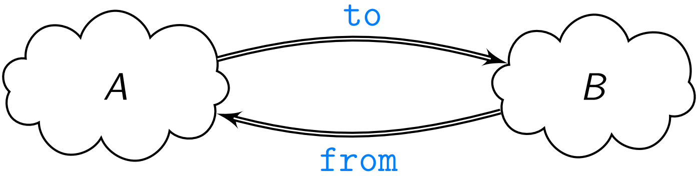
# Programming Language Approaches to Bidirectional Transformation

Janis Voigtländer

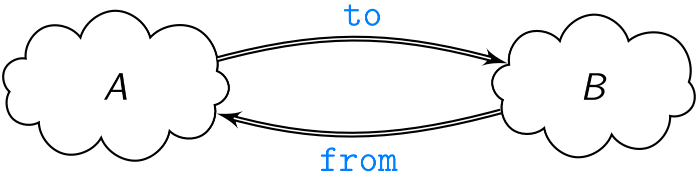
University of Bonn

LDTA'12

# Bidirectional Transformations (BX)

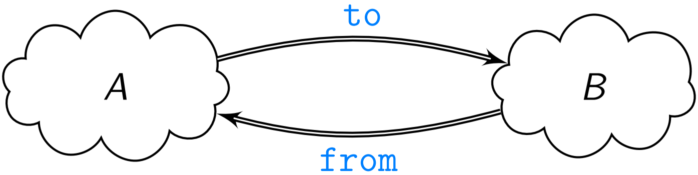


# Bidirectional Transformations (BX)



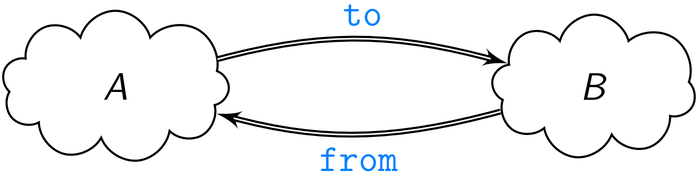

concrete syntax $\Leftrightarrow$ abstract syntax

# Bidirectional Transformations (BX)



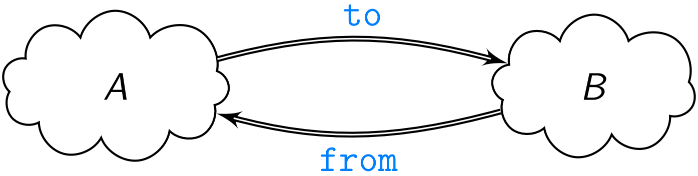

database source $\Leftrightarrow$ materialized view

# Bidirectional Transformations (BX)



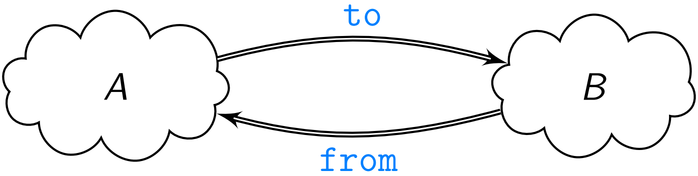

document representation $\Leftrightarrow$ screen visualization

# Bidirectional Transformations (BX)



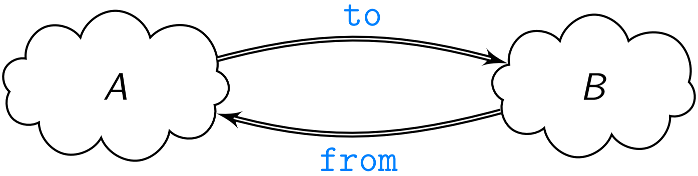

software model ⇔ code

# Bidirectional Transformations (BX)



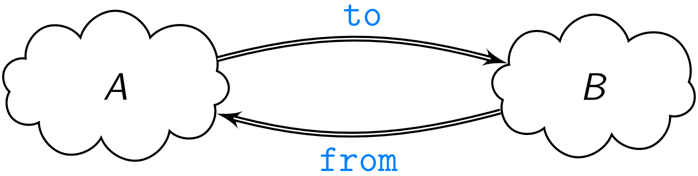

abstract datatype $\Leftrightarrow$ actual implementation

# Bidirectional Transformations (BX)



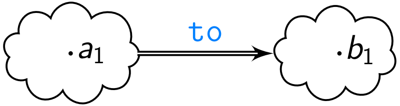

program input $\Leftrightarrow$ program output

# Bidirectional Transformations (BX)

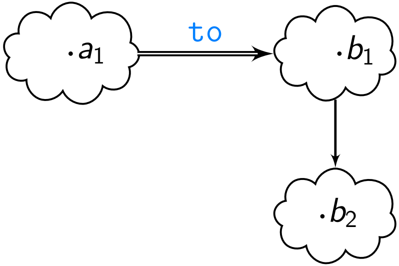



| | | |
|---|---|---|
| concrete syntax | $\Leftrightarrow$ | abstract syntax |
| database source | $\Leftrightarrow$ | materialized view |
| document representation | $\Leftrightarrow$ | screen visualization |
| software model | $\Leftrightarrow$ | code |
| abstract datatype | $\Leftrightarrow$ | actual implementation |
| program input | $\Leftrightarrow$ | program output |

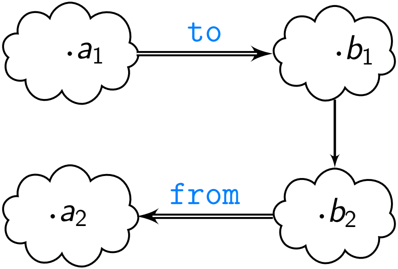
# Bidirectional Transformations



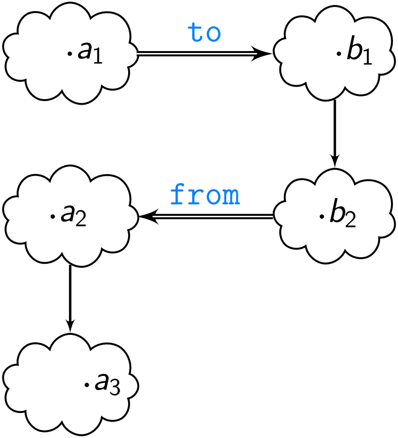
# Bidirectional Transformations



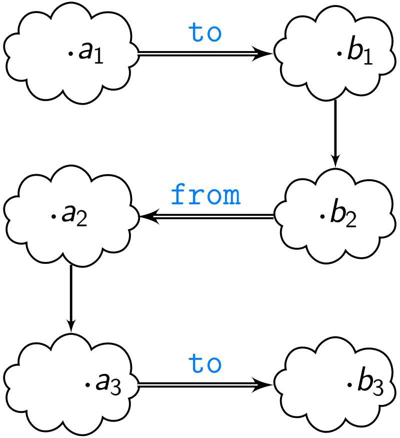
# Bidirectional Transformations



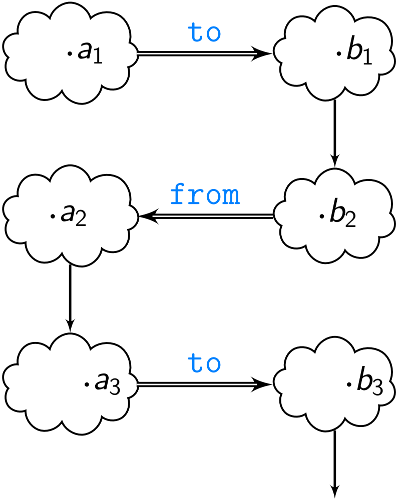
# Bidirectional Transformations



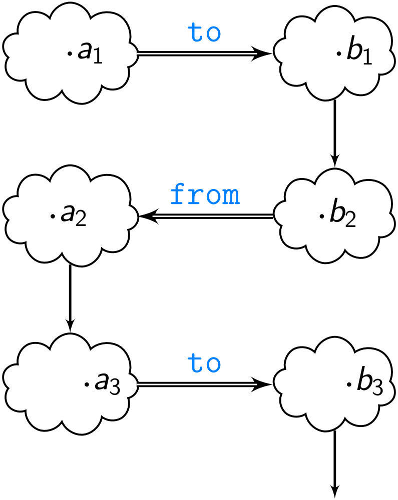
# Bidirectional Transformations



# Bidirectional Transformations

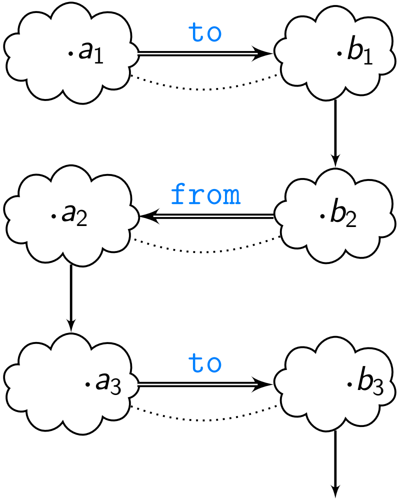


# Bidirectional Transformations




unless bijective, typically additional information needed/useful

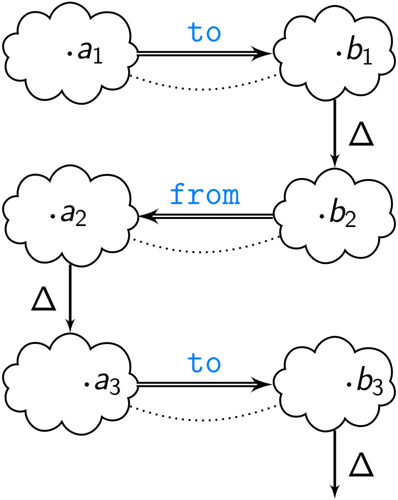
# Bidirectional Transformations




unless bijective, typically additional information needed/useful:

- about connections between $A$ and $B$ (objects)

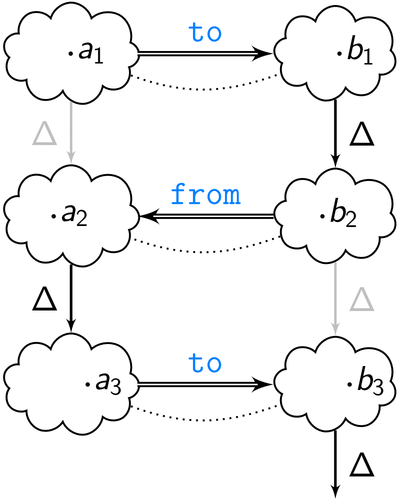
# Bidirectional Transformations




unless bijective, typically additional information needed/useful:

- about connections between $A$ and $B$ (objects)
- about the updates on either side

# Bidirectional Transformations



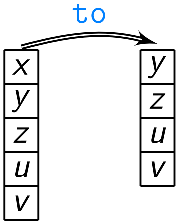

unless bijective, typically additional information needed/useful:

- about connections between $A$ and $B$ (objects)
- about the updates on either side

# Bidirectional Transformations

A closer look at representing $a_i$ $b_i$ connections.

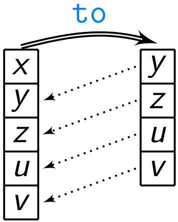
For example:



# Bidirectional Transformations

A closer look at representing $a_i$ $b_i$ connections.

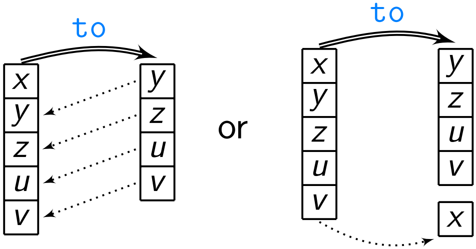
For example:



# Bidirectional Transformations

A closer look at representing $a_i$ $b_i$ connections.

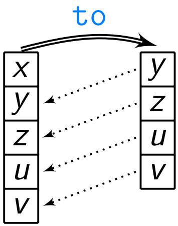
For example:



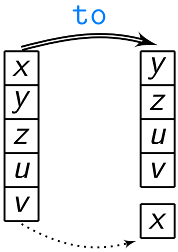
# Bidirectional Transformations

A closer look at representing $a_i \cdots b_i$ connections.

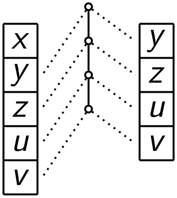
For example:




or



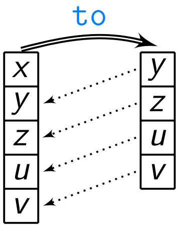

or



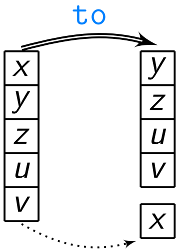
# Bidirectional Transformations

A closer look at representing $a_i$ $b_i$ connections.

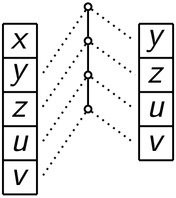
For example:



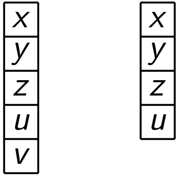

or




or



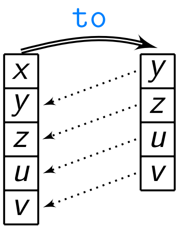

Why is it not enough to look just at the data?



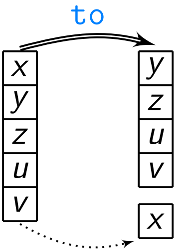
# Bidirectional Transformations

A closer look at representing $a_i$ $b_i$ connections.

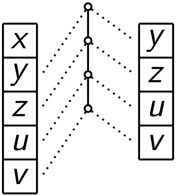
For example:



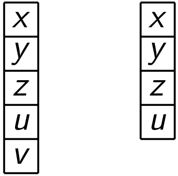

or



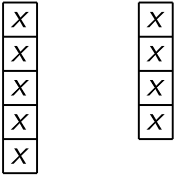

or




Why is it not enough to look just at the data?




Because of:



# Bidirectional Transformations

Some further relevant aspects:

- What artefacts need to be specified?
  - both `to` and `from`
  - only one of them, the other derived
  - a more abstract artefact, from which both derivable

# Bidirectional Transformations

Some further relevant aspects:

- What artefacts need to be specified?
  - both `to` and `from`
  - only one of them, the other derived
  - a more abstract artefact, from which both derivable
- How are they specified, manipulated, analyzed?

# Bidirectional Transformations

Some further relevant aspects:

- What artefacts need to be specified?
  - both `to` and `from`
  - only one of them, the other derived
  - a more abstract artefact, from which both derivable
- How are they specified, manipulated, analyzed?
- What properties are they expected to have?

## Bidirectional Transformations

Some further relevant aspects:

- What artefacts need to be specified?
  - both `to` and `from`
  - only one of them, the other derived
  - a more abstract artefact, from which both derivable
- How are they specified, manipulated, analyzed?
- What properties are they expected to have?
- What influence does a user, modeller, programmer have?

# Bidirectional Transformations

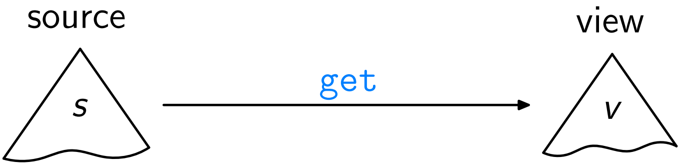
Some further relevant aspects:

- What artefacts need to be specified?
  - both `to` and `from`
  - only one of them, the other derived
  - a more abstract artefact, from which both derivable
- How are they specified, manipulated, analyzed?
- What properties are they expected to have?
- What influence does a user, modeller, programmer have?

answers/approaches vary with field

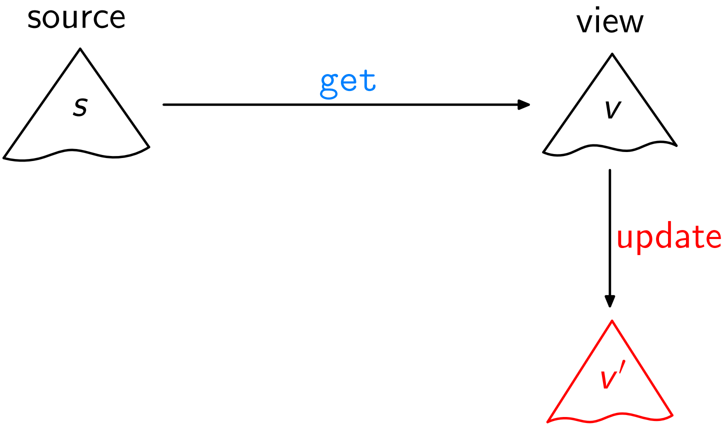
# Bidirectional Transformations

A specific (asymmetric) setting:



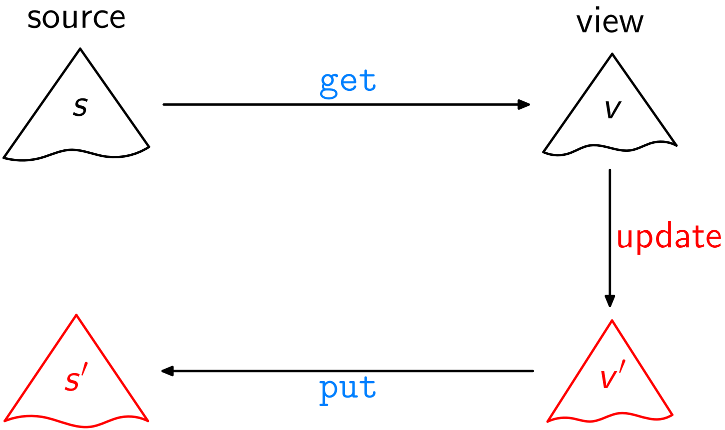
# Bidirectional Transformations

A specific (asymmetric) setting:



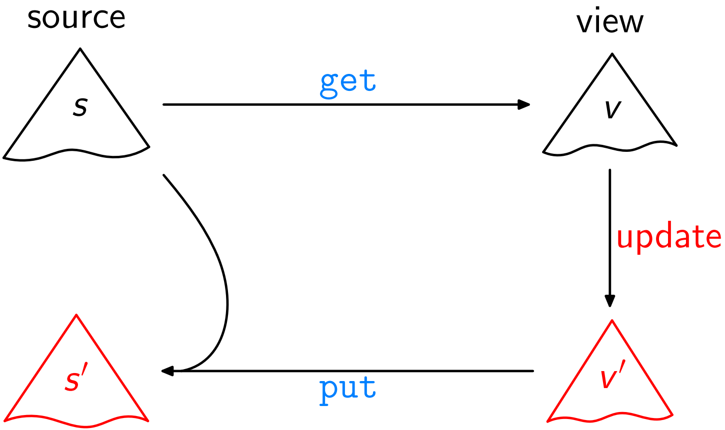
# Bidirectional Transformations

A specific (asymmetric) setting:



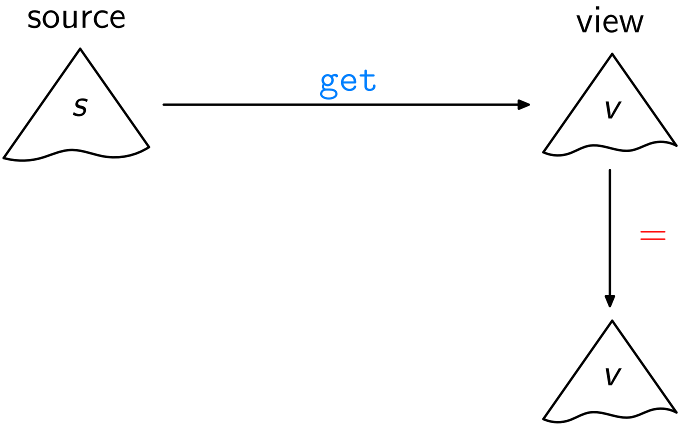
# Bidirectional Transformations

A specific (asymmetric) setting:



# Bidirectional Transformations

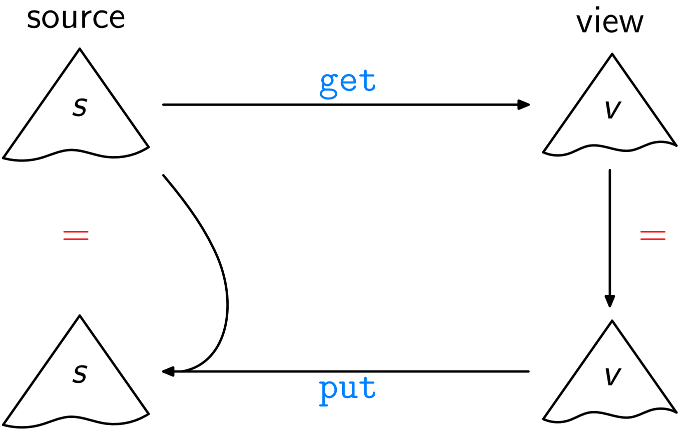
A specific (asymmetric) setting:




GetPut law

# Bidirectional Transformations

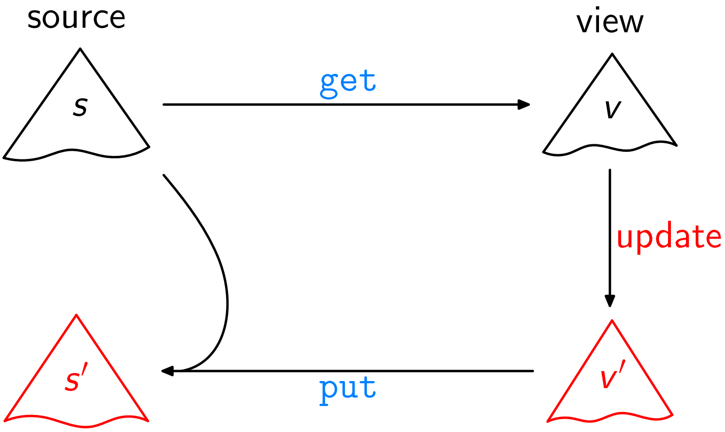
A specific (asymmetric) setting:




GetPut law

# Bidirectional Transformations

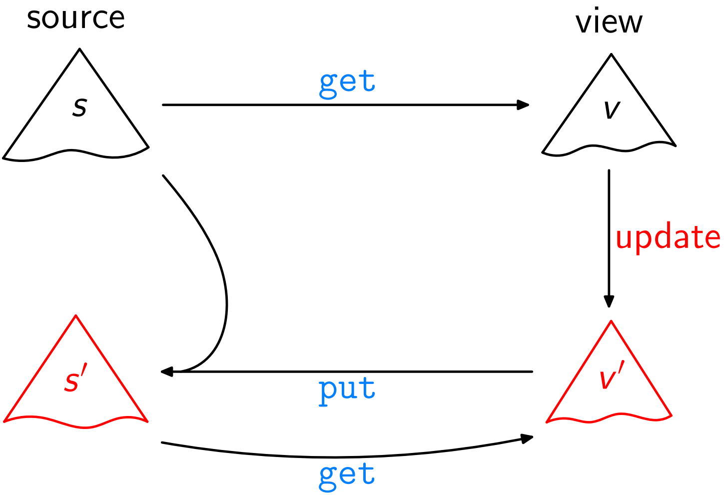
A specific (asymmetric) setting:




PutGet law

# Bidirectional Transformations

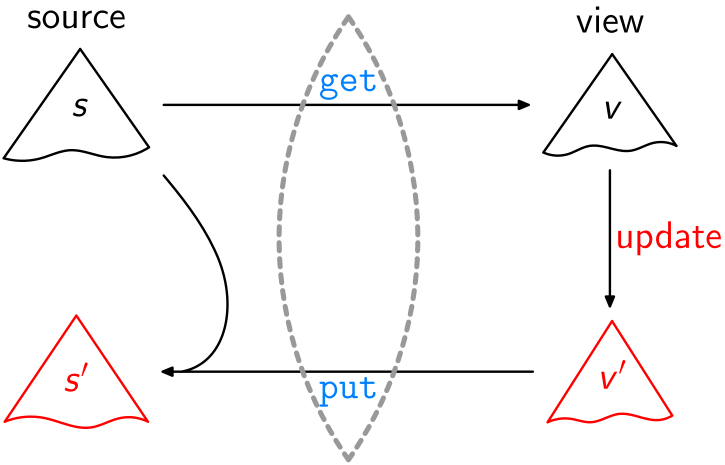
A specific (asymmetric) setting:




PutGet law

# Bidirectional Transformations

A specific (asymmetric) setting:



## Bidirectionalization “by Hand”

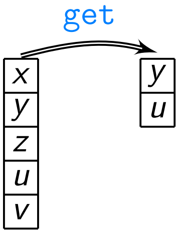
A simple example:

$$
\begin{aligned}
&\texttt{get} :: [\alpha] \to [\alpha] \\
&\texttt{get}\ [\,] &&= [\,] \\
&\texttt{get}\ [x] &&= [\,] \\
&\texttt{get}\ (x : y : zs) &&= y : (\texttt{get}\ zs)
\end{aligned}
$$

# Bidirectionalization "by Hand"

A simple example:

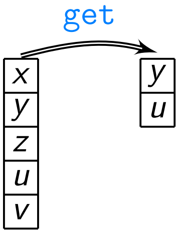
$$\begin{aligned}
&\texttt{get} :: [\alpha] \to [\alpha] \\
&\texttt{get}\ [\,] && = [\,] \\
&\texttt{get}\ [x] && = [\,] \\
&\texttt{get}\ (x : y : zs) && = y : (\texttt{get}\ zs)
\end{aligned}$$



# Bidirectionalization “by Hand”

A simple example:

$$
\begin{aligned}
&\texttt{get} :: [\alpha] \to [\alpha] \\
&\texttt{get}\ [\,] && = [\,] \\
&\texttt{get}\ [x] && = [\,] \\
&\texttt{get}\ (x : y : zs) && = y : (\texttt{get}\ zs)
\end{aligned}
$$



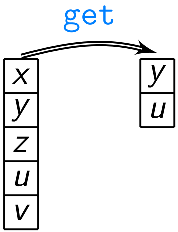

One possible backwards transformation:

$$
\begin{aligned}
&\texttt{put}\ [\,] && [\,] && = [\,] \\
&\texttt{put}\ [\,] && [x] && = [x] \\
&\texttt{put}\ (y' : v') && (x : y : zs) && = x : y' : (\texttt{put}\ v'\ zs)
\end{aligned}
$$

# Bidirectionalization "by Hand"

A simple example:

$$
\begin{aligned}
&\mathtt{get} :: [\alpha] \to [\alpha] \\
&\mathtt{get}\ [\,] && = [\,] \\
&\mathtt{get}\ [x] && = [\,] \\
&\mathtt{get}\ (x : y : zs) && = y : (\mathtt{get}\ zs)
\end{aligned}
$$




One possible backwards transformation:

$$
\begin{aligned}
&\mathtt{put}\ [\,] && [\,] && = [\,] \\
&\mathtt{put}\ [\,] && [x] && = [x] \\
&\mathtt{put}\ (y' : v') && (x : y : zs) && = x : y' : (\mathtt{put}\ v'\ zs)
\end{aligned}
$$

not total!

# Programming Language Approaches

There has been, and is ongoing, great work in the "lenses" PL/DSLs tradition [Foster et al., ACM TOPLAS'07, …]. Not covered today.

# Programming Language Approaches

There has been, and is ongoing, great work in the "lenses" PL/DSLs tradition [Foster et al., ACM TOPLAS'07, ...]. Not covered today.

We will mention/look at:

- syntactic program transformation
- semantic/type-based transformation
- benefits of higher-order types and abstraction
- search-based program synthesis (if time permits, otherwise see PEPM'12 short paper)

# A Principled Approach: Constant-Complement [Bancilhon & Spyratos, ACM TODS'81]

Given

$$\texttt{get} :: S \to V$$

# A Principled Approach: Constant-Complement [Bancilhon & Spyratos, ACM TODS'81]

Given

$$\texttt{get} :: S \to V$$

define a $C$ and

$$\texttt{res} :: S \to C$$

# A Principled Approach: Constant-Complement [Bancilhon & Spyratos, ACM TODS'81]

Given

$$\mathtt{get} :: S \to V$$

define a $C$ and

$$\mathtt{res} :: S \to C$$

such that

$$\mathtt{paired} = \lambda s \to (\mathtt{get}\ s, \mathtt{res}\ s)$$

is injective

# A Principled Approach: Constant-Complement [Bancilhon & Spyratos, ACM TODS'81]

Given

$$\texttt{get} :: S \to V$$

define a $C$ and

$$\texttt{res} :: S \to C$$

such that

$$\texttt{paired} = \lambda s \to (\texttt{get}\ s, \texttt{res}\ s)$$

is injective and has an inverse $\texttt{inv} :: (V, C) \to S$.

# A Principled Approach: Constant-Complement [Bancilhon & Spyratos, ACM TODS'81]

Given

$$\mathtt{get} :: S \to V$$

define a $C$ and

$$\mathtt{res} :: S \to C$$

such that

$$\mathtt{paired} = \lambda s \to (\mathtt{get}\ s, \mathtt{res}\ s)$$

is injective and has an inverse $\mathtt{inv} :: (V, C) \to S$.

Then:

$$\mathtt{put} :: V \to S \to S$$
$$\mathtt{put}\ v'\ s = \mathtt{inv}\ (v', \mathtt{res}\ s)$$

# A Principled Approach: Constant-Complement [Bancilhon & Spyratos, ACM TODS'81]

Given

$$\texttt{get} :: S \to V$$

define a $C$ and

$$\texttt{res} :: S \to C$$

such that

$$\texttt{paired} = \lambda s \to (\texttt{get}\ s, \texttt{res}\ s)$$

is injective and has an inverse $\texttt{inv} :: (V, C) \to S$.

has to be effective!

Then:

$$\texttt{put} :: V \to S \to S$$
$$\texttt{put}\ v'\ s = \texttt{inv}\ (v', \texttt{res}\ s)$$

## A Principled Approach: Constant-Complement

Guarantees "very-well-behavedness":

- $\texttt{put}\ (\texttt{get}\ s)\ s = s$
- $\texttt{get}\ (\texttt{put}\ v'\ s) = v'$
- $\texttt{put}\ v''\ (\texttt{put}\ v'\ s) = \texttt{put}\ v''\ s$

# A Principled Approach: Constant-Complement

Guarantees "very-well-behavedness":

- $\texttt{put}\ (\texttt{get}\ s)\ s = s$
- $\texttt{get}\ (\texttt{put}\ v'\ s) = v'$
- $\texttt{put}\ v''\ (\texttt{put}\ v'\ s) = \texttt{put}\ v''\ s$

Example:

$$\texttt{get} :: \mathsf{Nat} \to \mathsf{Nat}$$
$$\texttt{get}\ n = n\ \text{`}\texttt{div}\text{`}\ 2$$

# A Principled Approach: Constant-Complement

Guarantees "very-well-behavedness":

- $\mathtt{put}\ (\mathtt{get}\ s)\ s = s$
- $\mathtt{get}\ (\mathtt{put}\ v'\ s) = v'$
- $\mathtt{put}\ v''\ (\mathtt{put}\ v'\ s) = \mathtt{put}\ v''\ s$

Example:

$$\mathtt{get} :: \mathsf{Nat} \to \mathsf{Nat} \qquad \mathtt{res} :: \mathsf{Nat} \to \mathsf{Nat}_2$$
$$\mathtt{get}\ n = n\ \text{`}\mathtt{div}\text{`}\ 2 \qquad \mathtt{res}\ n = n\ \text{`}\mathtt{mod}\text{`}\ 2$$

# A Principled Approach: Constant-Complement

Guarantees "very-well-behavedness":

- $\texttt{put}\ (\texttt{get}\ s)\ s = s$
- $\texttt{get}\ (\texttt{put}\ v'\ s) = v'$
- $\texttt{put}\ v''\ (\texttt{put}\ v'\ s) = \texttt{put}\ v''\ s$

Example:

$$\begin{array}{ll} \texttt{get} :: \mathsf{Nat} \to \mathsf{Nat} & \qquad \texttt{res} :: \mathsf{Nat} \to \mathsf{Nat}_2 \\ \texttt{get}\ n = n\ \text{`}\texttt{div}\text{`}\ 2 & \qquad \texttt{res}\ n = n\ \text{`}\texttt{mod}\text{`}\ 2 \end{array}$$

$$\begin{array}{l} \texttt{inv} :: (\mathsf{Nat}, \mathsf{Nat}_2) \to \mathsf{Nat} \\ \texttt{inv}\ (v', c) = 2 * v' + c \end{array}$$

# A Principled Approach: Constant-Complement

Example:

$$\texttt{get} :: \mathsf{Nat} \to \mathsf{Nat} \qquad \texttt{res} :: \mathsf{Nat} \to \mathsf{Nat}_2$$
$$\texttt{get}\ n = n \text{ `}\texttt{div}\text{` } 2 \qquad \texttt{res}\ n = n \text{ `}\texttt{mod}\text{` } 2$$

$$\texttt{inv} :: (\mathsf{Nat}, \mathsf{Nat}_2) \to \mathsf{Nat}$$
$$\texttt{inv}\ (v', c) = 2 * v' + c$$

# A Principled Approach: Constant-Complement

Example:

$$\mathtt{get} :: \mathsf{Nat} \to \mathsf{Nat} \qquad \mathtt{res} :: \mathsf{Nat} \to \mathsf{Nat}_2$$
$$\mathtt{get}\ n = n\ \text{`}\mathtt{div}\text{`}\ 2 \qquad \mathtt{res}\ n = n\ \text{`}\mathtt{mod}\text{`}\ 2$$

$$\mathtt{inv} :: (\mathsf{Nat}, \mathsf{Nat}_2) \to \mathsf{Nat}$$
$$\mathtt{inv}\ (v', c) = 2 * v' + c$$

Then:

$$\mathtt{put} :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$
$$\mathtt{put}\ v'\ s = \mathtt{inv}\ (v', \mathtt{res}\ s)$$

# A Principled Approach: Constant-Complement

Example:

$$\texttt{get} :: \mathsf{Nat} \to \mathsf{Nat} \qquad \texttt{res} :: \mathsf{Nat} \to \mathsf{Nat}_2$$
$$\texttt{get}\ n = n\ \text{`}\texttt{div}\text{`}\ 2 \qquad \texttt{res}\ n = n\ \text{`}\texttt{mod}\text{`}\ 2$$

$$\texttt{inv} :: (\mathsf{Nat}, \mathsf{Nat}_2) \to \mathsf{Nat}$$
$$\texttt{inv}\ (v', c) = 2 * v' + c$$

Then:

$$\begin{aligned} \texttt{put} &:: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\ \texttt{put}\ v'\ s &= \texttt{inv}\ (v', \texttt{res}\ s) \\ &= 2 * v' + s\ \text{`}\texttt{mod}\text{`}\ 2 \end{aligned}$$

# A Principled Approach: Constant-Complement

Example:

$$\texttt{get} :: \mathsf{Nat} \to \mathsf{Nat} \qquad \texttt{res} :: \mathsf{Nat} \to \mathsf{Nat}_2$$
$$\texttt{get}\ n = n \ \text{`}\texttt{div}\text{`}\ 2 \qquad \texttt{res}\ n = n \ \text{`}\texttt{mod}\text{`}\ 2$$

$$\texttt{inv} :: (\mathsf{Nat}, \mathsf{Nat}_2) \to \mathsf{Nat}$$
$$\texttt{inv}\ (v', c) = 2 * v' + c$$

other choices possible, and give different behavior

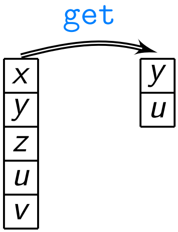
Then:

$$\texttt{put} :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$
$$\begin{aligned}\texttt{put}\ v'\ s &= \texttt{inv}\ (v', \texttt{res}\ s)\\ &= 2 * v' + s \ \text{`}\texttt{mod}\text{`}\ 2\end{aligned}$$

# Automatic Bidirectionalization by Example

Let:

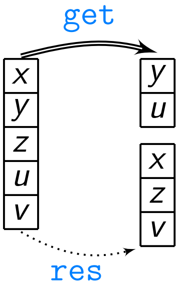
$$\begin{aligned}
&\mathtt{get} :: [\alpha] \to [\alpha] \\
&\mathtt{get}\ [\,] && = [\,] \\
&\mathtt{get}\ [x] && = [\,] \\
&\mathtt{get}\ (x : y : zs) && = y : (\mathtt{get}\ zs)
\end{aligned}$$



# Automatic Bidirectionalization by Example

Let:

$$\begin{aligned}
&\texttt{get} :: [\alpha] \to [\alpha] \\
&\texttt{get}\ [\,] &&= [\,] \\
&\texttt{get}\ [x] &&= [\,] \\
&\texttt{get}\ (x : y : zs) &&= y : (\texttt{get}\ zs)
\end{aligned}$$




A syntactically derived complement function:

$$\begin{aligned}
&\texttt{res}\ [\,] &&= \mathsf{C}_1 \\
&\texttt{res}\ [x] &&= \mathsf{C}_2\ x \\
&\texttt{res}\ (x : y : zs) &&= \mathsf{C}_3\ x\ (\texttt{res}\ zs)
\end{aligned}$$

# Automatic Bidirectionalization by Example

A syntactically derived complement function:

$$\begin{aligned}
\texttt{res}\ [\,] &= \mathsf{C}_1 \\
\texttt{res}\ [x] &= \mathsf{C}_2\ x \\
\texttt{res}\ (x : y : zs) &= \mathsf{C}_3\ x\ (\texttt{res}\ zs)
\end{aligned}$$

Syntactic pairing:

$$\begin{aligned}
\texttt{paired}\ [\,] &= ([\,] \quad , \mathsf{C}_1) \\
\texttt{paired}\ [x] &= ([\,] \quad , \mathsf{C}_2\ x) \\
\texttt{paired}\ (x : y : zs) &= (y : v, \mathsf{C}_3\ x\ c) \\
&\quad \textbf{where}\ (v, c) = \texttt{paired}\ zs
\end{aligned}$$

## Automatic Bidirectionalization by Example

Syntactic pairing:

$$
\begin{aligned}
&\texttt{paired}\ [\,] &&= ([\,] \quad , \mathrm{C}_1) \\
&\texttt{paired}\ [x] &&= ([\,] \quad , \mathrm{C}_2\ x) \\
&\texttt{paired}\ (x : y : zs) &&= (y : v, \mathrm{C}_3\ x\ c) \\
& && \quad \textbf{where}\ (v, c) = \texttt{paired}\ zs
\end{aligned}
$$

Syntactic inversion:

$$
\begin{aligned}
&\texttt{inv}\ ([\,] \quad , \mathrm{C}_1) &&= [\,] \\
&\texttt{inv}\ ([\,] \quad , \mathrm{C}_2\ x) &&= [x] \\
&\texttt{inv}\ (y : v, \mathrm{C}_3\ x\ c) &&= x : y : zs \\
& && \quad \textbf{where}\ zs = \texttt{inv}\ (v, c)
\end{aligned}
$$

# Automatic Bidirectionalization by Example

Syntactic inversion:

$$
\begin{aligned}
&\mathtt{inv}\,([\,]\quad, C_1) &&= [\,] \\
&\mathtt{inv}\,([\,]\quad, C_2\ x) &&= [x] \\
&\mathtt{inv}\,(y : v, C_3\ x\ c) &&= x : y : zs \\
& && \quad \textbf{where}\ zs = \mathtt{inv}\,(v, c)
\end{aligned}
$$

Then,

$$\mathtt{put}\ v'\ s = \mathtt{inv}\,(v', \mathtt{res}\ s)$$

## Automatic Bidirectionalization by Example

Syntactic inversion:

$$
\begin{aligned}
&\texttt{inv}\ ([\,]\quad, \mathsf{C}_1) &&= [\,] \\
&\texttt{inv}\ ([\,]\quad, \mathsf{C}_2\ x) &&= [x] \\
&\texttt{inv}\ (y : v, \mathsf{C}_3\ x\ c) &&= x : y : zs \\
& && \quad\ \textbf{where}\ zs = \texttt{inv}\ (v, c)
\end{aligned}
$$

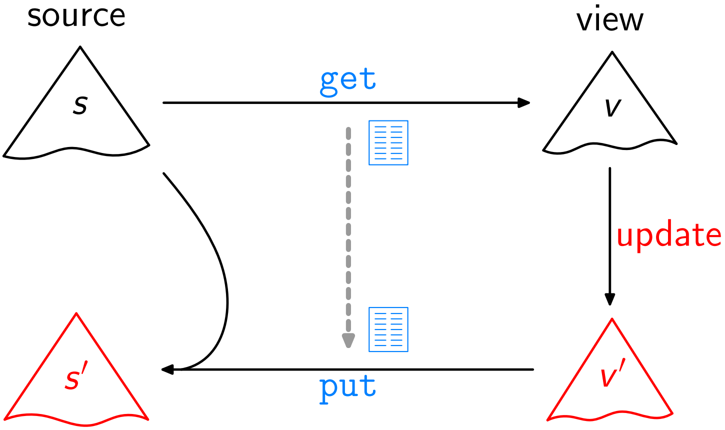
Then,

$$\texttt{put}\ v'\ s = \texttt{inv}\ (v', \texttt{res}\ s)$$

corresponds to (the earlier seen):

$$
\begin{aligned}
&\texttt{put}\ [\,] && [\,] && = [\,] \\
&\texttt{put}\ [\,] && [x] && = [x] \\
&\texttt{put}\ (y' : v') && (x : y : zs) && = x : y' : (\texttt{put}\ v'\ zs)
\end{aligned}
$$

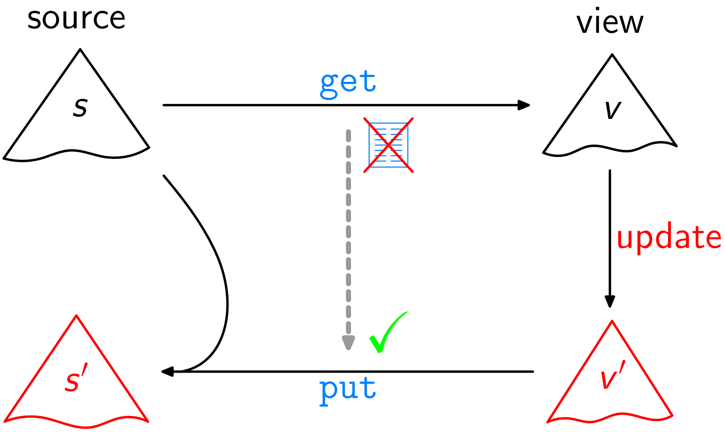
# Automatic Bidirectionalization




Syntactic Bidirectionalization

[Matsuda et al., ICFP'07]

# Automatic Bidirectionalization




Semantic Bidirectionalization

[V., POPL'09]

# Semantic Bidirectionalization

Aim: Write higher-order function `bff`[†] such that any `get` and `bff get` satisfy GetPut, PutGet, . . . .

† "Bidirectionalization for free!"

## Semantic Bidirectionalization

Aim: Write higher-order function `bff`[†] such that any `get` and `bff get` satisfy GetPut, PutGet, ....

Examples:

$$\text{“abc”} \xrightarrow{\texttt{tail}} \text{“bc”}$$

[†] “Bidirectionalization for free!”

# Semantic Bidirectionalization

Aim: Write higher-order function `bff`† such that any `get` and `bff get` satisfy GetPut, PutGet, . . . .

Examples:

$$\begin{tikzcd}
\text{``abc''} \arrow[rr, "\texttt{tail}"] & & \text{``bc''} \arrow[d, "\text{update}"] \\
& & \text{``de''}
\end{tikzcd}$$

† "Bidirectionalization for free!"

# Semantic Bidirectionalization

Aim: Write higher-order function `bff`† such that any `get` and `bff get` satisfy GetPut, PutGet, ....

Examples:

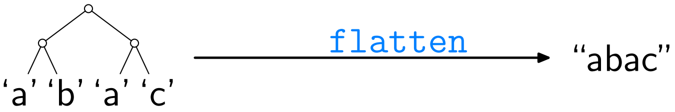
$$\begin{tikzcd}[column sep=huge, row sep=huge]
\text{"abc"} \arrow[r, "\texttt{tail}"] \arrow[d, bend left] & \text{"bc"} \arrow[d, "\text{update}"] \\
\text{"ade"} & \text{"de"} \arrow[l, "\texttt{bff tail}"]
\end{tikzcd}$$

† "Bidirectionalization for free!"

# Semantic Bidirectionalization

Aim: Write higher-order function `bff`† such that any `get` and `bff get` satisfy GetPut, PutGet, . . . .

Examples:




† “Bidirectionalization for free!”

# Semantic Bidirectionalization

Aim: Write higher-order function `bff`† such that any `get` and `bff get` satisfy GetPut, PutGet, . . . .

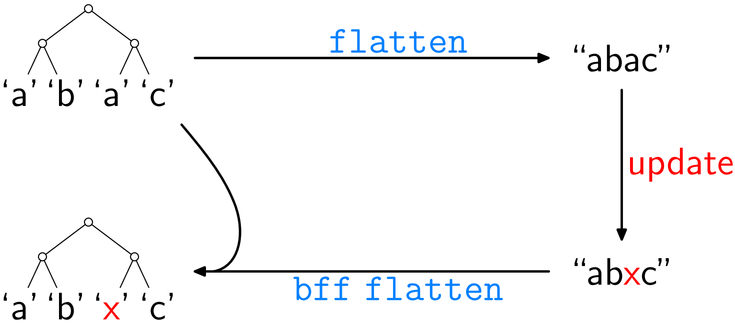
Examples:


† "Bidirectionalization for free!"

# Semantic Bidirectionalization

Aim: Write higher-order function `bff`† such that any `get` and `bff get` satisfy GetPut, PutGet, . . . .

Examples:




† "Bidirectionalization for free!"

# Analyzing Specific Instances

Assume we are given some

$$\texttt{get} :: [\alpha] \to [\alpha]$$

How can we, or `bff`, analyze it without access to its source code?

# Analyzing Specific Instances

Assume we are given some

$$\texttt{get} :: [\alpha] \to [\alpha]$$

How can we, or `bff`, analyze it without access to its source code?

Idea: How about applying `get` to some input?

# Analyzing Specific Instances

Assume we are given some

$$\texttt{get} :: [\alpha] \to [\alpha]$$

How can we, or `bff`, analyze it without access to its source code?

Idea: How about applying `get` to some input?

Like:

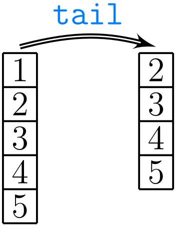
$$\texttt{get}\ [1..n] = \begin{cases} [2..n] & \text{if } \texttt{get} = \texttt{tail} \\ [n..1] & \text{if } \texttt{get} = \texttt{reverse} \\ [1..(\texttt{min}\ 5\ n)] & \text{if } \texttt{get} = \texttt{take}\ 5 \\ & \vdots \end{cases}$$

# Analyzing Specific Instances

Like:

$$\texttt{get}\ [1..n] = \begin{cases} [2..n] & \text{if } \texttt{get} = \texttt{tail} \\ [n..1] & \text{if } \texttt{get} = \texttt{reverse} \\ [1..(\texttt{min}\ 5\ n)] & \text{if } \texttt{get} = \texttt{take}\ 5 \\ & \vdots \end{cases}$$

Indeed, this gives us traceability for free:

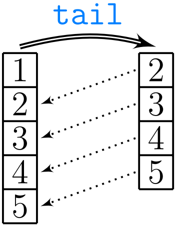


# Analyzing Specific Instances

Like:

$$\texttt{get}\ [1..n] = \begin{cases} [2..n] & \text{if } \texttt{get} = \texttt{tail} \\ [n..1] & \text{if } \texttt{get} = \texttt{reverse} \\ [1..(\texttt{min}\ 5\ n)] & \text{if } \texttt{get} = \texttt{take}\ 5 \\ & \vdots \end{cases}$$

Indeed, this gives us traceability for free:

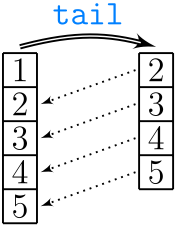


# Analyzing Specific Instances

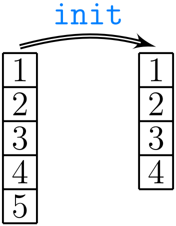
Like:

$$\mathtt{get}\ [1..n] = \begin{cases} [2..n] & \text{if } \mathtt{get} = \mathtt{tail} \\ [n..1] & \text{if } \mathtt{get} = \mathtt{reverse} \\ [1..(\mathtt{min}\ 5\ n)] & \text{if } \mathtt{get} = \mathtt{take}\ 5 \\ & \vdots \end{cases}$$

Indeed, this gives us traceability for free:




and

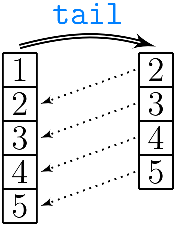


# Analyzing Specific Instances

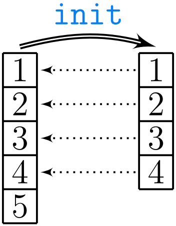
Like:

$$\texttt{get}\ [1..n] = \begin{cases} [2..n] & \text{if } \texttt{get} = \texttt{tail} \\ [n..1] & \text{if } \texttt{get} = \texttt{reverse} \\ [1..(\texttt{min}\ 5\ n)] & \text{if } \texttt{get} = \texttt{take}\ 5 \\ \vdots & \end{cases}$$

Indeed, this gives us traceability for free:




and

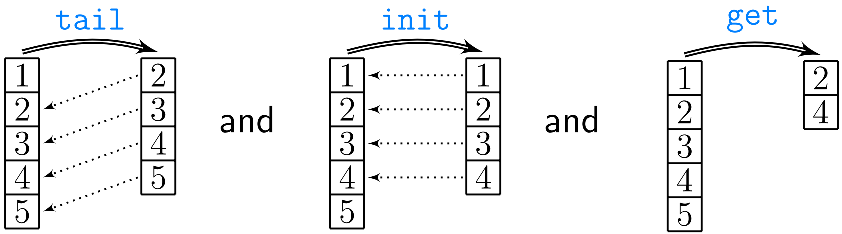


# Analyzing Specific Instances

Like:

$$\texttt{get}\ [1..n] = \begin{cases} [2..n] & \text{if } \texttt{get} = \texttt{tail} \\ [n..1] & \text{if } \texttt{get} = \texttt{reverse} \\ [1..(\texttt{min}\ 5\ n)] & \text{if } \texttt{get} = \texttt{take}\ 5 \\ & \vdots \end{cases}$$

Indeed, this gives us traceability for free:

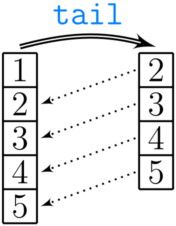


# Analyzing Specific Instances

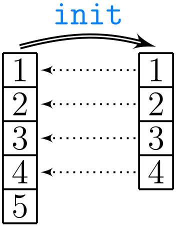
Like:

$$\mathtt{get}\ [1..n] = \begin{cases} [2..n] & \text{if } \mathtt{get} = \mathtt{tail} \\ [n..1] & \text{if } \mathtt{get} = \mathtt{reverse} \\ [1..(\mathtt{min}\ 5\ n)] & \text{if } \mathtt{get} = \mathtt{take}\ 5 \\ & \vdots \end{cases}$$

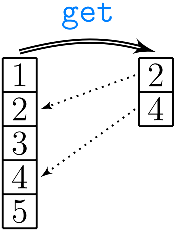
Indeed, this gives us traceability for free:




and




and

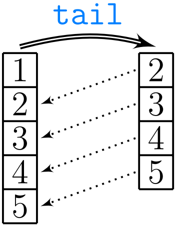


# Analyzing Specific Instances

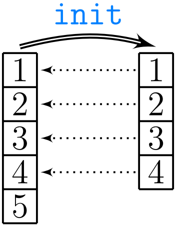
Like:

$$\texttt{get}\ [1..n] = \begin{cases} [2..n] & \text{if } \texttt{get} = \texttt{tail} \\ [n..1] & \text{if } \texttt{get} = \texttt{reverse} \\ [1..(\texttt{min}\ 5\ n)] & \text{if } \texttt{get} = \texttt{take}\ 5 \\ & \vdots \end{cases}$$

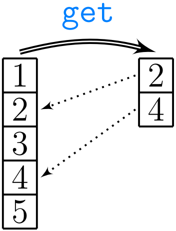
Indeed, this gives us traceability for free:




and

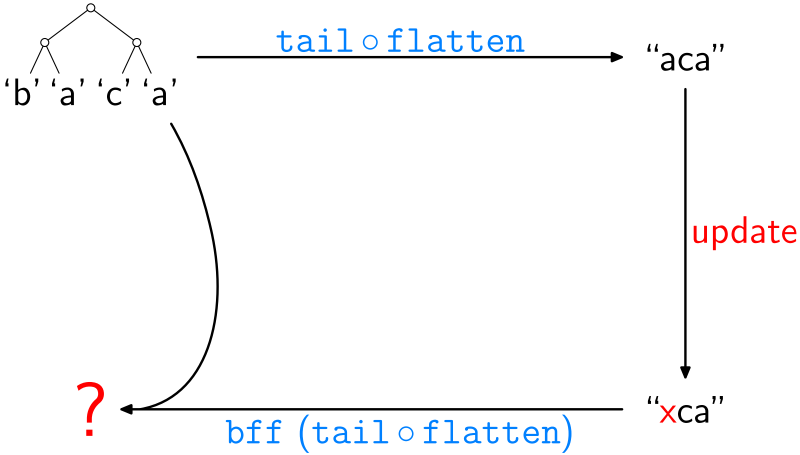



and

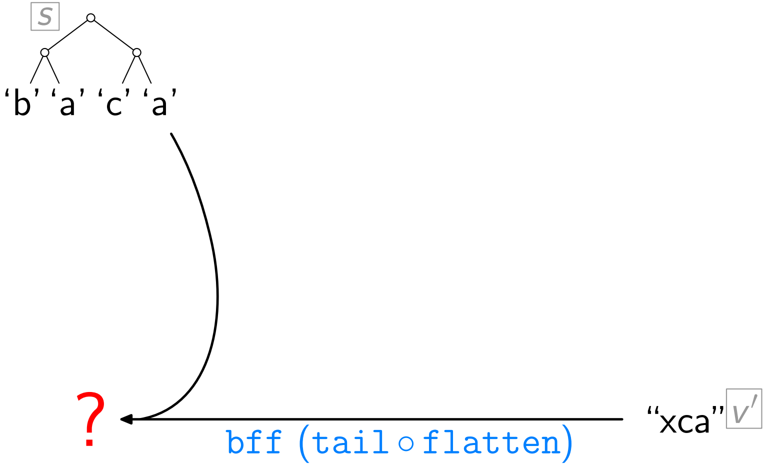



Then transfer the gained insights to arbitrary lists!

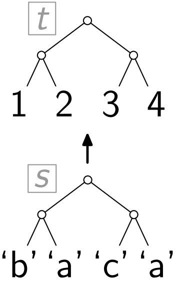
# Semantic Bidirectionalization by Example



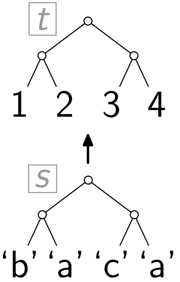
# Semantic Bidirectionalization by Example



# Semantic Bidirectionalization by Example



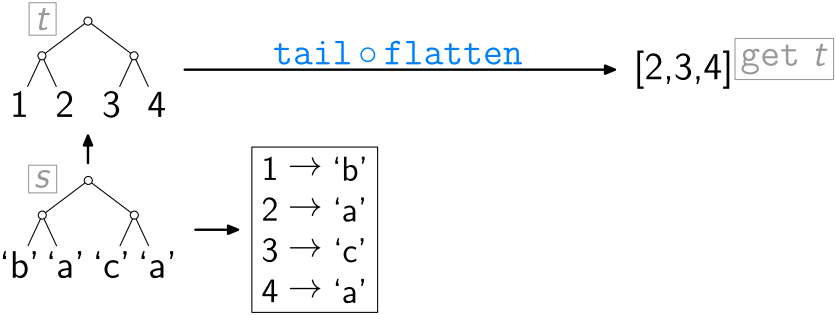
# Semantic Bidirectionalization by Example




$1 \to$ 'b'
$2 \to$ 'a'
$3 \to$ 'c'
$4 \to$ 'a'

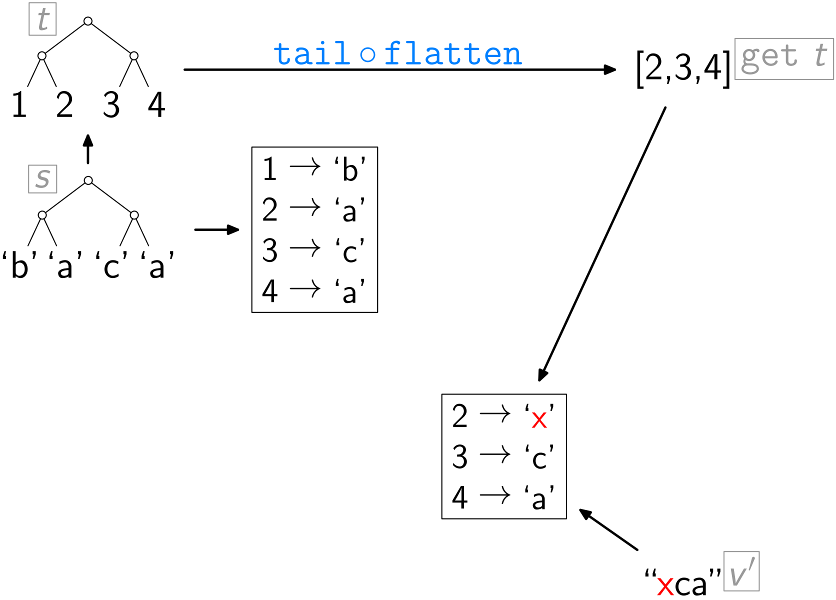
"xca" $v'$

# Semantic Bidirectionalization by Example

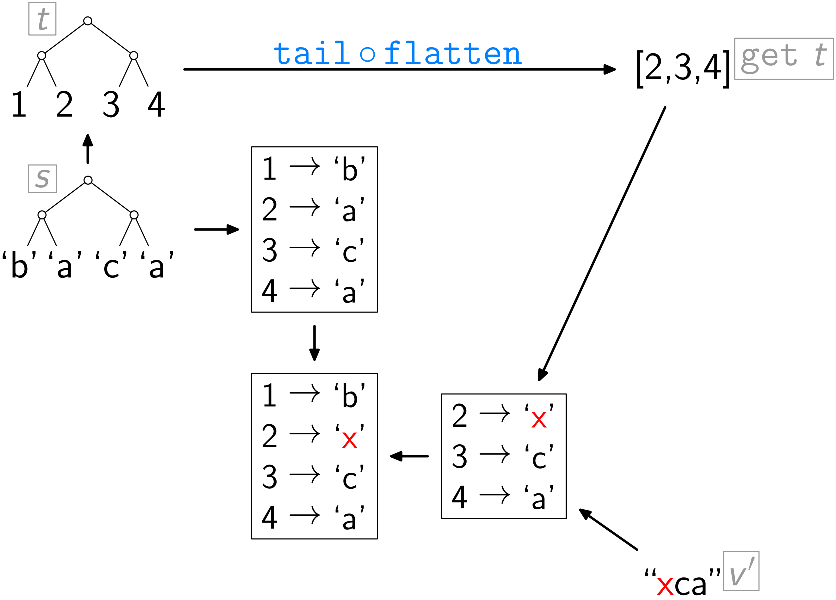



"xca" $v'$

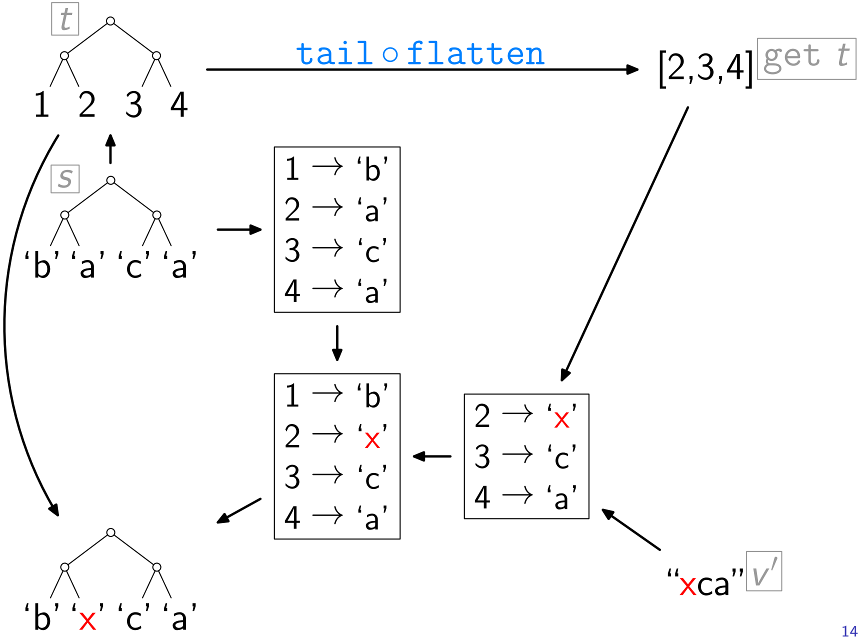
# Semantic Bidirectionalization by Example



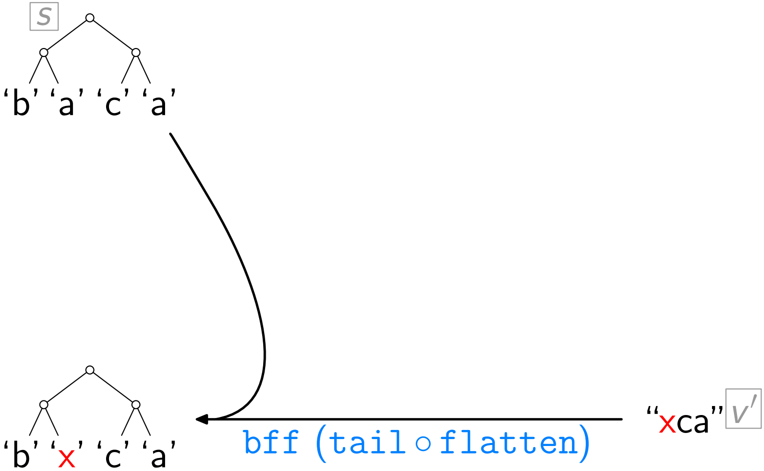
# Semantic Bidirectionalization by Example



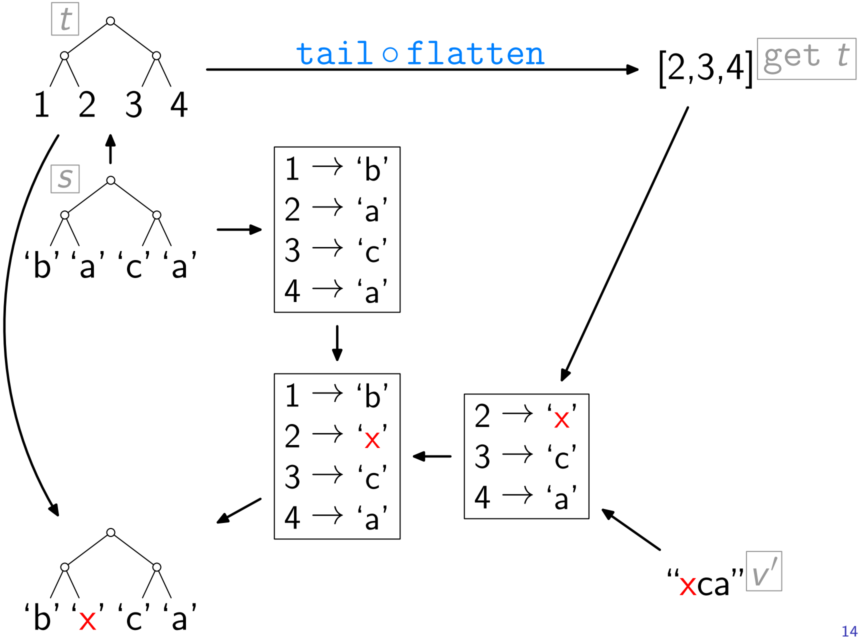
# Semantic Bidirectionalization by Example



# Semantic Bidirectionalization by Example



# Semantic Bidirectionalization by Example



# Taking Stock of Automatic Bidirectionalization

[Matsuda et al., ICFP'07]:

- depends on syntactic restraints
- allows (ad-hoc) some shape-changing updates

# Taking Stock of Automatic Bidirectionalization

[Matsuda et al., ICFP'07]:

- depends on syntactic restraints
- allows (ad-hoc) some shape-changing updates

[V., POPL'09]:

- very lightweight, easy access to bidirectionality
- essential role: polymorphic function types
- major problem: rejects shape-changing updates

# Taking Stock of Automatic Bidirectionalization

[Matsuda et al., ICFP'07]:

- depends on syntactic restraints
- allows (ad-hoc) some shape-changing updates

[V., POPL'09]:

- very lightweight, easy access to bidirectionality
- essential role: polymorphic function types
- major problem: rejects shape-changing updates

[V. et al., ICFP'10]:

- synthesis of the two techniques
- inherits limitations in program coverage from both
- strictly better in terms of updatability than either

# References I


F. Bancilhon and N. Spyratos.
Update semantics of relational views.
*ACM Transactions on Database Systems*, 6(3):557–575, 1981.

J.N. Foster, M.B. Greenwald, J.T. Moore, B.C. Pierce, and A. Schmitt.
Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem.
*ACM Transactions on Programming Languages and Systems*, 29(3):17, 2007.

S. Katayama.
Systematic search for lambda expressions.
In *Trends in Functional Programming 2005, Revised Selected Papers*, pages 111–126. Intellect, 2007.

# References II

E. Kitzelmann and U. Schmid.
Inductive synthesis of functional programs: An explanation based generalization approach.
*Journal of Machine Learning Research*, 7:429–454, 2006.

K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi.
Bidirectionalization transformation based on automatic derivation of view complement functions.
In *International Conference on Functional Programming, Proceedings*, pages 47–58. ACM Press, 2007.

J. Voigtländer, Z. Hu, K. Matsuda, and M. Wang.
Combining syntactic and semantic bidirectionalization.
In *International Conference on Functional Programming, Proceedings*, pages 181–192. ACM Press, 2010.

# References III


J. Voigtländer.
Bidirectionalization for free!
In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009.