An Efficient Composition of Bidirectional Programs by Memoization and Lazy Update

Kanae Tsushima[1], Bach Nguyen Trong[1], Robert Glück[2], and Zhenjiang Hu[3]

[1] National Institute of Informatics, Japan
{k_tsushima,bach}@nii.ac.jp
[2] University of Copenhagen, Denmark
glueck@acm.org
[3] Peking University, China
huzj@pku.edu.cn

Abstract. Bidirectional transformations (BX) are a solution to the view update problem and widely used for synchronizing data. The semantics and correctness of bidirectional programs have been investigated intensively during the past years, but their efficiency and optimization are not yet fully understood. In this paper, as a first step, we study different evaluation methods to optimize their evaluation. We focus on the interpretive evaluation of BX compositions because we found that these compositions are an important cause of redundant computations if the compositions are not right associative. For evaluating BX compositions efficiently, we investigate two memoization methods. The first method, minBiGUL$_m$, uses memoization, which improves the runtime of many BX programs by keeping intermediate results for later reuse. A disadvantage is the familiar tradeoff for keeping and searching values in a table. When inputs become large, the overhead increases and the effectiveness decreases. To deal with large inputs, we introduce the second method, *xpg*, that uses tupling, lazy update and lazy evaluation as optimizations. Lazy updates delay updates in closures and enables them to use them later. Both evaluation methods were fully implemented for minBiGUL. The experimental results show that our methods are faster than the original method of BiGUL for the non-right associative compositions.

Keywords: Bidirectional transformation · Implementation technique · Efficiency · Optimization · Tupling.

1 Introduction

The synchronization of data is a common problem. In the database community this problem is known as "the view update problem" and has been investigated for a long time [1]. Bidirectional transformation (BX) provides a systematic approach to solving this problem. Consider a small BX program of *phead*[4], which consists of two functions: *get* (for getting the head of an input list) and *put* (for

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

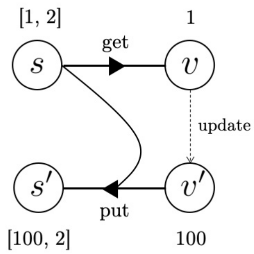

Fig. 1. Evaluating *phead*

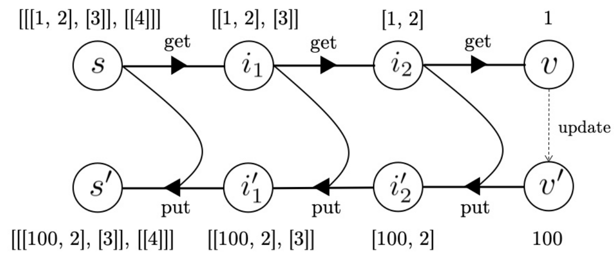

Fig. 2. Evaluating $phead \mathbin{\tilde{\circ}} phead \mathbin{\tilde{\circ}} phead$

reflecting the output to the head of the input). Figure 1 shows an example of the bidirectional behavior of *phead*. Let $[1, 2]$ be the original source s. The function *get* is a projection: *get* of *phead* picks the first element of the given original source $[1, 2]$ and returns 1 as a view v. Supposing that the view is updated to 100, *put* of *phead* will construct a new source s' of $[100, 2]$ from the updated view v' of 100 and the original source s of $[1, 2]$.

The composition of BX programs is a fundamental construct to build more complex BX programs [2, 3]. Let bx_1 (defined by get_{bx_1} and put_{bx_1}) and bx_2 (defined by get_{bx_2} and put_{bx_2}) be two bidirectional programs, then their composition $bx_1 \mathbin{\tilde{\circ}} bx_2$ is defined by

$$get_{bx_1 \tilde{\circ} bx_2}\ s = get_{bx_2} (get_{bx_1}\ s) \tag{1}$$

$$put_{bx_1 \tilde{\circ} bx_2}\ s\ v' = put_{bx_1} s\ (put_{bx_2}\ (get_{bx_1}\ s)\ v') \tag{2}$$

Unlike function composition, the composition of bidirectional programs is read left-to-right. We use this order because it is helpful to understand the behavior if we consider data flows from left to right. One feature of this composition is that $put_{bx_1 \tilde{\circ} bx_2}$ needs to call get_{bx_1} to compute the intermediate result for put_{bx_2} to use, which would introduce an efficiency problem if we compute *put* for composition of many bidirectional programs. Generally, for a composition of $O(n)$ bidirectional programs, we need to call *get* for $O(n^2)$ times. To be concrete, consider the evaluation of the following composition (which will be used as our running example in this paper):

$$lp3 = (phead \mathbin{\tilde{\circ}} phead) \mathbin{\tilde{\circ}} phead$$

which is illustrated by Figure 2 with the original source s being $[[[1, 2], [3]], [[4]]]$ and the updated view 100. To obtain the final updated source s', *put* for $lp3$ needs to evaluate *put* of *phead* three times. The first is from i_2 and v' to obtain i'_2, which needs to call *get* twice to compute i_2; the second is from i_1 and i'_2 to obtain i'_1, which needs to call *get* once, and the last is from s and i'_1 to obtain s', which is just a direct *put* computation.

One direct solution to avoid this repeated *get* computation is to compute compositions in a right associative manner. For instance, if we transform $lp3$ to $rp3$:

$$rp3 = phead \mathbin{\tilde{\circ}} (phead \mathbin{\tilde{\circ}} phead)$$

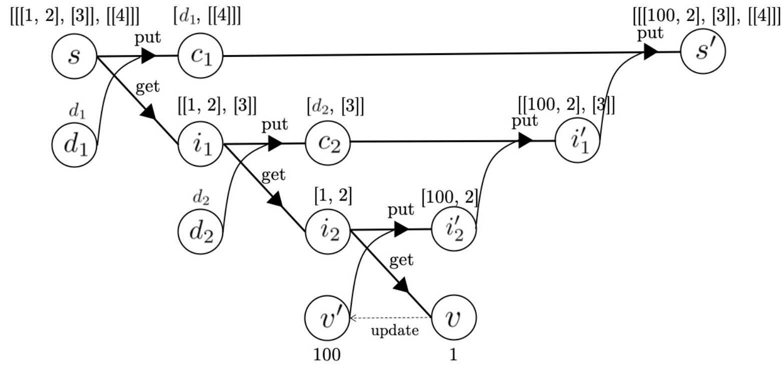

Fig. 3. Evaluating *phead* $\tilde{\circ}$ *phead* $\tilde{\circ}$ *phead* by keeping complements

then the *put* for *rp*3 only needs to compute *get* of *phead* twice, one time less than that for *lp*3. However, this transformation is not always easy to do. For instance, let us consider *breverse*, a bidirectional version of the traditional 'reverse' program for reversing a list. It is defined using *bfoldr*, a bidirectional version of the traditional *foldr*, whose definition is shown in the last part of Section 2. Informally, *bfoldr* is a recursive bidirectional program defined in a way like

$$\mathit{bfoldr}\ \mathit{bx}\ \cdots = \cdots (\mathit{bfoldr}\ \cdots)\ \tilde{\circ}\ \mathit{bx} \cdots$$

where the composition is inherently left associative, and the number of composition is dynamically determined by the length of the source list. This makes it hard to do the above transformation statically. The same efficiency problem occurs in all BX languages.

In this paper, we make the first attempt for seriously considering the efficiency of evaluating BX compositions, and solve the problem by introducing two methods based on memoization to gain fast evaluation for (left associative) BX compositions. The first method uses straightforward memoization: "keeping *intermediate states in a table* and using them when needed". This avoids repeated *get* computations and improves the runtime in many cases. However, this simple memoization needs to keep and search values in a table, which may introduce big cost for large inputs. We explain this method in Section 3.

To treat large inputs, we propose the second method based on memoization: "keeping *complements in a closure* and using them when needed". Here, complements are information from sources that makes *get* injective, which is in turn needed to evaluate *put*. In the middle *put* and *get* of Figure 2, we use i_1, $[[1,2],[3]]$ and i'_2, $[100,2]$ to obtain the updated source $[[100,2],[3]]$. However, $[1,2]$ is simply replaced by $[100,2]$ and not used to construct the result of *put*. In this case we can use [..., [3]] as a complement. The key idea of the second approach is straightforward: Complements are smaller than intermediate states. For obtaining complements, we tuple *put* and *get*, and produce a new function *pg*. Because *put* produces new complements for *get*, we can shrink the size. Let us reconsider our example *phead* $\tilde{\circ}$ *phead* $\tilde{\circ}$ *phead* in Figure 3, where c_1 and c_2 are complements and d_1 and d_2 are valid views for s and i_1. Here, two points are worth noting. First, after evaluating the leftmost *pg*, the original source s need not be kept because its complete contents are in c_1 and i_1. Second, the complements are smaller than the intermediate states in Figure 2. Actually, this simple *pg* alone is not yet effective for left associative compositions because it requires two more *put*s, which can be seen on the right side of Figure 3. To achieve an efficient evaluation, we combine two techniques, lazy update and lazy evaluation. We explain this second method and all optimizations in Section 4.

Both methods have been fully implemented for *minBiGUL*, a core bidirectional language, which is a subset of the full bidirectional language *BiGUL*. The experimental results show that our methods are much faster than the original evaluation strategy. We give detailed experimental results in Section 5, discuss related work in Section 6, and conclude in Section 7.

Although we will introduce the basics of bidirectional transformation in the next session, it is not complete due to space limitations. Please refer BiGUL papers [7, 14] for the details if needed.

2 Bidirectional Programming Language: minBiGUL

The target language in this paper, minBiGUL, is a very-well behaved 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 the following functions:

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

BiGUL is well-behaved [4] since two functions $\mathit{put}\,[\![bx]\!]$ and $\mathit{get}\,[\![bx]\!]$ satisfy the round-trip laws as follows:

$$\begin{array}{ll} \mathit{put}\,[\![bx]\!]\ s\ (\mathit{get}\,[\![bx]\!]\ s) = s & [\textsc{GetPut}] \\ \mathit{get}\,[\![bx]\!]\ (\mathit{put}\,[\![bx]\!]\ s\ v) = v & [\textsc{PutGet}] \end{array}$$

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 inherits from BiGUL both transformations, *put* and *get*, which satisfy the two laws above. Because we restrict the 'adaptive case' of BiGUL in minBiGUL, *put* and *get* satisfy one more law, namely the PutPut law [5]:

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

The PutPut law means that a source update should overwrite the effect of previous source updates. Because minBiGUL satisfies all three laws, GetPut, PutGet and PutPut, it is very well-behaved [5].

2.1 Syntax

The syntax of minBiGUL is briefly written as follows:

$$
\begin{aligned}
bx ::= {} & \mathit{Skip}\ h \mid \mathit{Replace} \mid \mathit{Prod}\ bx_1\ bx_2 \mid \mathit{RearrS}\ f_1\ f_2\ bx \mid \mathit{RearrV}\ g_1\ g_2\ bx \\
& \mid \mathit{Case}\ \mathit{cond}_{sv}\ \mathit{cond}_s\ bx_1\ bx_2 \mid \mathit{Compose}\ bx_1\ bx_2
\end{aligned}
$$

A minBiGUL program is either a skip of a function, a replacement, a product of two programs, a source/view rearrangement, a case combinator (without adaptive cases), or a composition of two programs. We use numbers, pairs and lists to construct the program inputs including the source and/or the view.

For source/view rearrangement, BiGUL uses a 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 to require 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: $\mathit{Prod}\ bx_1\ bx_2 \equiv bx_1 \times bx_2$, $\mathit{Compose}\ bx_1\ bx_2 \equiv bx_1 \circ bx_2$. The compose symbol $\tilde{\circ}$ used in the previous section will be replaced with the more common one, $\circ$. In general, $\circ$ has a higher priority than $\times$. Their associativity precedence can be either left or right or mixture, but are not set by default. We need to explicitly write programs that use these operators.

2.2 Semantics

The semantics of *put* and *get* is shown in Definitions 1 and 2, respectively. Instead of using the name v' for the updated view in the *put* direction, like Figures 1, 2 and 3, we simply use v below. The later definitions also follow this convention.

Definition 1. $\mathit{put}\,[\![bx]\!]\ s\ v$

$$
\begin{aligned}
& \mathit{put}\,[\![\mathit{Skip}\ h]\!]\ s\ v = \\
& \quad \text{if } h\ s = v \text{ then } s \text{ else } \mathit{undefined} \\
& \mathit{put}\,[\![\mathit{Replace}]\!]\ s\ v = v \\
& \mathit{put}\,[\![bx_1 \times bx_2]\!]\ (s_1, s_2)\ (v_1, v_2) = \\
& \quad (\mathit{put}\,[\![bx_1]\!]\ s_1\ v_1, \mathit{put}\,[\![bx_2]\!]\ s_2\ v_2) \\
& \mathit{put}\,[\![\mathit{RearrS}\ f_1\ f_2\ bx]\!]\ s\ v = \\
& \quad f_2\ (\mathit{put}\,[\![bx]\!]\ (f_1\ s)\ v) \\
& \mathit{put}\,[\![\mathit{RearrV}\ g_1\ g_2\ bx]\!]\ s\ v = \\
& \quad \mathit{put}\,[\![bx]\!]\ s\ (g_1\ v) \\
& \mathit{put}\,[\![\mathit{Case}\ \mathit{cond}_{sv}\ \mathit{cond}_s\ bx_1\ bx_2]\!]\ s\ v = \\
& \quad \text{if } \mathit{cond}_{sv}\ s\ v \\
& \quad \text{then } s' \Leftarrow \mathit{put}\,[\![bx_1]\!]\ s\ v \\
& \quad \text{else } s' \Leftarrow \mathit{put}\,[\![bx_2]\!]\ s\ v \\
& \quad \text{fi } \mathit{cond}_s\ s';\ \text{return } s' \\
& \mathit{put}\,[\![bx_1 \circ bx_2]\!]\ s\ v = \\
& \quad \mathit{put}\,[\![bx_1]\!]\ s\ (\mathit{put}\,[\![bx_2]\!]\ (\mathit{get}\,[\![bx_1]\!]\ s)\ v)
\end{aligned}
$$

Definition 2. $\mathit{get}\,[\![bx]\!]\ s$

$$
\begin{aligned}
& \mathit{get}\,[\![\mathit{Skip}\ h]\!]\ s = \\
& \quad h\ s \\
& \mathit{get}\,[\![\mathit{Replace}]\!]\ s = s \\
& \mathit{get}\,[\![bx_1 \times bx_2]\!]\ (s_1, s_2) = \\
& \quad (\mathit{get}\,[\![bx_1]\!]\ s_1, \mathit{get}\,[\![bx_2]\!]\ s_2) \\
& \mathit{get}\,[\![\mathit{RearrS}\ f_1\ f_2\ bx]\!]\ s = \\
& \quad \mathit{get}\,[\![bx]\!]\ (f_1\ s) \\
& \mathit{get}\,[\![\mathit{RearrV}\ g_1\ g_2\ bx]\!]\ s = \\
& \quad g_2\ (\mathit{get}\,[\![bx]\!]\ s) \\
& \mathit{get}\,[\![\mathit{Case}\ \mathit{cond}_{sv}\ \mathit{cond}_s\ bx_1\ bx_2]\!]\ s = \\
& \quad \text{if } \mathit{cond}_s\ s \\
& \quad \text{then } v' \Leftarrow \mathit{get}\,[\![bx_1]\!]\ s \\
& \quad \text{else } v' \Leftarrow \mathit{get}\,[\![bx_2]\!]\ s \\
& \quad \text{fi } \mathit{cond}_{sv}\ s\ v';\ \text{return } v' \\
& \mathit{get}\,[\![bx_1 \circ bx_2]\!]\ s = \\
& \quad \mathit{get}\,[\![bx_2]\!]\ (\mathit{get}\,[\![bx_1]\!]\ s)
\end{aligned}
$$

The two definitions use if-then-else-fi statements to define the semantics of $\mathit{put}\,[\![\mathit{Case}]\!]$ and $\mathit{get}\,[\![\mathit{Case}]\!]$, where $\Leftarrow$ denotes an assignment. 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, if E_1 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:

$$
\begin{aligned}
&\text{if } E_1 \text{ then } X_1 \text{ else } X_2 \text{ fi } E_2; S\\
&\equiv \text{if } E_1 = \mathit{true} \text{ then } \{X_1;\ \text{if } E_2 = \mathit{true} \text{ then } S \text{ else } \mathit{undefined}\}\\
&\quad \text{else } \{X_2;\ \text{if } E_2 = \mathit{false} \text{ then } S \text{ else } \mathit{undefined}\}
\end{aligned}
$$

Also in the semantics of $put\,[\![Case]\!]$ and $get\,[\![Case]\!]$, the return statements are used to express clearly the value of functions. Variables s' and v' wrapped in these returns are necessary for checking the fi conditions.

2.3 Examples

As an example of minBiGUL program, consider the definition of *phead*:

$$
\begin{aligned}
&phead = RearrS\ f_1\ f_2\ bx_s \text{ where: } f_1 = \lambda(s :: ss).(s, ss),\ f_2 = \lambda(s, ss).(s :: ss),\\
&\quad bx_s = RearrV\ g_1\ g_2\ bx_v \text{ where: } g_1 = \lambda v.(v, ()),\ g_2 = \lambda(v, ()).v,\\
&\qquad bx_v = Replace \times (Skip\ (\lambda\_.())
\end{aligned}
$$

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 want to 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:

$$
\begin{aligned}
&put\,[\![phead \circ phead]\!]\ [[1,2,3],[\,],[4,5]]\ 100 = [[100,2,3],[\,],[4,5]]\\
&get\,[\![phead \circ phead]\!]\ [[1,2,3],[\,],[4,5]] = 1
\end{aligned}
$$

In the same way with *phead*, we can define *ptail* in minBiGUL. $put\,[\![ptail]\!]\ s\ v$ accepts a source list s and a view list v to produce a new list by replacing the tail of s with v. $get\,[\![ptail]\!]\ s$ returns the tail of the source list s

Next let us look at another more complex example, *bsnoc*:

$$
\begin{aligned}
&bsnoc = Case\ cond_{sv}\ cond_s\ bx_1\ bx_2 \text{ where:}\\
&\ cond_{sv} = \lambda s.\lambda v.(\text{length } v = 1),\ cond_s = \lambda s.(\text{length } s = 1)\\
&\ bx_1 = Replace, \quad bx_2 = RearrS\ f_1\ f_2\ bx_s \text{ where:}\\
&\quad f_1 = f_2^{-1} = \lambda(x : y : ys).(y, (x : ys)),\\
&\quad bx_s = RearrV\ g_1\ g_2\ bx_v \text{ where:}\\
&\qquad g_1 = g_2^{-1} = \lambda(v : vs).(v, vs), \quad bx_v = Replace \times bsnoc
\end{aligned}
$$

$put\,[\![bsnoc]\!]$ requires the source s and the view v are non-empty lists and the length of v is not larger than the length of s. If $cond_{sv}$ is true, i.e. v is singleton, a replacement will be executed to produce a new list which should be equal to v. Because the length of the new list is 1, the exit condition $cond_s$ comes true, so we obtain the updated source. If v is a list of more than one elements, there will be two rearrangements on the source and the view before conducting a product.

The program rearranges the source $x : y : ys$ to a pair of its second element y and a list $x : ys$ created from the remaining elements in the original order, and the view to a pair of its head and tail. Then we can use y to replace the head of the view and pair $(x : ys)$ with the tail of the view to form the input of a recursive call *bsnoc*. The obtained source update in this case should be non-singleton since the value of the exit condition $cond_s$ needs to be false. We omit the behavior description of $get\,[\![bsnoc]\!]$ that accepts a source list s, checks $cond_s$ to know how to evaluate the view v, then does one more checking, $cond_{sv}$, before resulting. Intuitively, $put\,[\![bsnoc]\!]\ s_0\ v_0$ produces a new list by moving the last element of v_0 to its first position if the length of v_0 is not larger than the length of s_0. $get\,[\![bsnoc]\!]\ s_0$ returns another list by moving the first element of the list s_0 to its end position. For instance, $put\,[\![bsnoc]\!]\ [1,2,3]\ [4,5,6] = [6,4,5]$ and $get\,[\![bsnoc]\!]\ [1,2,3] = [2,3,1]$.

Now, let us see the minBiGUL definition of *bfoldr* which is a putback function of an important higher-order function on lists, *foldr*:

$$
\begin{array}{l}
\mathit{bfoldr}\ bx = Case\ cond_{sv}\ cond_s\ bx_1\ bx_2 \text{ where:}\\
\quad cond_{sv} = \lambda(s_1,s_2).\lambda v.(s_1 = [\,]),\ cond_s = \lambda(s_1,s_2).(s_1 = [\,])\\
\quad bx_1 = \mathit{RearrV}\ g_1\ g_2\ bx_v \text{ where:}\\
\qquad g_1 = g_2^{-1} = \lambda[v].(v,[\,]), bx_v = (\mathit{Skip}\ (\lambda\_.())) \times \mathit{Replace}\\
\quad bx_2 = \mathit{RearrS}\ f_1\ f_2\ bx_s \text{ where:}\\
\qquad f_1 = f_2^{-1} = \lambda((x:xs),e).(x,(xs,e)),\ bx_s = ((\mathit{Replace} \times \mathit{bfoldr}\ bx) \circ bx)
\end{array}
$$

If we think that a minBiGUL program *bx* has a type of $\mathit{MinBiGUL}\ s\ v$, the type of $bfoldr$ will be look like $\mathit{MinBiGUL}\ (a,b)\ b \rightarrow \mathit{MinBiGUL}\ ([a],b)\ b$. You can easily find the similarity between the above definition of *bfoldr* with the following definition of *foldr*:

$$
\begin{array}{l}
\mathit{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b\\
\mathit{foldr}\ f\ e\ [\,] = e\\
\mathit{foldr}\ f\ e\ (x:xs) = f\ x\ (\mathit{foldr}\ f\ e\ xs)
\end{array}
$$

Each branch in a case of *bfoldr* corresponds to a pattern of $foldr$. In *bfoldr*, the composition is inherently left associative, and the number of composition is dynamically determined by the length of the source list. Because $\circ$ has a higher priority than $\times$, it is in general not possible to transform *bfoldr* from the left associative composition style to the right one.

Using *foldr*, we can define other functions like $reverse = \mathit{foldr}\ \mathit{snoc}\ [\,]$. With *bfoldr*, we can also write the bidirectional version $breverse$ as follows:

$$
\begin{array}{l}
breverse = RearrS\ f_1\ f_2\ bx \text{ where:}\\
\qquad f_1 = f_2^{-1} = \lambda s \rightarrow (s,[\,]),\quad bx = \mathit{bfoldr}\ \mathit{bsnoc}
\end{array}
$$

3 Adding Memoization: minBiGUL$_m$

When evaluating the composition of several BX programs, the same *get*s are evaluated repeatedly. This problem was illustrated in Figure 2. To avoid reevaluating *get*s, and as our first approach to avoid this inefficiency, we introduce memoization in the minBiGUL interpreter. To keep it simple, the intermediate state of a composition is saved in a key-value table where the key is a pair of program bx and source s, and the value is the result of evaluating $get\,[\![bx]\!]\ s$. Later the value in the table is used instead of recomputing it.

The memoizing version, minBiGUL$_m$, needs only two modifications: get_m and put_m (Definitions 3 and 4).

Definition 3. *Memoization version of put*

$put_m\,[\![bx]\!]\ s\ v = \text{match}\ bx\ \ \text{with}$
$\quad|\ bx_1 \circ bx_2 \rightarrow put_m\,[\![bx_1]\!]\ s\ (put_m\,[\![bx_2]\!]\ (get_m\,[\![bx_1]\!]\ s)\ v)$
$\quad|\ \_ \rightarrow$ *similar to put*

Definition 4. *Memoization version of get*

$get_m\,[\![bx]\!]\ s = \text{match}\ bx\ \ \text{with}$
$\quad|\ bx_1 \circ bx_2 \rightarrow$
$\qquad\text{try}\ (Hashtbl.find\ table_g\ \ (bx, s))$
$\qquad\text{with}\ Not\_found\ \rightarrow$
$\qquad\quad i \Leftarrow get_m\,[\![bx_1]\!]\ s;\quad Hashtbl.add\ \ table_g\ (bx_1, s)\ i;$
$\qquad\quad v \Leftarrow get_m\,[\![bx_2]\!]\ i;\quad Hashtbl.add\ \ table_g\ (bx_2, i)\ v;$
$\qquad\quad Hashtbl.add\ \ table_g\ (bx, s)\ v$
$\qquad\quad v$
$\quad|\ \_ \rightarrow$ *similar to get*

The evaluation of $put_m\,[\![bx_1 \circ bx_2]\!]\ s\ v$ includes two recursive calls of put_m and an external call of get_m, which is relatively similar to the evaluation of $put\,[\![bx_1 \circ bx_2]\!]\ s\ v$. Meanwhile, the evaluation of $get_m\,[\![bx_1 \circ bx_2]\!]\ s$ does not merely invoke get_m recursively twice. In case that bx is a composition, the key (bx, s) needs to be looked up in the table and the corresponding value would be used for the next steps in the evaluation. If there is no such key, the value of the intermediate state i and the value of $get\,[\![bx]\!]\ s$ in v will be calculated. These values along with the corresponding keys will also be stored in the table where the interpreter may later leverage instead of reevaluating some states. Note in particular that the interpreter does not save all states when evaluating a program, only the intermediate states of a composition.

4 Tupling and Lazy Updates: xpg

4.1 Tupling: pg

Another solution for saving intermediate states is tupling. If *put* and *get* are evaluated simultaneously, there is potential to reduce the number of recomputed *get*s. The following function, *pg*, accepts the pair of a source and a view as the input to produce a new pair that contains the actual result of the corresponding minBiGUL program.

Definition 5. $pg\,[\![bx]\!](s, v) = (put\,[\![bx]\!]\ s\ v, get\,[\![bx]\!]\ s)$

Now, let us see how we construct *pg* recursively.

$$
\begin{aligned}
& pg\,[\![Skip\ h]\!](s,v) \stackrel{1}{=} (\text{if } h\ s = v \text{ then } s \text{ else } \mathit{undefined}, h\ s) \\
& \quad \stackrel{2}{=} \text{if } h\ s = v \text{ then } (s, h\ s) \text{ else } \mathit{undefined} \\
& \quad \stackrel{3}{=} \text{if } h\ s = v \text{ then } (s, v) \text{ else } \mathit{undefined}
\end{aligned}
$$

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. This is a trick since in some cases, the result of *pg* may be undefined although the result is not undefined when evaluating $get\,[\![Skip\ h]\!]$. The last equality is a function application.

$$
\begin{aligned}
& pg\,[\![Replace]\!](s,v) = (v,s) \\
& pg\,[\![bx_1 \times bx_2]\!]((s_1,s_2),(v_1,v_2)) \\
& \quad \stackrel{1}{=} ((put\,[\![bx_1]\!]\ s_1\ v_1, put\,[\![bx_2]\!]\ s_2\ v_2), (get\,[\![bx_1]\!]\ s_1, get\,[\![bx_2]\!]\ s_2)) \\
& \quad \stackrel{2}{=} (s'_1, v'_1) \Leftarrow pg\,[\![bx_1]\!](s_1,v_1); \\
& \qquad (s'_2, v'_2) \Leftarrow pg\,[\![bx_2]\!](s_2,v_2); \\
& \qquad\quad ((s'_1, s'_2), (v'_1, v'_2)) \\
& pg\,[\![RearrS\ f_1\ f_2\ bx]\!](s,v) \stackrel{1}{=} (f_2\ (put\,[\![bx]\!]\ (f_1\ s)\ v), get\,[\![bx]\!]\ (f_1\ s)) \\
& \quad \stackrel{2}{=} (s', v') \Leftarrow pg\,[\![bx]\!](f_1\ s, v); \\
& \qquad\quad (f_2\ s', v') \\
& pg\,[\![RearrV\ g_1\ g_2\ bx]\!](s,v) \stackrel{1}{=} (put\,[\![bx]\!]\ s\ (g_1\ v), g_2\ (get\,[\![bx]\!]\ s)) \\
& \quad \stackrel{2}{=} (s', v') \Leftarrow pg\,[\![bx]\!](s, g_1\ v); \\
& \qquad\quad (s', g_2\ v')
\end{aligned}
$$

Constructions of *pg* for the replacement, the product and the source/view rearrangements are simple. We just pair *put* and *get*, and change them to *pg*. The values of these *pg* functions are obtained from the final expression in the corresponding sequences. We only use the return keyword to express explicitly the evaluated value of a function in the situation of *Case*.

$$
\begin{aligned}
& pg\,[\![Case\ cond_{sv}\ cond_s\ bx_1\ bx_2]\!](s,v) \\
& \quad \stackrel{1}{=} (\text{if } cond_{sv}\ s\ v \qquad\qquad\qquad\quad \text{if } cond_s\ s \\
& \qquad \text{then } s' \Leftarrow put\,[\![bx_1]\!]\ s\ v \qquad\quad \text{then } v' \Leftarrow get\,[\![bx_1]\!]\ s \\
& \qquad \text{else } s' \Leftarrow put\,[\![bx_2]\!]\ s\ v \qquad\quad \text{else } v' \Leftarrow get\,[\![bx_2]\!]\ s \\
& \qquad \text{fi } cond_s\ s';\ \text{return } s' \qquad ,\quad \text{fi } cond_{sv}\ s\ v';\ \text{return } v') \\
& \quad \stackrel{2}{=} \text{if } cond_{sv}\ s\ v\ \&\&\ cond_s\ s \text{ then} \\
& \qquad\quad (s', v') \Leftarrow (put\,[\![bx_1]\!]\ s\ v, get\,[\![bx_1]\!]\ s); \\
& \qquad\quad \text{if } cond_s\ s'\ \&\&\ cond_{sv}\ s\ v' \text{ then return } (s', v') \text{ else } \mathit{undefined} \\
& \qquad \text{else if } cond_{sv}\ s\ v\ \&\&\ not\ (cond_s\ s) \text{ then} \\
& \qquad\quad (s', v') \Leftarrow (put\,[\![bx_1]\!]\ s\ v, get\,[\![bx_2]\!]\ s); \\
& \qquad\quad \text{if } cond_s\ s'\ \&\&\ not\ (cond_{sv}\ s\ v') \text{ then return } (s', v') \text{ else } \mathit{undefined} \\
& \qquad \text{else if } not\ (cond_{sv}\ s\ v)\ \&\&\ cond_s\ s \text{ then} \\
& \qquad\quad (s', v') \Leftarrow (put\,[\![bx_2]\!]\ s\ v, get\,[\![bx_1]\!]\ s); \\
& \qquad\quad \text{if } not\ (cond_s\ s')\ \&\&\ cond_{sv}\ s\ v' \text{ then return } (s', v') \text{ else } \mathit{undefined} \\
& \qquad \text{else if } not\ (cond_{sv}\ s\ v)\ \&\&\ not\ (cond_s\ s) \text{ then}
\end{aligned}
$$

$$
\begin{aligned}
&\qquad (s', v') \Leftarrow (put\,[\![bx_2]\!]\ s\ v, get\,[\![bx_2]\!]\ s);\\
&\qquad \text{if } not\ (cond_s\ s')\ \&\&\ not\ (cond_{sv}\ s\ v') \text{ then return } (s', v') \text{ else } undefined\\
&\overset{3}{=} (*\text{ with restriction } *)\\
&\quad \text{if } cond_{sv}\ s\ v\ \&\&\ cond_s\ s\\
&\quad \text{then } (s', v') \Leftarrow pg\,[\![bx_1]\!](s, v)\\
&\quad \text{else } (s', v') \Leftarrow pg\,[\![bx_2]\!](s, v)\\
&\quad \text{fi } cond_s\ s'\ \&\&\ cond_{sv}\ s\ v';\ \text{return } (s', v')
\end{aligned}
$$

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 occurs, there will be 4 combinations from these conditions. This means two entering conditions of $put\,[\![Case]\!]$ and $get\,[\![Case]\!]$ are not always simultaneously satisfied. 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. Because they evaluate different *bx* for *put* and *get*, we can not evaluate them efficiently. This does not happen for the others which is used in the construction of $pg\,[\![Case]\!]$.

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

The construction of $pg\,[\![bx_1 \circ bx_2]\!]$ is the most important part in 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 $(s_1, v_1) \Leftarrow pg\,[\![bx_1]\!](s, dummy)$. This is possible when 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. Since there is no real view, this *dummy* is necessary to pair with the original source s to form the input of $put\,[\![bx_1]\!]$. In general, *dummy* depends on the source s, the view v and/or the program bx_1. Programmers can be required to give a way to construct *dummy*, but it may be inessential for ill-typed systems where choosing *dummy* as v is one of the easiest ways to meet our expectation. That setting is used in our experiments. The last equality changes from $(s_3, v_3) \Leftarrow pg\,[\![bx_1]\!](s, s_2)$ to $(s_3, v_3') \Leftarrow pg\,[\![bx_1]\!](s_1, s_2)$, where s and v_3 are replaced with s_1 and v_3' respectively. Because s_1 is a source update of $put\,[\![bx_1]\!]\ s\ dummy$, so under the PutPut law, it is possible to substitute s by s_1. The substitution of v_3 by v_3' is simply replacing the variable name since v_3 and v_3' hold different results of $get\,[\![bx_1]\!]\ s$ and $get\,[\![bx_1]\!]\ s_1$ respectively. Because both variables are no longer used later, this substitution does not affect the outcome of the function.

4.2 Lazy Update: cpg

When evaluating $pg\,[\![bx_1 \circ bx_2]\!]$, 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 of pg for left associative BX programs is inevitable. To solve that, we introduce a new function, cpg, accumulates updates on the source and the view. $cpg[\![bx]\!](ks, kv, s, v)$ 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 same as pg. 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:
$fst = \lambda(x_1, x_2).x_1,\ snd = \lambda(x_1, x_2).x_2,\ con = \lambda ks_1.\lambda ks_2.\lambda x.((ks_1\ x), (ks_2\ x))$

Definition 6. $cpg[\![bx]\!](ks, kv, s, v)$

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

In the places where third and/or fourth argument (s and v) are updated by applications, the computations are also accumulated in ks and/or kv. Thanks to these accumulations, there are only two cpg calls in $cpg\,[\![bx_1 \circ bx_2]\!]$. The first call $cpg\,[\![bx_1]\!]$ requires parameter $(ks, id, s, dummy)$ where s and ks are corresponding to the source and the update over source. Since there is no real view here, we need a dummy same as pg. Then the continuation updating on this dummy should be initiated as the identity function. The first cpg call is assigned to a 4-tuple (ks_1, kv_1, s_1, v_1). In the next assignment, 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 recursively 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:

$$\begin{aligned}&(ks, kv, s, v) \Leftarrow cpg[\![bx]\!](\lambda\_.s_0, id, s_0, v_0);\\ &\ \ (s, v)\end{aligned}$$

In general, the beginning of a continuation should be the 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 needs to be initiated as the constant function from that source. This constant function helps to retain the discarded things in the source.

The result pair (s, v) obtained from *cpg* as above should be same with the result of $pg\,[\![bx]\!](s_0, v_0)$. More generally, we have the following relationship:

$$cpg[\![bx]\!](ks, kv, s, v) = pg\,[\![bx]\!](ks\ s, kv\ v)$$

Note that, in $cpg\,[\![bx_1 \circ bx_2]\!]$, $\underline{s_1}$ is redundant because this evaluated variable is not used in the later steps. In the next session, we will optimize this redundancy.

4.3 Lazy Computation: kpg

The problem for *cpg* lies in redundant computations during the evaluation. To prevent such redundant computations from occurring, we introduce an extension named *kpg*. While *cpg* evaluates values eagerly, *kpg* does the opposite. Every value is evaluated lazily in a computation of *kpg*. The input of $kpg\,[\![bx]\!]$ is expanded to a 6-tuple (ks, kv, lks, lkv, s, v) where ks and kv keep the modification information same as *cpg*, s and v hold evaluated values, and lks and lkv are used for lazy evaluation of actual values. The output of this function is also a 6-tuple (ks, kv, lks, lkv, 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:

$$\begin{aligned}&(ks, kv, lks, lkv, s, v) \Leftarrow kpg[\![bx]\!](\lambda\_.s_0, id, id, id, s_0, v_0);\\ &\ \ (lks\ s, lkv\ v)\end{aligned}$$

The beginning of accumulative functions lks and lkv are set as the identity function, while ks and kv are initiated as the same with the corresponding ones in *cpg*. The relationship among *kpg*, *cpg* and *pg* can be shown as follows:

$$\begin{aligned}&kpg[\![bx]\!](ks, kv, lks, lkv, s, v) = cpg[\![bx]\!](ks \circ lks, kv \circ lkv, s, v)\\ &\qquad = pg\,[\![bx]\!](ks\ (lks\ s), kv\ (lkv\ v))\end{aligned}$$

Definition 7. $kpg[\![bx]\!](ks, kv, lks, lkv, s, v)$

$$
\begin{array}{l}
kpg[\![Skip\ h]\!](ks,kv,lks,lkv,s,v) = \\
\quad es \Leftarrow lks\ s; \quad ev \Leftarrow lkv\ v; \\
\quad \text{if } h\ es = ev \text{ then } (ks,kv,id,id,es,ev) \text{ else } \mathit{undefined} \\
kpg[\![Replace]\!](ks,kv,lks,lkv,s,v) = (kv,ks,lkv,lks,v,s) \\
kpg[\![bx_1 \times bx_2]\!](ks,kv,lks,lkv,s,v) = \\
\quad es \Leftarrow lks\ s; \quad ev \Leftarrow lkv\ v; \\
\quad (ks_1,kv_1,lks_1,lkv_1,s_1,v_1) \Leftarrow \\
\qquad\qquad kpg[\![bx_1]\!](fst \circ ks, fst \circ kv, fst, fst, es, ev); \\
\quad (ks_2,kv_2,lks_2,lkv_2,s_2,v_2) \Leftarrow \\
\qquad\qquad kpg[\![bx_2]\!](snd \circ ks, snd \circ kv, snd, snd, es, ev); \\
\qquad (\ con\ ks_1\ ks_2,\ con\ kv_1\ kv_2, \\
\qquad\quad con\ (lks_1 \circ fst)\ (lks_2 \circ snd),\ con\ (lkv_1 \circ fst)\ (lkv_2 \circ snd), \\
\qquad\quad (s_1,s_2),\ (v_1,v_2)\ ) \\
kpg[\![RearrS\ f_1\ f_2\ bx]\!](ks,kv,lks,lkv,s,v) = \\
\quad (ks',kv',lks',lkv',s',v') \Leftarrow kpg[\![bx]\!](f_1 \circ ks, kv, f_1 \circ lks, lkv, s, v); \\
\qquad (f_2 \circ ks', kv', f_2 \circ lks', lkv', s', v') \\
kpg[\![RearrV\ g_1\ g_2\ bx]\!](ks,kv,lks,lkv,s,v) = \\
\quad (ks',kv',lks',lkv',s',v') \Leftarrow kpg[\![bx]\!](ks, g_1 \circ kv, lks, g_1 \circ lkv, s, v); \\
\qquad (ks', g_2 \circ kv', lks', g_2 \circ lkv', s', v') \\
kpg[\![Case\ cond_{sv}\ cond_s\ bx_1\ bx_2]\!](ks,kv,lks,lkv,s,v) = \\
\quad es \Leftarrow lks\ s; \quad ev \Leftarrow lkv\ v; \\
\quad \text{if } cond_{sv}\ es\ ev\ \&\&\ cond_s\ es \\
\quad \text{then } (ks',kv',lks',lkv',s',v') \Leftarrow kpg[\![bx_1]\!](ks,kv,id,id,es,ev) \\
\quad \text{else } (ks',kv',lks',lkv',s',v') \Leftarrow kpg[\![bx_2]\!](ks,kv,id,id,es,ev) \\
\quad \underline{\text{fi}}\ cond_s\ (lks'\ s')\ \&\&\ cond_{sv}\ es\ (lkv'\ v');\ \text{return } (ks',kv',lks',lkv',s',v') \\
kpg[\![bx_1 \circ bx_2]\!](ks,kv,lks,lkv,s,v) = \\
\quad (ks_1,kv_1,\underline{lks_1},lkv_1,\underline{s_1},v_1) \Leftarrow kpg[\![bx_1]\!](ks,id,lks,id,s,dummy); \\
\quad (ks_2,kv_2,lks_2,lkv_2,s_2,v_2) \Leftarrow kpg[\![bx_2]\!](kv_1,kv,lkv_1,lkv,v_1,v); \\
\qquad (ks_1 \circ ks_2, kv_2, ks_1 \circ lks_2, lkv_2, s_2, v_2)
\end{array}
$$

In *kpg*, basically, functions for the updates are kept (but not evaluated) in *lks* and *lkv*. In $kpg\,[\![RearrS]\!]$ and $kpg\,[\![RearrV]\!]$, f_1 and g_1 are accumulated in *lks* and *lkv*. The kept functions are evaluated in $kpg\,[\![Skip]\!]$ and $kpg\,[\![Case]\!]$ by applications of $lks\ s$ and $lkv\ v$. At the same time, the third and fourth argument of recursive calls are updated with the identity function. This evaluation is needed because these definitions require the actual values, *es* and *ev*. Thanks to this update, accumulation in *kpg*, $\underline{lks_1}$ and $\underline{s_1}$ in $kpg\,[\![bx_1 \circ bx_2]\!]$ are not evaluated as much as possible.

Additionally we did two optimizations in *kpg*. The first is in $kpg\,[\![bx_1 \times bx_2]\!]$. Because *es* and *ev* are not used in this definition, we do not need to evaluate. However, if we accumulate *lks* and *lkv*, both might be evaluated independently in two assignments using $kpg\,[\![bx_1]\!]$ and $kpg\,[\![bx_2]\!]$. This includes the same computation. To remove duplicate evaluations, we evaluate actual values *es* and *ev* before calling $kpg\,[\![bx_1]\!]$ and $kpg\,[\![bx_2]\!]$. The second is in $kpg\,[\![Case]\!]$ and not shown in the definition. We need to evaluate $lks'\ s'$ and $lkv'\ v'$ to check the $\underline{\text{fi}}$ condition before returning the 6-tuple. Such evaluations can be done lazily to make programs run faster. We use the above small optimizations in our implementation.

4.4 Combination of pg and kpg: xpg

The purpose we introduced *cpg* and *kpg* is to avoid redundant recursive call and keep the dropped parts from the source in a function. On the other hand, these accumulations in *cpg* and *kpg* will be an overhead if they are not necessary. The problem in $pg\,[\![bx_1 \circ bx_2]\!]$ is that there are two recursive calls of $pg\,[\![bx_1]\!]$ and there is no problem in the recursive call of $pg\,[\![bx_2]\!]$. Therefore, we combine *pg* and *kpg* to take advantage of both approaches.

Definition 8. $xpg\,[\![bx]\!](s, v)$

$$
\begin{array}{l}
xpg\,[\![bx]\!](s,v) = \text{match } bx \ \text{ with} \\
\quad |\ bx_1 \circ bx_2 \rightarrow \\
\qquad (ks_1, kv_1, lks_1, lkv_1, s_1, v_1) \Leftarrow kpg[\![bx_1]\!](\lambda\_.s, id, id, id, s, dummy); \\
\qquad (s_2, v_2) \Leftarrow xpg\,[\![bx_2]\!](lkv_1\ v_1, v); \\
\qquad\quad (ks_1\ s_2, v_2) \\
\quad |\ \_ \rightarrow \textit{similar to pg}
\end{array}
$$

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*, while the second call is based on *pg*.

5 Experiments

We have fully implemented and tested all methods[5,6] described in the previous sections. Our target language is untyped. Some dummies used for *pg*, *cpg*, *kpg* and *xpg* are replaced with the current updated views. This helps a program in the *put* direction valid.

5.1 Test Cases

We have selected seven test cases (Table 1) to represent non-trivial cases of practical significance. The test cases use left associative compositions because we focus on this kind of inefficiency in this paper. In the last two columns, s and v are the updated source and view, respectively. They are produced by applying *put* and *get* to the original source s_0 and view v_0. That is, $s = put\,[\![bx]\!]\ s_0\ v_0$ and $v = get\,[\![bx]\!]\ s_0$, where bx is the program indicated in the second column of the table. Results s and v are independent of the associativity of the composition.

[5] All experiments on macOS 10.14.6, processor Intel Core i7 (2.6 GHz), RAM 16 GB 2400 MHz DDR4, OCaml 4.07.1. The OCaml runtime system options and garbage collection parameters are set as default.

[6] The implementation is available: https://github.com/k-tsushima/pgs

Table 1. Composition test cases (number of compositions = n)

No	Name	Type	Input		Output	
			s_0	v_0	s	v
1	lcomp-phead-ldata	straight line	$\underbrace{[[\ldots[1]\ldots]]}_{\text{n+1 times}}$	100	$\underbrace{[[\ldots[100]\ldots]]}_{\text{n+1 times}}$	1
2	lcomp-ptail	straight line	[1,. . .,n+1]	[1,. . .,10]	s_0 @ v_0	[ ]
3	lcomp-ptail-ldata	straight line	$\underbrace{[L,\ldots,L]}_{\text{n+1 times}}$	$\underbrace{[L,\ldots,L]}_{\text{10 times}}$	$\underbrace{[L,\ldots,L]}_{\text{n+11 times}}$	[ ]
4	lcomp-bsnoc	straight line	[1,. . .,n-1]	[1,. . .,n-1]	[1,. . .,n-1]	[1,. . .,n-1]
5	lcomp-bsnoc-ldata	straight line	$\underbrace{[L,\ldots,L]}_{\text{n-1 times}}$	$\underbrace{[L,\ldots,L]}_{\text{n-1 times}}$	$\underbrace{[L,\ldots,L]}_{\text{n-1 times}}$	$\underbrace{[L,\ldots,L]]}_{\text{n-1 times}}$
6	breverse	recursion	[1,. . .,n]	[1,. . .,n]	[n,. . .,1]	[n,. . .,1]
7	breverse-ldata	recursion	$\underbrace{[L,\ldots,L]}_{\text{n times}}$	$\underbrace{[L,\ldots,L]}_{\text{n times}}$	$\underbrace{[L,\ldots,L]}_{\text{n times}}$	$\underbrace{[L,\ldots,L]}_{\text{n times}}$

The first five test cases (1–5) are n straight-line (non-recursive) compositions of the same $n+1$ programs. The prefix *lcomp* in the name of a test case indicates that the textual compositions are left associative. The suffix *ldata* indicates that the input size is considered large. The symbol L in the input column stands for a list $L = [T, \ldots, T]$ with $T = [A, \ldots, A]$ of length 10 and $A = [1, \ldots, 5]$. They are only intended to generate test data that is large enough for measuring results.

We introduced the composition $phead \circ phead$ earlier in Section 1. The composition of many *phead*s works similarly. The head of a head element inside a deeply nested list, which is the source, is updated by the changed view. Because of the type of the source, this program is categorized as a *ldata* case.

Next, we briefly explain the behavior of the remaining compositions in Table 1.

The composition of many *ptail*s, in the *put* direction, replaces a part of the tail of the source list by the view list and, in the *get* direction, returns such a tail from the source.

The composition of many *bsnoc*s, in the *put* direction, creates a permutation of the view list if its length is not larger than the length of the source list and, in the *get* direction, produces a permutation of the source list.

breverse is defined in terms of *bfoldr*, that appeared in Section 2. In the *put* direction, it produces a reverse of the view list if its length is not larger than the length of the source list and, in the *get* direction, produces a reverse of the source list. Note that compositions are by the recursions of *breverse* and the number of compositions are dynamically determined by the length of the source list.

5.2 Results

Figure 4 shows the evaluation times for each of the seven test cases using the three methods: *put* in minBiGUL, put_m in minBiGUL$_m$ and *xpg*. We also did similar experiments with *pg*, *cpg* and *kpg*, but their results are slower than the corresponding ones of *xpg*. The slowness was caused by the exponential number

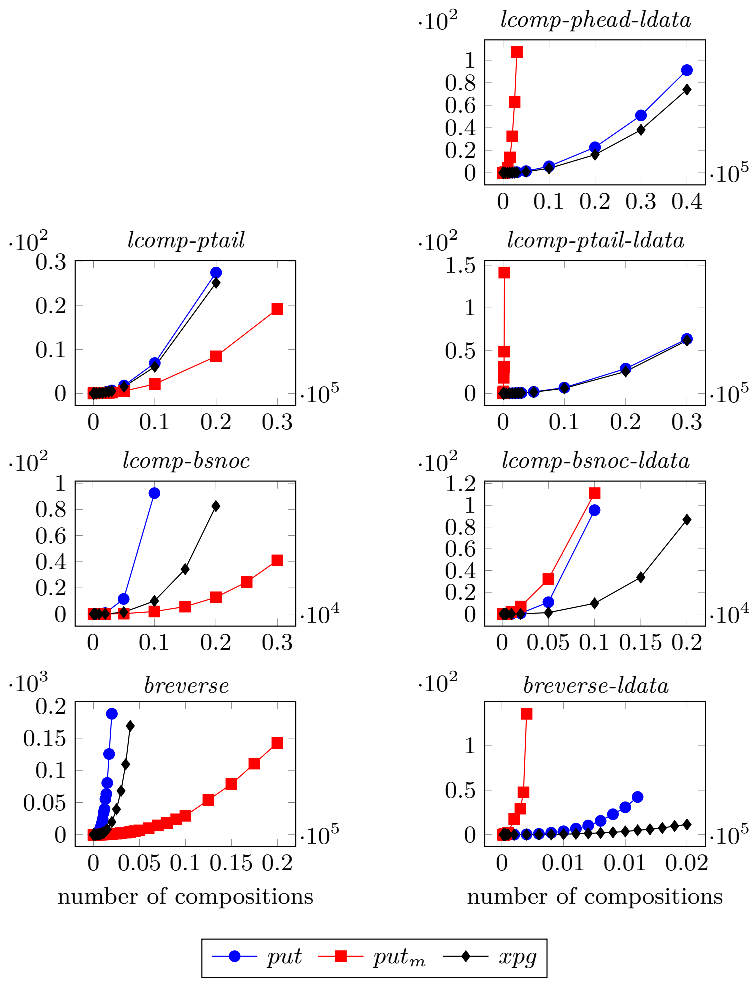

Fig. 4. Evaluation time (secs) against number of compositions

of pg calls in the case of pg, redundant evaluations (cpg), and redundant overhead for constructing closures (kpg). Therefore we simply omit these results in Figure 4. As we all know, put works poorly for left associative compositions because of the number of reevaluated gets. The left part of the figure contains tests using not-large inputs, and we see that put_m is the fastest method for them. However if the input size is large enough, in the cases of the right part of the figure, put_m will be slower quickly due to time for manipulating data in the table. At that time, xpg is the most effective method.

Note that this result concerns BX programs that use many compositions. If the number of compositions is small, the original put without memoization will be fastest because of the overhead for memoization (put_m) and the overhead for keeping complements in closures (xpg).

6 Related Work

Since the pioneering work of lens [5], many BX languages have been proposed [2, 3, 6–11]. Although much progress has been made on the semantics and correctness of BX programs for the past years, as far as we are aware, little work has been done on optimization of BX programs [12]. Anjorin et. al. introduces the first benchmark[7] for BX languages and compared them [13], but a systematic improvement for practical implementation of BX languages is still missing. This paper shows the first attempt of improving efficiency of BX composition evaluation.

The baseline of this work is the BX language BiGUL [7, 14], and we compare BiGUL's method (in Section 2) with our methods (in Sections 3 and 4). From experimental results of left associative BX composition programs, we can see that our memoization methods are faster than the original BiGUL's evaluation method. While we focus on BiGUL, our methods are general and should be applicable to other BX languages.

Our work is related to many known optimization methods for unidirectional programs. Memoization [15, 16] is a technique to avoid repeated redundant computation. In our case, we show that two specific memoization methods can be used for bidirectional programs. To deal with inefficiency due to compositions, many fusion methods have been studied [17] to merge a composition of two (recursive) programs into one. However, under the context of bidirectional programs, we need to consider not only compositions of recursive programs but also compositions inside a recursive program (as we have seen in *bfoldr*). This paper focuses on the composition inside a recursion, where compositions are produced dynamically at runtime. We tackled the problem by using tupling [18], lazy update and lazy computation [19, 20].

7 Conclusion and Future Work

In this paper, we focus on efficiency of composition of BX programs. To achieve fast evaluation, we introduce two different methods using memoization. From the experimental results of left associative BX composition programs, we know that xpg is fastest method if input size is large and put_m is fastest for other left associative programs. This shows that if programmers choose one method based on their BX programs and inputs, they can use an efficient evaluation method.

We will continue our work on the following four points. First is overcoming our limitation in xpg. In xpg, there are two limitations: only for very-well-behaved programs and restriction for $pg\,[\![Case]\!]$ in Section 4.1. To be a practical

[7] The BX programs in their benchmark are basically BX programs without composition. Because we focus on the BX programs that include many compositions, their benchmark is not applicable for our evaluation.

evaluation method, an extension of the target language is needed. After extension of the language, we can evaluate more various programs for experimental results. Second is introducing an automatic analysis about BX programs and inputs to choose the best evaluation method. Currently, programmers have to choose the evaluation method by themselves based on their BX programs and inputs. If this analysis is achieved, we can reduce programmers' burden. Third is introducing a type system to our target language, especially the datatype of sources and view. If we introduce this, we can avoid runtime errors like statically-typed functional languages. For this, we need to investigate more about how to construct dummy values because the current definition used in our experiments will cause type errors. Fourth is using a lazy language. Although we used a strict language OCaml in this paper, if we use a lazy language, we might get laziness for free.

Acknowledgment We thank the anonymous reviewers for valuable comments and suggestions. This work has been partially supported by JSPS KAKENHI Grant Number JP17H06099 and ROIS NII Open Collaborative Research 2018.

References

1. Bancilhon, F. & Spyratos, N. Update Semantics of Relational Views. *ACM Trans. Database Syst.* **6,** 557–575. ISSN: 0362-5915 (Dec. 1981).
2. Bohannon, A., Pierce, B. C. & Vaughan, J. A. *Relational Lenses: A Language for Updatable Views* in *Principles of Database Systems* (2006), 42–67.
3. Bohannon, A., Foster, J. N., Pierce, B. C., Pilkiewicz, A. & Schmitt, A. *Boomerang: Resourceful Lenses for String Data* in *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (ACM, San Francisco, California, USA, 2008), 407–419. ISBN: 978-1-59593-689-9.
4. Pacheco, H., Hu, Z. & Fischer, S. *Monadic Combinators for "Putback" Style Bidirectional Programming* in *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation* (ACM, San Diego, California, USA, 2014), 39–50. ISBN: 978-1-4503-2619-3.
5. Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C. & Schmitt, A. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem. *ACM Trans. Program. Lang. Syst.* **29.** ISSN: 0164-0925 (May 2007).
6. Buchmann, T. *BXtend - A Framework for (Bidirectional) Incremental Model Transformations* in *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development* (SCITEPRESS - Science and Technology Publications, Lda, Funchal, Madeira, Portugal, 2018), 336–345. ISBN: 978-989-758-283-7.

7. Ko, H.-S., Zan, T. & Hu, Z. *BiGUL: A Formally Verified Core Language for Putback-based Bidirectional Programming* in *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (ACM, St. Petersburg, FL, USA, 2016), 61–72. ISBN: 978-1-4503-4097-7.
8. Leblebici, E., Anjorin, A. & Schürr, A. *Developing eMoflon with eMoflon* in *Theory and Practice of Model Transformations* (eds Di Ruscio, D. & Varró, D.) (Springer International Publishing, Cham, 2014), 138–145. ISBN: 978-3-319-08789-4.
9. Samimi-Dehkordi, L., Zamani, B. & Rahimi, S. K. EVL+Strace: a novel bidirectional model transformation approach. *Information and Software Technology* **100,** 47–72. ISSN: 0950-5849 (2018).
10. Cicchetti, A., Di Ruscio, D., Eramo, R. & Pierantonio, A. *JTL: A Bidirectional and Change Propagating Transformation Language* in *Software Language Engineering* (eds Malloy, B., Staab, S. & van den Brand, M.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2011), 183–202. ISBN: 978-3-642-19440-5.
11. Hinkel, G. & Burger, E. Change Propagation and Bidirectionality in Internal Transformation DSLs. *Softw. Syst. Model.* **18,** 249–278. ISSN: 1619-1366 (Feb. 2019).
12. Horn, R., Perera, R. & Cheney, J. Incremental Relational Lenses. *Proc. ACM Program. Lang.* **2,** 74:1–74:30. ISSN: 2475-1421 (July 2018).
13. Anjorin, A. *et al.* Benchmarking bidirectional transformations: theory, implementation, application, and assessment. *Software and Systems Modeling.* ISSN: 1619-1374 (Sept. 2019).
14. Ko, H.-S. & Hu, Z. An Axiomatic Basis for Bidirectional Programming. *Proc. ACM Program. Lang.* **2,** 41:1–41:29. ISSN: 2475-1421 (Dec. 2017).
15. Bellman, R. E. *Dynamic Programming* ISBN: 0486428095 (Dover Publications, Inc., New York, NY, USA, 2003).
16. Michie, D. "Memo" Functions and Machine Learning. *Nature* **218** (1968).
17. Wadler, P. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.* **73,** 231–248. ISSN: 0304-3975 (Jan. 1988).
18. Fokkinga, M. M. *Tupling and Mutumorphisms* Appeared in: The Squigollist, Vol 1, Nr 4, 1990, pages 81–82. June 1990.
19. Henderson, P. & Morris Jr., J. H. *A Lazy Evaluator* in *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages* (ACM, Atlanta, Georgia, 1976), 95–103.
20. Hudak, P., Hughes, J., Peyton Jones, S. & Wadler, P. *A History of Haskell: Being Lazy with Class* in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (ACM, San Diego, California, 2007), 12–1–12–55. ISBN: 978-1-59593-766-7.