15th International Symposium on Functional and Logic Programming

Sep 14-16, 2020

An Efficient Composition of Bidirectional Programs by Memoization and Lazy Update

Kanae Tsushima [1,2], Bach Nguyen Trong [1,2], Robert Glück [3], Zhenjiang Hu [4]

[1] National Institute of Informatics, Tokyo, Japan
[2] The Graduate University for Advanced Studies, SOKENDAI, Kanagawa, Japan
[3] University of Copenhagen, Copenhagen, Denmark
[4] Peking University, Beijing, China

Presenter: Bach Nguyen Trong

Bidirectional Transformation

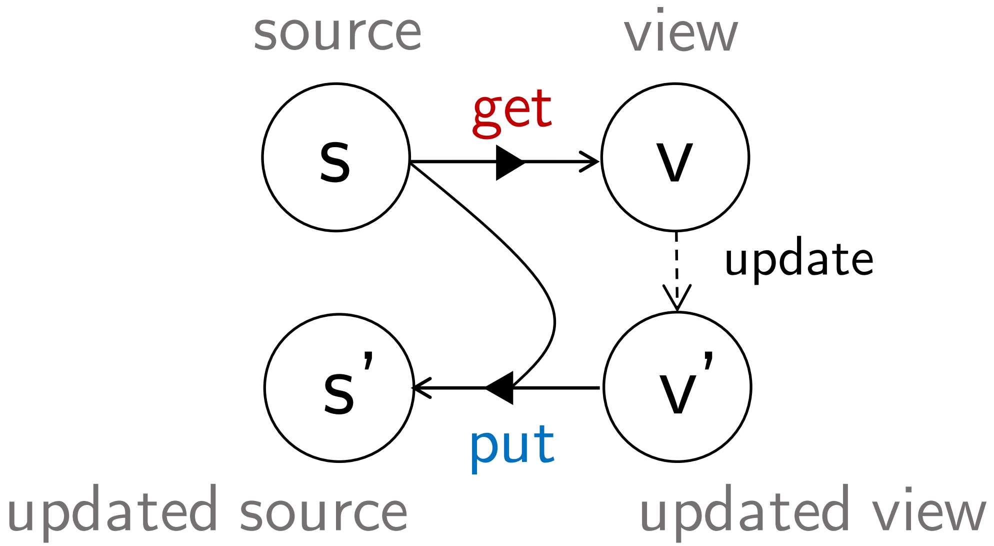
- Bidirectional transformation (BX):
  - a means to synchronize – or maintain consistency – between multiple representations of related and often overlapping information
  - when a representation is modified, the others may need updating to restore the consistency

- Applications:
  - databases (eg. the view update problem, ...)
  - user interface design (eg. synchronizing between different graphical layouts, ...)
  - model-driven development (eg. synchronizing between UML and source code, ...)
  - ...

Bidirectional Transformation

- BX comprises 2 transformations: forward and backward transformations

  [get] [put]

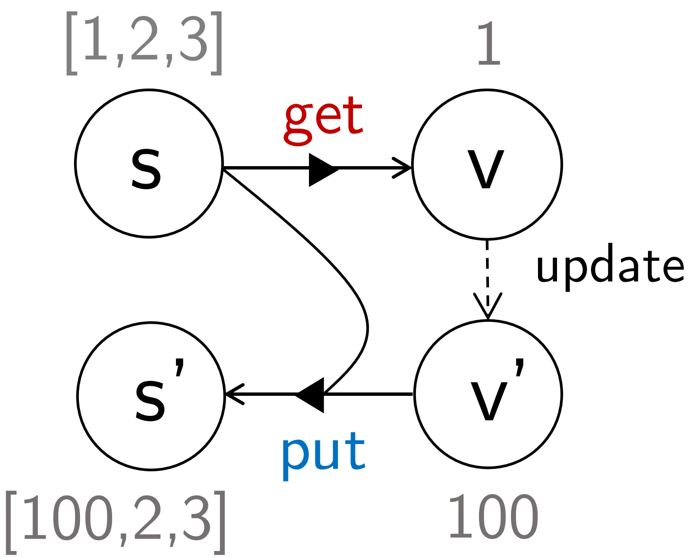


BX : Source Domain $\leftrightarrow$ View Domain

[*] Foster et al. 2007, *Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem*, TOPLAS

Example: phead

$$\text{phead: [Int]} \leftrightarrow \text{Int} \quad \begin{cases} \text{get}_{\text{phead}}\ \text{s} = \text{head s} \\ \text{put}_{\text{phead}}\ \text{s v'} = \text{v' :: tail s} \end{cases}$$



$$\text{get}_{\text{phead}}\ [1,2,3] = 1$$
$$\text{put}_{\text{phead}}\ [1,2,3]\ 100 = [100,2,3]$$

Well-behaveness [*]

A BX is *well-behaved* if get and put obey GetPut and PutGet

GetPut $\quad \text{put s (get s)} = \text{s}$

[no change to the view should be reflected as no change in the source]

PutGet $\quad \text{get (put s v')} = \text{v'}$

[the updated view can be recovered by applying get to the updated source]

[*] Foster et al. 2007, *Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem*, TOPLAS

Research on Bidirectional Transformation

- Semantics and correctness have been investigated intensively during the past years
  - Bohannon et al., *Relational Lenses: A Language for Updatable Views*, PODS'06
  - Bohannon et al., *Boomerang: Resourceful Lenses for String Data*, POPL'08
  - Cicchetti et al., *JTL: A Bidirectional and Change Propagating Transformation Language*, SLE'10
  - Leblebici et al., *Developing eMoflon with eMoflon*, ICMT'14
  - Ko et al., BiGUL: *A Formally Verified Core Language for Putback-based Bidirectional Progr.*, PEPM'16
  - Ko et al., *An Axiomatic Basis for Bidirectional Programming*, POPL'18
  - Van-Dang et al., *Programmable View Update Strategies on Relations*, VLDB'20

- Efficiency and optimization have not yet been fully understood
  - Horn et al., *Incremental Relational Lenses*, ICFP'18

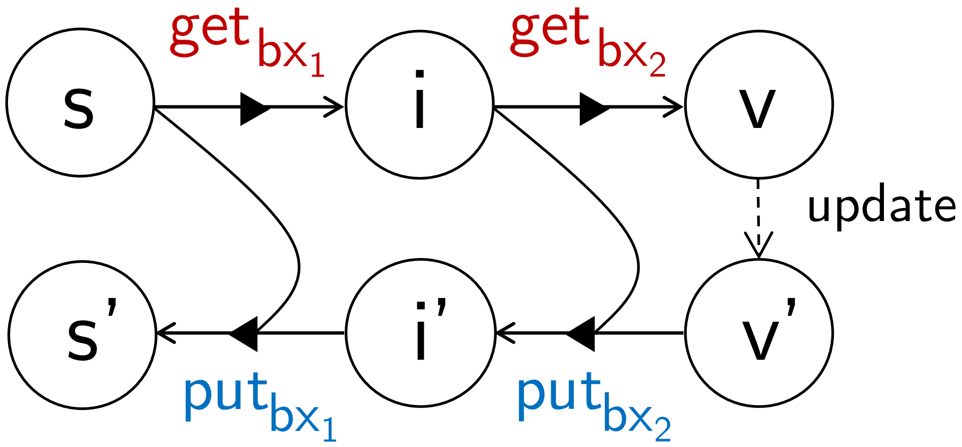
This talk: an *efficient* **composition** of bidirectional programs

Composition of BXs

- Given bx_1 and bx_2 are BXs.
- Composition $\mathbf{bx_1 \;\tilde{\circ}\; bx_2}$ is defined by:

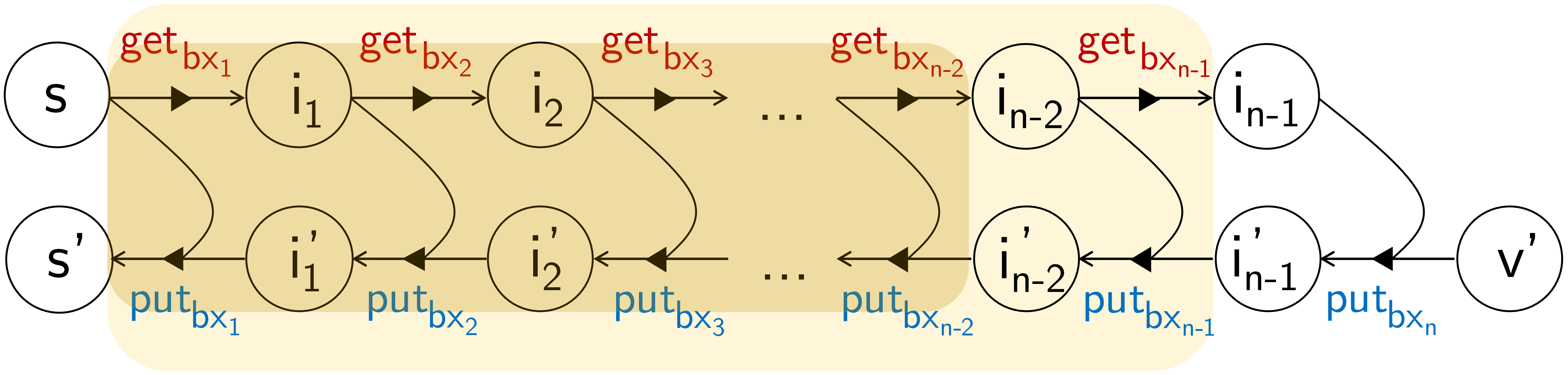
$$
\begin{aligned}
get_{bx_1 \,\tilde{\circ}\, bx_2}\; s &= get_{bx_2}\,(get_{bx_1}\; s) \\
put_{bx_1 \,\tilde{\circ}\, bx_2}\; s\; v' &= put_{bx_1}\; s\,(put_{bx_2}\,(get_{bx_1}\; s)\; v')
\end{aligned}
$$

[unlike traditional function compositions, a composition of BXs is read left-to-right]



Inefficiency Issue of Left-associative Comp.

$$\text{left\_bx\_n} = ((\ldots((bx_1 \;\tilde{o}\; bx_2) \;\tilde{o}\; bx_3) \;\ldots\; \tilde{o}\; bx_{n-2}) \;\tilde{o}\; bx_{n-1}) \;\tilde{o}\; bx_n$$



Evaluating put$_{\text{left_bx_n}}$ requires reevaluating same get$_{bx_i}$ several times

for getting intermediate results

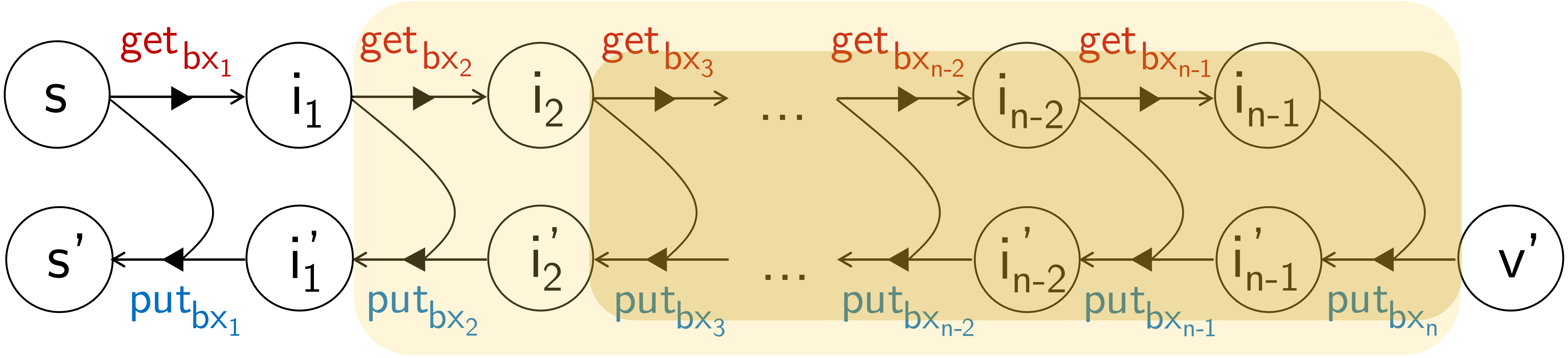
$\boxed{O(n^2)\ \text{get}_{bx_i}}$

Naive Solution

- Change associativity in composition (if #comp. is fixed)

$$\text{left\_bx\_n} = ((\ldots((bx_1 \tilde{o}\ bx_2)\ \tilde{o}\ bx_3)\ \ldots\ \tilde{o}\ bx_{n-2})\ \tilde{o}\ bx_{n-1})\ \tilde{o}\ bx_n$$

=> $$\text{right\_bx\_n} = bx_1\ \tilde{o}\ (bx_2\ \tilde{o}\ (bx_3\ \tilde{o}\ \ldots\ (bx_{n-2}\ \tilde{o}\ (bx_{n-1}\ \tilde{o}\ bx_n))\ldots))$$



Evaluating $put_{right\_bx\_n}$ requires no reevaluation of same get_{bx_i}

$\boxed{O(n)\ get_{bx_i}}$

Limitation

- Not always possible to transform from a left-associative comp. to a right-associative comp.

  - Eg: bfoldr (bidirectional version of foldr)

    bfoldr bf … = … (… bfoldr …) õ bf …

    bfoldr is inherently left-associative comp.

    ```
    foldr :: (a → b → b) → b → [a] → b
    foldr f e [ ] = e
    foldr f e (x : xs) = f x (foldr f e xs)
    ```

  - Eg: breverse: [Int] ↔ [Int] (bidirectional version of reverse)

    breverse … = … bfoldr bsnoc …

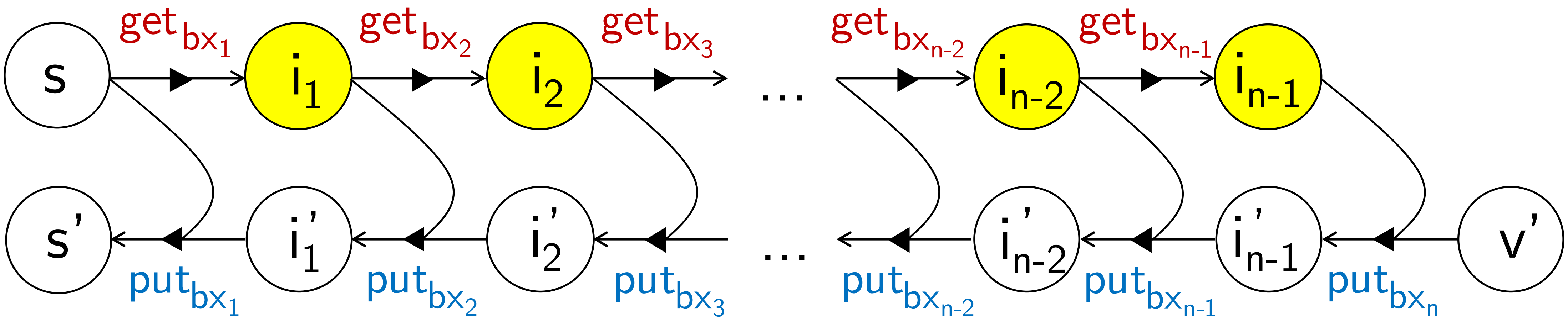
    ```
    reverse = foldr snoc [ ]
    ```

Our Work

- Propose 2 solutions to avoid redundant reevaluation by
  - S1: Memoization
  - S2: Tupling + Lazy update

Solution 1: Memoization

- Save intermediate results when evaluating a comp. in a key-value table:
  - key = (bx, s)
  - value = get_{bx} s



- Require times for manipulating (inserting, searching, …) data in the table

$$\boxed{O(n)\, get_{bx_i} + \text{Cost(manipulating data in table)}}$$

Solution 2: Tupling + Lazy Update

- Tupling put and get then evaluating them at the same time possibly avoid recomputing

$\boxed{\text{Tupling}}$

$$pg_{bx}\ (s\ ,\ v') = (put_{bx}\ s\ v'\ ,\ get_{bx}\ s)$$

$$(s'\ ,\ v) \Leftarrow pg_{bx}\ (s\ ,\ v')$$

$\boxed{\text{Tupling + Lazy update}}$

$$(ks'\ ,\ kv\ ,\ s'\ ,\ v) \Leftarrow cpg_{bx}\ (ks\ ,\ kv'\ ,\ s\ ,\ v')$$

[ks, kv, ks', kv' are continuations holding modified info. on s, v, s', v' resp.]

Tupling : pg

$$pg_{bx}\ (s\ ,\ v') = (put_{bx}\ s\ v'\ ,\ get_{bx}\ s)$$

[⇓] definitions + restrictions

$$\begin{aligned} pg_{bx1\ \tilde{o}\ bx2}\ (s\ ,\ v') = \\ (c\ ,\ i) &\Leftarrow pg_{bx1}\ (s,\ d) \\ (i'\ ,\ v) &\Leftarrow pg_{bx2}\ (i\ ,\ v') \\ (s'\ ,\ d') &\Leftarrow pg_{bx1}\ (c\ ,\ i') \\ (s'\ ,\ v) \end{aligned}$$

Using pg to evaluate $(\ldots((bx_1\ \tilde{o}\ bx_2)\ \tilde{o}\ bx_3)\ldots\ \tilde{o}\ bx_{n-1})\ \tilde{o}\ bx_n$ requires:

$\boxed{O(2^n)\ pg + \text{Cost(keeping complements c)}}$

Tupling + Lazy Update : cpg

$(s' , v) \Leftarrow pg_{bx} (s , v')$

$pg_{bx1\ \tilde{o}\ bx2} (s , v') =$
$\quad (c , i) \Leftarrow pg_{bx1} (s, d)$
$\quad (i' , v) \Leftarrow pg_{bx2} (i , v')$
$\quad (s' , d') \Leftarrow pg_{bx1} (c , i')$
$\quad\quad (s' , v)$

2 pg_{bx1} + 1 pg_{bx2}

$\downarrow$

$O(2^n)$ pg
\+ Cost(keeping complements)

$(ks' , kv , s' , v) \Leftarrow cpg_{bx} (ks , kv' , s , v')$

$cpg_{bx1\ \tilde{o}\ bx2} (ks , kv', s , v') =$
$\quad (kc , ki , c , i) \Leftarrow cpg_{bx1} (ks , id , s , d)$
$\quad (ki' , kv , i', v) \Leftarrow cpg_{bx2} (ki , kv' , i , v')$
$\quad\quad (kc \circ ki' , kv , kc\ i' , v)$

1 cpg_{bx1} + 1 cpg_{bx2} + 1 func. app.

$\downarrow$

$O(n)$ cpg
\+ Cost(manipulating data)

Tupling + Lazy Updates + Other Optimizations: xpg

$$
\begin{aligned}
&\mathrm{cpg}_{bx1\ \tilde{o}\ bx2}\ (ks\ ,\ kv',\ s\ ,\ v') = \\
&\quad (kc\ ,\ ki\ ,\ c\ ,\ i) \Leftarrow \mathrm{cpg}_{bx1}\ (ks\ ,\ id\ ,\ s\ ,\ d) \\
&\quad (ki'\ ,\ kv\ ,\ i',\ v) \Leftarrow \mathrm{cpg}_{bx2}\ (ki\ ,\ kv'\ ,\ i\ ,\ v') \\
&\qquad (kc\ o\ ki'\ ,\ kv\ ,\ kc\ i'\ ,\ v)
\end{aligned}
$$

$O(n)$ cpg + Cost(manipulating data)

reduced by doing
lazy evaluation + additional optimizations

The last optimized evaluation function: xpg

Experiment

- Target language: *core* bidirectional language: minBiGUL
  - a very-well-behaved subset of BiGUL [*]
  - untyped
- OCaml 4.07.1
- MacOS 10.14.6, Intel Core i7 (2.6 GHz), RAM 16 GiB 2400 MHz DDR4

[*] Ko et al., BiGUL: *A Formally Verified Core Language for Putback-based Bidirectional Programming*, PEPM'16

Results

most effective method when evaluating backward trans. of left-associative comp.?

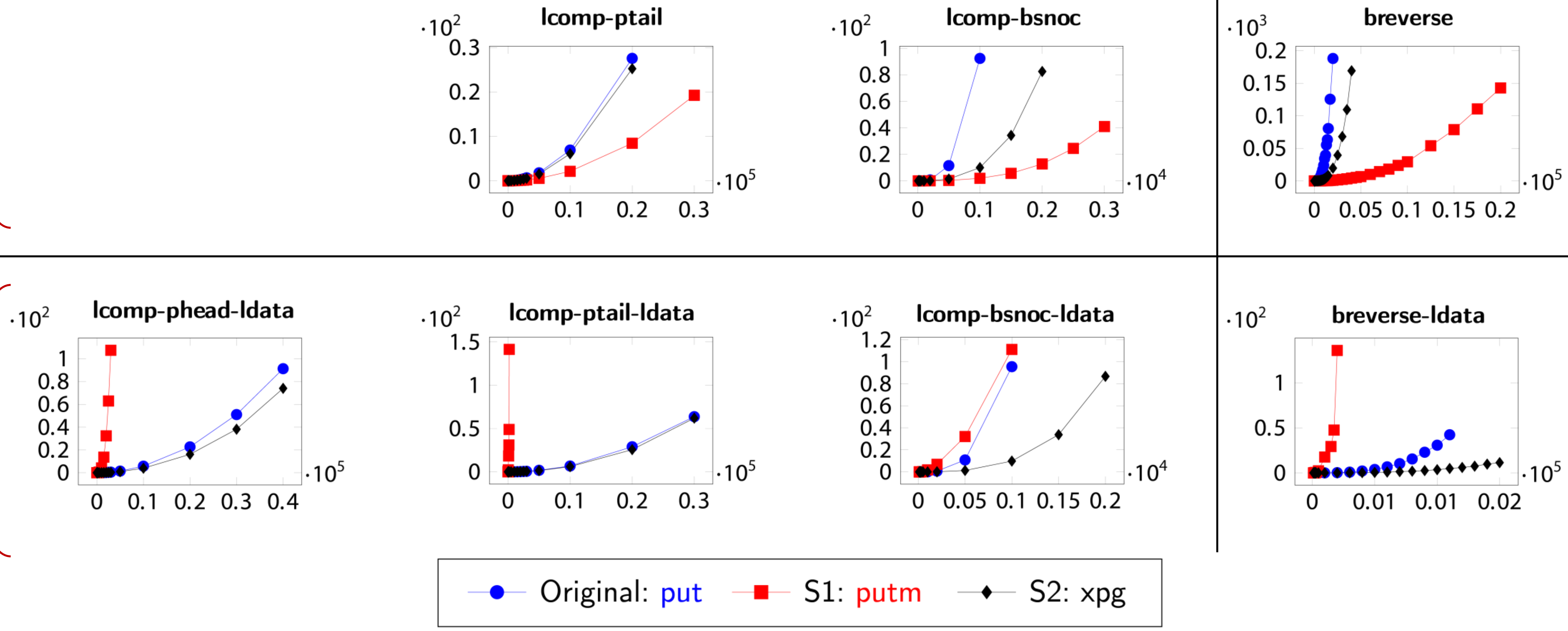
- small input: S1: putm
- large input: S2: xpg

straight line: $(\ldots((bx_1 \; \tilde{o} \; bx_2) \; \tilde{o} \; bx_3) \; \ldots \; \tilde{o} \; bx_{n-1}) \; \tilde{o} \; bx_n$

recursive comp.

small input

large input



evaluation time (secs) against #comp

Summary

- Inefficiency issue:
  - evaluating put of a left-assoc. comp. requires to reevaluating same gets
- Naive solution:
  - transforming from left-assoc. comp. to right-assoc. comp.
  - be not always possible
- Main work:
  - optimize evaluation of the backward transformation of left-assoc. comp. using memoization and lazy update

Future Work

- Introduce an automatic analysis about BX programs and inputs to choose best evaluation method
- Overcome current restrictions
- Use lazy language to get laziness for free

Any Questions?