))\ s_2$

$$= f_2\ (\ (fst(g_1(id(s_2)))\ ,\ snd(f_1(ks_0(s_2))))\ )$$
$$= fst(g_1(id(s_2))) :: snd(f_1(ks_0(s_2))) = [100, 2, 3] :: snd(f_1(ks_0([100, 2, 3])))$$

If ks_0 is an identity function, $ks_1\ s_2 = [100, 2, 3] :: [2, 3]$. This is an unexpected result when we target it to be the updated source. If $ks_0 = \lambda\_.s$ where $s = [[1, 2, 3], [\,], [4, 5]]$, $ks_1\ s_2 = [100, 2, 3] :: [[\,], [4, 5]] = [[100, 2, 3], [\,], [4, 5]]$. This time, the result is what we want to see. Through the above example, using the continuation ks as a constant function at the beginning contributes to keep the unchanged parts in the source.

5 kpg

In the previous section, we have known that there are some variables which are evaluated but not used later when evaluating $cpg\,[\![bx_1 \circ bx_2]\!]$. Now we introduce kpg, an extension of cpg, for keeping away such redundant computations. While cpg evaluate values eagerly, kpg does the opposite. Every values are evaluated

lazily in a computation of kpg. The input of $kpg\,[\![bx]\!]$ is expanded to a 6-tuple (ks, kv, ks', kv', s, v) where ks and kv keep the modifcation information, s and v hold evaluated values, and ks' and kv' are used for lazy evaluation of actual values. The output of this function is also a 6-tuple (ks, kv, ks', kv', s, v).
Suppose that we have a source s_0 and a view v_0. The pair of the updated source and view (s, v) where $s = put\,[\![bx]\!]\ s_0\ v_0$ and $v = get\,[\![bx]\!]\ s_0$ can be obtained using kpg as follows:

$$
\begin{array}{l}
s \Leftarrow s_0; v \Leftarrow v_0; (ks, kv, ks', kv', s, v) \Leftarrow kpg[\![bx]\!](\lambda\_.s, id, id, id, s, v);\\
\quad (ks'\ s; kv'\ v)
\end{array}
$$

The begining of accumulative functions ks' and kv' are set as identity, while ks and kv are initiated as the same with the corresponding ones in cpg.

Definition 5. $kpg[\![bx]\!](ks, kv, ks', kv', s, v)$

$$
\begin{array}{l}
kpg[\![Skip\ h]\!](ks, kv, ks', kv', s, v) =\\
\quad s \Leftarrow ks'\ s;\quad v \Leftarrow kv'\ v;\quad ks' \Leftarrow id;\quad kv' \Leftarrow id;\\
\quad \mathit{if}\ h\ s = v\ \mathit{then}\ (ks, kv, ks', kv', s, v)\ \mathit{else\ fail}\\
kpg[\![Replace]\!](ks, kv, ks', kv', s, v) = (kv, ks, kv', ks', v, s)\\
kpg[\![bx_1 \times bx_2]\!](ks, kv, ks', kv', s, v) =\\
\quad s \Leftarrow ks'\ s;\quad v \Leftarrow kv'\ v;\quad ks' \Leftarrow id;\quad kv' \Leftarrow id;\\
\quad (ks_1, kv_1, ks'_1, kv'_1, s_1, v_1) \Leftarrow kpg[\![bx_1]\!](fst \circ ks, fst \circ kv, fst \circ ks', fst \circ kv', s, v);\\
\quad (ks_2, kv_2, ks'_2, kv'_2, s_2, v_2) \Leftarrow kpg[\![bx_2]\!](snd \circ ks, snd \circ kv, snd \circ ks', snd \circ kv', s, v);\\
\qquad (con\ ks_1\ ks_2, con\ kv_1\ kv_2, con\ (ks'_1 \circ fst)\ (ks'_2 \circ snd),\\
\qquad con\ (kv'_1 \circ fst)\ (kv'_2 \circ snd),\\
\qquad (s_1, s_2), (v_1, v_2))\\
kpg[\![RearrS\ f_1\ f_2\ bx]\!](ks, kv, ks', kv', s, v) =\\
\quad (ks, kv, ks', kv', s, v) \Leftarrow kpg[\![bx]\!](f_1 \circ ks, kv, f_1 \circ ks', kv', s, v);\\
\qquad (f_2 \circ ks, kv, f_2 \circ ks', kv', s, v)\\
kpg[\![RearrV\ g_1\ g_2\ bx]\!](ks, kv, ks', kv', s, v) =\\
\quad (ks, kv, ks', kv', s, v) \Leftarrow kpg[\![bx]\!](ks, g_1 \circ kv, ks', g_1 \circ kv', s, v);\\
\qquad (ks, g_2 \circ kv, ks', g_2 \circ kv', s, v)\\
kpg[\![Case\ cond_{sv}\ cond_s\ bx_1\ bx_2]\!](ks, kv, ks', kv', s, v) =\\
\quad s \Leftarrow ks'\ s;\quad v \Leftarrow kv'\ v;\quad ks' \Leftarrow id;\quad kv' \Leftarrow id;\\
\quad \mathit{if}\ cond_{sv}\ s\ v \&\&\ cond_s\ s\\
\quad \mathit{then}\ (ks, kv, ks', kv', s', v') \Leftarrow kpg[\![bx_1]\!](ks, kv, ks', kv', s, v)\\
\quad \mathit{else}\ (ks, kv, ks', kv', s', v') \Leftarrow kpg[\![bx_2]\!](ks, kv, ks', kv', s, v)\\
\quad \mathit{fi}\ cond_s\ (ks'\ s')\ \&\&\ cond_{sv}\ s\ (kv'\ v');\ \mathit{return}\ (ks, kv, ks', kv', s', v')\\
kpg[\![bx_1 \circ bx_2]\!](ks, kv, ks', kv', s, v) =\\
\quad (ks_1, kv_1, \underline{ks'_1}, kv'_1, \underline{s_1}, v_1) \Leftarrow kpg[\![bx_1]\!](ks, id, ks', id, s, construct\_dummy\ s);\\
\quad (ks_2, kv_2, \overline{ks'_2}, kv'_2, s_2, v_2) \Leftarrow kpg[\![bx_2]\!](kv_1, kv, kv'_1, kv', v_1, v);\\
\qquad (ks_1 \circ ks_2, kv_2, ks_1 \circ ks'_2, kv'_2, s_2, v_2)
\end{array}
$$

In the construction of kpg, s and v hold actual values only in case of $Skip$ and $Case$. Except them, the functions for computation will be kept in ks' and kv'. When evaluating $kpg\,[\![bx_1 \circ bx_2]\!]$, ks'_1 and s_1 in the result of the first assignment are still redundant but they are not evaluated. In $kpg\,[\![bx_1\ prod\ bx_2]\!]$, the evaluation of ks' and kv' will be done indendently in two assingments using $kpg\,[\![bx_1]\!]$

and $kpg\,[\![bx_2]\!]$. There may be some recomputations in $fst \circ ks'$ and $snd \circ ks'$ as well as $fst \circ kv'$ and $snd \circ kv'$. It is possible to evaluate actual values in s and v before calling $kpg\,[\![bx_1]\!]$ to remove the redundancy like that.

6 xpg

So far, when computing a composition, we only recursively call an unique function, either *pg* or *cpg* or *kpg*. To be more flexible, we make a new extension, *xpg*, allowing to call external functions.

Definition 6. $xpg\,[\![bx]\!](s, v)$

$$
\begin{array}{l}
xpg\,[\![bx]\!](s, v) = same\ with\ pg\ \ if\ bx \neq bx_1 \circ bx_2 \\
xpg\,[\![bx_1 \circ bx_2]\!](s, v) = \\
\quad (ks_1, kv_1, ks'_1, kv'_1, s_1, v_1) \Leftarrow kpg[\![bx_1]\!](\lambda\_.s, id, id, id, s, construct\_dummy\ s); \\
\quad (s_2, v_2) \Leftarrow xpg\,[\![bx_2]\!](kv'_1\ v_1, v); \\
\qquad (ks_1\ s_2, v_2)
\end{array}
$$

Similar to *pg*, $xpg\,[\![bx]\!]$ accepts a pair of the source and the view (s, v) to produce the new pair. The constructions of $xpg\,[\![bx]\!]$ when bx is not a composition are the same as the ones of $pg\,[\![bx]\!]$. Note that, *xpg* is called recursively instead of *pg*. For $xpg\,[\![bx_1 \circ bx_2]\!]$, we use two function calls and a function application to calculate the result. The first call and the function application come from *kpg* approaches, while the second call is based on *pg* approach.

7 Experiments

This section describes the experiments involving $minBiGUL$, *pg*, *cpg*, *kpg* and *xpg*. The evaluation time and the memory allocation are considered in each test.

7.1 Experiment environment & Test cases

We implemented 5 approaches with OCaml 4.07.1 in the same environment as follows: macOS 10.14.6, processor Intel Core i7 (2.6 GHz), RAM 16 GB 2400 MHz DDR4. The OCaml runtime system options and garbage collection parameters are set as default.
We also conducted tests on 5 cases, including many composition styles such as left or right associative, non-recursive or recursive. Table 1 shows more details on these cases.
In table 1, s_r and v_r are respectively updated source and view which are produced by applying *put* and *get* on original source s_0 and original view v_0. This means: $s_r = put\,[\![bx]\!]\ s_0\ v_0$ and $v_r = get\,[\![bx]\!]\ s_0$, where bx is one of the 5 cases mentioned in the table. Note that the results s_r and v_r do not depend on the associative style of the composition. The first four test cases simply use n compose operators to make a composition of $n{+}1$ similar BX programs which are non-recursive. For example, *lassoc-comp-replace*, left-associative composition of *Replace*s, looks like

Table 1. Test cases

No	Name	Associative	Recursive	Input		Output	
				s_0	v_0	s_r	v_r
1	lassoc-comp-replace	left	no	1	100	100	1
2	rassoc-comp-replace	right	no	1	100	100	1
3	lassoc-comp-phead	left	no	$\underbrace{[[\ldots[1]\ldots]]}_{\text{n+1 times}}$	100	$\underbrace{[[\ldots[100]\ldots]]}_{\text{n+1 times}}$	1
4	rassoc-comp-phead	right	no	$\underbrace{[[\ldots[1]\ldots]]}_{\text{n+1 times}}$	100	$\underbrace{[[\ldots[100]\ldots]]}_{\text{n+1 times}}$	1
5	breverse	left/right	yes	[1..n]	[1..n]	[n..1]	[n..1]

$(\ldots((\mathit{Replace} \circ \mathit{Replace}) \circ \mathit{Replace}) \circ \ldots \circ \mathit{Replace}) \circ \mathit{Replace}$, while *rassoc-comp-replace*, right-associative composition of *Replace*s, is like $\mathit{Replace} \circ (\mathit{Replace} \circ (\mathit{Replace} \circ \ldots \circ (\mathit{Replace} \circ \mathit{Replace}) \ldots))$, where there are $n+1$ *Replace*s in each case. The last test case, *breverse*, is defined in terms of *bFoldr*, a putback function for *foldr* which is an important higher-order function on lists. This case is more natural than the other since bFoldr is a Case analysis whose a branch contains a composition $bx_1 \circ bx_2$ where bx_1 is a product of a simple function and $bFoldr$. It will look like a composition of more BX programs at some point if we slow down the evaluation. Note that $\circ$ has a higher priority than $\times$. This leads to seemingly impossible to transform *bReverse* as well as *bFoldr* from left-associative style to right-associative style.

7.2 Results and discussions

Firstly, from definitions of *put*, *cpg*, *kpg* and *xpg*, we have No(*cpg*) = No(*kpg*) = No(*xpg*) = No(*put*) where No(f) is the number of function call f. Here we consider that No(*xpg*) in evaluating $\mathit{xpg}\,[\![bx_1 \circ bx_2]\!]$ includes both the number of *kpg*s which are used in $\mathit{kpg}\,[\![bx_1]\!]$ and the number of *xpg*s which are used in $\mathit{xpg}\,[\![bx_2]\!]$. For right-associative and non-recursive compositions (*rassoc-comp-replace* and *rassoc-comp-phead*), No(*get*), No(*put*) and No(*pg*) are linear. For left-associative and non-recursive compositions (*lassoc-comp-replace* and *lassoc-comp-phead*), No(*put*) is still linear while No(*get*) is quadratic and No(*pg*) is under an exponential distribution. For recursive composition (*breverse*), all three are nonlinear. The appendix will provide more complete insights about those statements. Now we take a closer look at the empirical results.

Fig 1a illustrates the evaluation times in 5 test cases. *pg* times are always exponential for both left-associative compositions (*lassoc-comp-replace*, *lassoc-comp-phead*) and recursive compositions (*breverse*) since No(*pg*) in each case is under an exponential distribution. For left-associative and non-recursive compositions, *kpg* times are approximate to *xpg* times, and they are the most efficient. In case of *lassoc-comp-replace*, *kpg* time and *xpg* time are linear, *cpg* time and $minBiGUL$ time are nonlinear. They are quite reasonable because the data sizes are constant, No(*get*) is quadractic for $minBiGUL$ while there is a linear quantity of

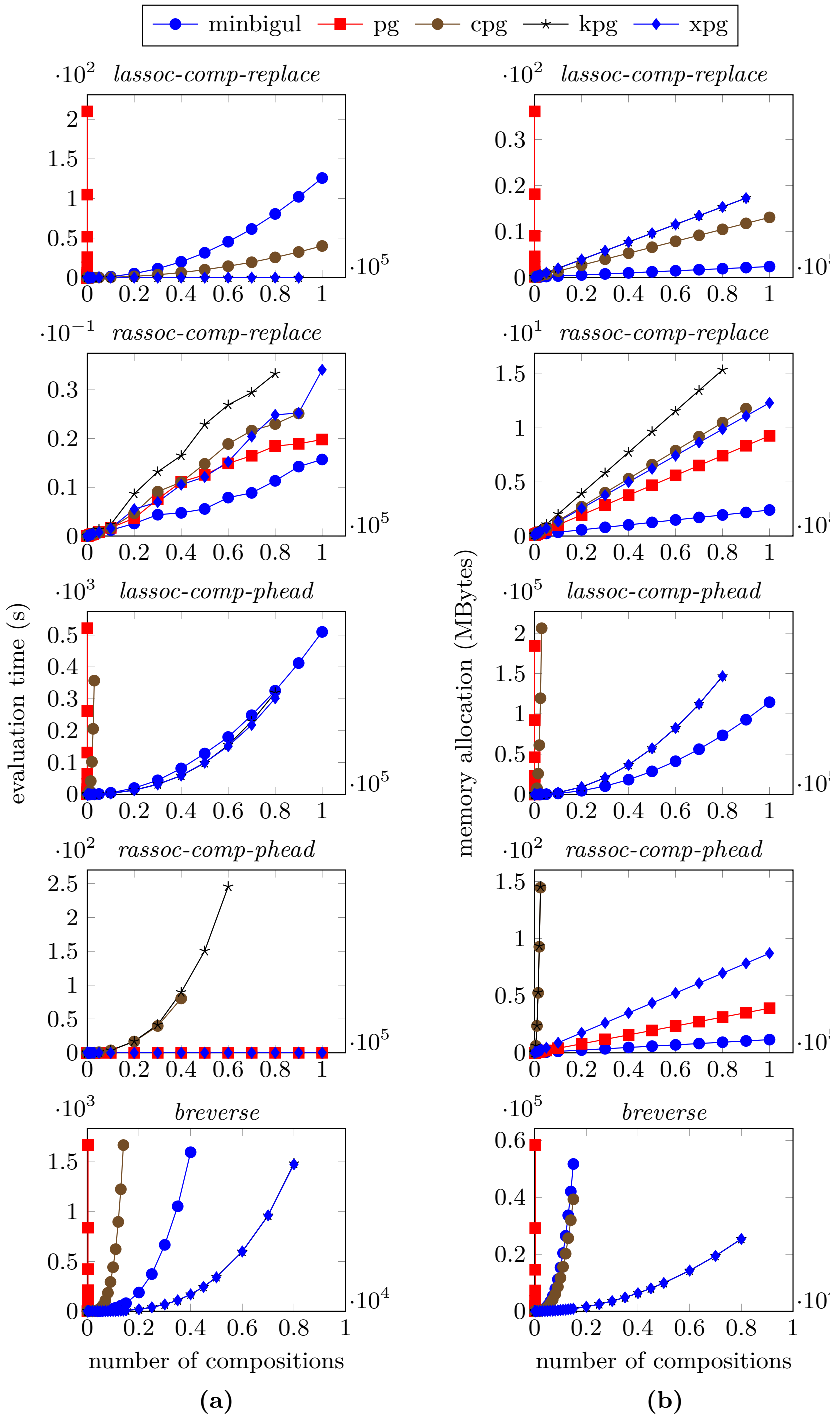


Fig. 1. Experimental results **(a)** evaluation time **(b)** memory allocation

redundant evaluations in *cpg*. In case of *lassoc-comp-phead*, since the data sizes are linear and No(f) are either linear or quadractic or exponential, all times are nonlinear. For right-associative and non-recursive compositions, $minBiGUL$, *pg* and *xpg* times are linear. In case of *rassoc-comp-replace*, all times are small enough to consider all them as linear. In case of *rassoc-comp-phead*, both *cpg* and *kpg* times are quadratic because the data sizes are linear and the number of redundant evaluations are also linear. For recursive compositions (*breverse*), all times should be nonlinear since there is no linear number of function calls.
Fig 1b shows the memory usage which depends on the input size and number of compositions. In general, $minBiGUL$ uses less memory than the others for non-recursive compositions (all tests except *breverse*). *pg* only uses memory efficient when handling right-associative compositions. Also for evaluating these compositions, both *cpg* and *kpg* are worst in memory allocation. For recursive compositions (*breverse*), $minBiGUL$ uses significantly more memory than *kpg* as well as *xpg*.

8 Related Work

BX langauges and implementations There are many BX languages [4] [6] [7] [8] [9] [10] (memo: BXtend, eMoflon, EVL+Strace, JTL, NMF. SDMLib should be added?) based on several application scenarios and BX researches [] (add papers for lenses). They mostly focus on the theory side, such as semantics and correctness, and do not focus on efficiency so much. For practical implementation of BX languages Anjorin et. al. introduces the first benchmark for BX [2]. We can say this paper will be first attempt to improve efficiency of BX composition evaluation. In this paper we focus on BiGUL [4] [5], its implementation uses "not keeping any intermediate states and obtaining them by evaluation when they are needed" strategy. Other BX languages might have the same problem, and our approach can be applicable.

Optimization techniques In this paper, we use several optimization techniques: tupling, lazy update, and lazy computation (also fusion?). Tupling [3] is an optimization technique by combining several computations. Lazy update .. delta lenses?
Lazy computation [] is also an old but effective widly-used technique.

- This part looks just short introduction of techniques ..

9 Conclusion and Future Work

In this paper we focus on efficiency of composition of BX programs. The essential finding comes from the idea of tupling: in very-well behaved BX programs we can use *put* as a compliment function for *get*, and vice versa. Based on the idea, we introduced *pg*, and improved it by several optimization techniques. From

experimental results, our fastest approach *xpg* is faster than other approaches for non purely right associative compositions. For right associative compositions, the original approach (miniBiGUL) is faster than *xpg* because *xpg* has some overhead cost. However this is not a problem, because usually programs are mix of left and right associative. If programmers know that their programs are purely right associative, they can choose miniBiGUL.
We will continue our work on the following points. First, our target language is limited to very-well behaved, because our main idea requires the put-put property. However, for practical programs, we need to extend our work to overcome the current limitations.

- Extend our approach – overcome our limitations
  - treat well-behaved programs – How to treat adaptive cases
  - In case expressions, the programs that use the different paths (put and get)
- How to obtain dummy?
- Type system & Typing – for safety

TODO: FIX reference, order, contents ..

References

1. F. Bancilhon and N. Spyratos. “Update semantics of relational views”. ACM Transactions on Database Systems, 6(4):557–575, 1981.
2. Anthony Anjorin, Thomas Buchmann, Bernhard Westfechtel, Zinovy Diskin, Hsiang-Shang Ko, Romina Eramo, Georg Hinkel, Leila Samimi-Dehkordi, and Albert Zündorf “Benchmarking bidirectional transformations: Theory, implementation, application, and assessment”, Software and Systems Modeling, ??, 2019
3. M. Fokkinga. “Tupling and mutumorphisms”. Squiggolist, 1(4), 1989.
4. Hsiang-Shang Ko, Tao Zan, Zhenjiang Hu, BiGUL: A Formally Verified Core Language for Putback-Based Bidirectional Programming, ACM SIGPLAN 2016 Workshop on Partial Evaluation and Program Manipulation (PEPM 2016), pp.61–72, 2016.
5. Hsiang-Shang Ko, Zhenjiang Hu, An Axiomatic Basis for Bidirectional Programming , 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2018), 2018.
6. Thomas Buchmann. BXtend – a framework for (bidirectional) model transformations. In Slimane Hamoudi,Luis Ferreira Pires, and Bran Selic, editors, Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD (MODELSWARD 2018), pages 336–345, 2018.
7. Erhan Leblebici, Anthony Anjorin, and Andy Sch”urr. Developing eMoflon with eMoflon. In Davide Di Ruscio and Daniel Varro, editors, Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, 2014. Proceedings, volume 8568 of Lecture Notes in Computer Science, pages 138–145. Springer, 2014.
8. Leila Samimi-Dehkordi, Bahman Zamani, and Shekoufeh Kolahdouz-Rahimi. EVL+Strace: A novel bidirectional transformation approach. Information and Software Technology, 100:47–72, 2018.

9. Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. JTL: A bidirectional and change propagating transformation language. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, Proceedings of the Third International Conference on Software Language Engineering (SLE 2010), volume 6563 of Lecture Notes of Computer Science, pages 183–202, Springer-Verlag.
10. Georg Hinkel and Erik Burger. Change propagation and bidirectionality in internal transformation DSLs. Soft-ware and Systems Modeling, 18(1):249–278, 2019.
11. SDMLib paper?
12. Lense papers?
13. Tutorial paper of BiGUL