Programming Language Approaches to Bidirectional Transformation

Janis Voigtländer

University of Bonn

LDTA’12
Bidirectional Transformations (BX)
Bidirectional Transformations (BX)

\[
\begin{align*}
A & \iff B \\
\text{to} & \\
\text{from} & \\
\text{concrete syntax} & \iff \text{abstract syntax}
\end{align*}
\]
Bidirectional Transformations (BX)

database source ⇔ materialized view
Bidirectional Transformations (BX)

doctrine representation $\iff$ screen visualization
Bidirectional Transformations (BX)

software model $\iff$ code
Bidirectional Transformations (BX)

abstract datatype \iff actual implementation
Bidirectional Transformations (BX)

program input ⇔ program output
Bidirectional Transformations (BX)

- concrete syntax ⇔ abstract syntax
- database source ⇔ materialized view
- document representation ⇔ screen visualization
- software model ⇔ code
- abstract datatype ⇔ actual implementation
- program input ⇔ program output
Bidirectional Transformations

- $a_1$ to $b_1$

- unless bijective, typically additional information needed/useful:
  - about connections between $A$ and $B$ (objects)
  - about the updates on either side
Bidirectional Transformations

- $a_1$ to $b_1$
- $b_1$ to $b_2$

Unless bijective, typically additional information needed/useful:

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

- $a_1$ to $b_1$
- $a_2$ from $b_2$

Unless bijective, typically additional information needed/useful:
- about connections between $A$ and $B$ (objects)
- about the updates on either side
Bidirectional Transformations

- $a_1$ to $b_1$
- $a_2$ from $b_2$
- $a_3$

Unless bijective, typically additional information needed/useful:

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

- $a_1$ to $b_1$
- $a_2$ from $b_2$
- $a_3$ to $b_3$

unless bijective, typically additional information needed/useful:

▶ about connections between $A$ and $B$ (objects)
▶ about the updates on either side
Bidirectional Transformations

- $a_1$ to $b_1$
- $a_2$ from $b_2$
- $a_3$ to $b_3$

Unless bijective, typically additional information needed/useful:

- About connections between $A$ and $B$ (objects)
- About the updates on either side
Bidirectional Transformations

- $a_1$ to $b_1$
- $a_2$ from $b_2$
- $a_3$ to $b_3$

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)
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 $a_i \cdots b_i$ connections. For example:
Bidirectional Transformations

A closer look at representing \( a_i \rightarrow b_i \) connections.

For example:

\[
\begin{array}{cccc}
\bullet & \bullet & \bullet & \bullet \\
\cdot \cdot & \cdot \cdot & \cdot \cdot & \cdot \cdot \\
X & Y & Z & U \\
\end{array}
\]

to

\[
\begin{array}{cccc}
\bullet & \bullet & \bullet & \bullet \\
\cdot \cdot & \cdot \cdot & \cdot \cdot & \cdot \cdot \\
Y & Z & U & V \\
\end{array}
\]

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

Because of:
Bidirectional Transformations

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

For example:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

or

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Because of:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

or

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3
**Bidirectional Transfomations**

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

For example:

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

Because of:
Bidirectional Transformations

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

For example:

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

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

For example:

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?
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:

source $s$ ---get--- view $v$
Bidirectional Transformations

A specific (asymmetric) setting:

source

s

get

view

v

update

v′
Bidirectional Transformations

A specific (asymmetric) setting:

source

<table>
<thead>
<tr>
<th>s</th>
</tr>
</thead>
</table>

get

<table>
<thead>
<tr>
<th>v</th>
</tr>
</thead>
</table>

update

<table>
<thead>
<tr>
<th>v'</th>
</tr>
</thead>
</table>

put

<table>
<thead>
<tr>
<th>s'</th>
</tr>
</thead>
</table>
Bidirectional Transformations

A specific (asymmetric) setting:

source

\[ s \quad \text{get} \quad v \quad \text{update} \quad v' \]

\[ s' \quad \text{put} \]
Bidirectional Transformations

A specific (asymmetric) setting:

GetPut law
Bidirectional Transformations

A specific (asymmetric) setting:

GetPut law
Bidirectional Transformations

A specific (asymmetric) setting:

source

\[ s \to v \]

view

\[ v' \leftarrow s' \]

PutGet law
Bidirectional Transformations

A specific (asymmetric) setting:

source

\[ s \]

view

\[ v \]

get

PutGet law

update

\[ s' \]

\[ v' \]

put

get
Bidirectional Transformations

A specific (asymmetric) setting:

source

\[ s \]

view

\[ v \]

\[ s' \]

\[ v' \]

get

put

update
Bidirectionalization “by Hand”

A simple example:

\[
\text{get} :: \left[\alpha\right] \rightarrow \left[\alpha\right] \\
\text{get} \left[\right] = \left[\right] \\
\text{get} \left[x\right] = \left[\right] \\
\text{get} \left(x : y : zs\right) = y : \left(\text{get} \ zs\right)
\]
Bidirectionalization “by Hand”

A simple example:

\[ \text{get} :: [\alpha] \rightarrow [\alpha] \]
\[ \text{get} [\ ] = [\ ] \]
\[ \text{get} [x] = [\ ] \]
\[ \text{get} (x : y : zs) = y : (\text{get} zs) \]
Bidirectionalization “by Hand”

A simple example:

\[
\text{get} :: [\alpha] \rightarrow [\alpha]
\]

\[
\text{get} \ [\] = [\]
\]

\[
\text{get} \ [x] = [\]
\]

\[
\text{get} \ (x : y : zs) = y : (\text{get} \ zs)
\]

One possible backwards transformation:

\[
\text{put} \ [\] \ [\] = [\]
\]

\[
\text{put} \ [\] \ [x] = [x]
\]

\[
\text{put} \ (y' : v') \ (x : y : zs) = x : y' : (\text{put} \ v' \ zs)
\]
Bidirectionalization “by Hand”

A simple example:

\[
\begin{align*}
\text{get} :: [\alpha] & \rightarrow [\alpha] \\
\text{get} [\ ] & = [\ ] \\
\text{get} [x] & = [\ ] \\
\text{get} (x : y : zs) & = y : (\text{get} zs)
\end{align*}
\]

One possible backwards transformation:

\[
\begin{align*}
\text{put} [\ ] [\ ] & = [\ ] \\
\text{put} [\ ] [x] & = [x] \\
\text{put} (y' : v') (x : y : zs) & = x : y' : (\text{put} v' zs)
\end{align*}
\]

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 & Spyropoulos, ACM TODS’81]

Given

\[ \text{get} :: S \rightarrow V \]
A Principled Approach: Constant-Complement
[Bancilhon & Spyraøos, ACM TODS’81]

Given

\[ \text{get} :: S \rightarrow V \]

define a \( C \) and

\[ \text{res} :: S \rightarrow C \]
A Principled Approach: Constant-Complement
[Bancilhon & Spyropoulos, ACM TODS’81]

Given

\[
\text{get} :: S \rightarrow V
\]

define a \( C \) and

\[
\text{res} :: S \rightarrow C
\]

such that

\[
\text{paired} = \lambda s \rightarrow (\text{get} \ s, \text{res} \ s)
\]

is injective
A Principled Approach: Constant-Complement [Bancilhon & Spyrautos, ACM TODS’81]

Given

\[ \text{get} :: S \rightarrow V \]

define a \( C \) and

\[ \text{res} :: S \rightarrow C \]

such that

\[ \text{paired} = \lambda s \rightarrow (\text{get} \ s, \text{res} \ s) \]

is injective and has an inverse \( \text{inv} :: (V, C) \rightarrow S \).
A Principled Approach: Constant-Complement
[Bancilhon & Spyrtos, ACM TODS’81]

Given

\[ \text{get} :: S \rightarrow V \]

define a \( C \) and

\[ \text{res} :: S \rightarrow C \]

such that

\[ \text{paired} = \lambda s \rightarrow (\text{get} \ s, \text{res} \ s) \]

is injective and has an inverse \( \text{inv} :: (V, C) \rightarrow S \).

Then:

\[ \text{put} :: V \rightarrow S \rightarrow S \]

\[ \text{put} \ v' \ s = \text{inv} \ (v', \text{res} \ s) \]
A Principled Approach: Constant-Complement
[Bancilhon & Spyrratos, ACM TODS’81]

Given

\[ \text{get} :: S \rightarrow V \]

define a \( C \) and

\[ \text{res} :: S \rightarrow C \]

such that

\[ \text{paired} = \lambda s \rightarrow (\text{get} \ s, \text{res} \ s) \]

is injective and has an inverse \( \text{inv} :: (V, C) \rightarrow S \).

Then:

\[ \text{put} :: V \rightarrow S \rightarrow S \]

\[ \text{put} \ v' \ s = \text{inv} (v', \text{res} \ s) \]

has to be effective!
A Principled Approach: Constant-Complement

Guarantees “very-well-behavedness”:

- \( \text{put} \left( \text{get} \ s \right) s = s \)
- \( \text{get} \left( \text{put} \ v' \ s \right) = v' \)
- \( \text{put} \ v'' \left( \text{put} \ v' \ s \right) = \text{put} \ v'' \ s \)
A Principled Approach: Constant-Complement

Guarantees “very-well-behavedness”:

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

Example:

$\text{get} :: \text{Nat} \rightarrow \text{Nat}$

$\text{get} \ n = n \ '\text{div}' \ 2$
A Principled Approach: Constant-Complement

Guarantees “very-well-behavedness”:

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

Example:

- $\text{get} :: \text{Nat} \rightarrow \text{Nat}$
- $\text{res} :: \text{Nat} \rightarrow \text{Nat}_2$
- $\text{get} \ n = n \ '\text{div}' \ 2$
- $\text{res} \ n = n \ '\text{mod}' \ 2$

Other choices possible, and give different behavior
A Principled Approach: Constant-Complement

Guarantees “very-well-behavedness”:

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

Example:

\[
\begin{align*}
\text{get} &:: \text{Nat} \rightarrow \text{Nat} & \text{res} &:: \text{Nat} \rightarrow \text{Nat}_2 \\
\text{get } n & = n \ '\text{div}' \ 2 & \text{res } n & = n \ '\text{mod}' \ 2 \\
\text{inv} &:: (\text{Nat}, \text{Nat}_2) \rightarrow \text{Nat} \\
\text{inv } (v', c) & = 2 \ast v' + c 
\end{align*}
\]
A Principled Approach: Constant-Complement

Example:

\[
\begin{align*}
\text{get} &:: \text{Nat} \to \text{Nat} \\
\text{get } n &= n \div 2 \\
\text{res} &:: \text{Nat} \to \text{Nat}_2 \\
\text{res } n &= n \mod 2 \\
\text{inv} &:: (\text{Nat}, \text{Nat}_2) \to \text{Nat} \\
\text{inv } (v', c) &= 2 \times v' + c
\end{align*}
\]
A Principled Approach: Constant-Complement

Example:

\[
\begin{align*}
\text{get} \colon \text{Nat} &\rightarrow \text{Nat} & \text{res} \colon \text{Nat} &\rightarrow \text{Nat}_2 \\
\text{get } n &= n \div 2 & \text{res } n &= n \mod 2
\end{align*}
\]

\[
\begin{align*}
\text{inv} \colon (\text{Nat}, \text{Nat}_2) &\rightarrow \text{Nat} \\
\text{inv } (v', c) &= 2 \times v' + c
\end{align*}
\]

Then:

\[
\begin{align*}
\text{put} \colon \text{Nat} &\rightarrow \text{Nat} \rightarrow \text{Nat} \\
\text{put } v' s &= \text{inv } (v', \text{res } s)
\end{align*}
\]
A Principled Approach: Constant-Complement

Example:

\[
\begin{align*}
\text{get} & : \text{Nat} \rightarrow \text{Nat} & \text{res} & : \text{Nat} \rightarrow \text{Nat}_2 \\
\text{get } n & = n \div 2 & \text{res } n & = n \mod 2 \\
\text{inv} & : (\text{Nat}, \text{Nat}_2) \rightarrow \text{Nat} \\
\text{inv } (v', c) & = 2 \times v' + c
\end{align*}
\]

Then:

\[
\begin{align*}
\text{put} & : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \\
\text{put } v' s & = \text{inv } (v', \text{res } s) \\
& = 2 \times v' + s \mod 2
\end{align*}
\]
A Principled Approach: Constant-Complement

Example:

\[
\begin{align*}
\text{get} & \colon \text{Nat} \to \text{Nat} & \text{res} & \colon \text{Nat} \to \text{Nat}_2 \\
g \text{get } n = n \div 2 & & r \text{es } n = n \mod 2
\end{align*}
\]

\[
\begin{align*}
\text{inv} & \colon (\text{Nat}, \text{Nat}_2) \to \text{Nat} \\
\text{inv } (v', c) = 2 \ast v' + c
\end{align*}
\]

Then:

\[
\begin{align*}
\text{put} & \colon \text{Nat} \to \text{Nat} \to \text{Nat} \\
\text{put } v' s = \text{inv } (v', \text{res } s) \\
& = 2 \ast v' + s \mod 2
\end{align*}
\]
Automatic Bidirectionalization by Example

Let:

\[
\text{get} :: [\alpha] \rightarrow [\alpha]
\]

\[
\text{get} \ [()] = []
\]

\[
\text{get} \ [x] = []
\]

\[
\text{get} \ (x : y : zs) = y : (\text{get} \ zs)
\]
Automatic Bidirectionalization by Example

Let:

\[ \text{get} :: [\alpha] \rightarrow [\alpha] \]
\[ \text{get} [\ ] = [] \]
\[ \text{get} [x] = [] \]
\[ \text{get} (x : y : zs) = y : (\text{get} \ zs) \]

A syntactically derived complement function:

\[ \text{res} [\ ] = C_1 \]
\[ \text{res} [x] = C_2 \cdot x \]
\[ \text{res} (x : y : zs) = C_3 \cdot x \cdot (\text{res} \ zs) \]
Automatic Bidirectionalization by Example

A syntactically derived complement function:

\[
\begin{align*}
\text{res} \; [] & = C_1 \\
\text{res} \; [x] & = C_2 \; x \\
\text{res} \; (x : y : zs) & = C_3 \; x \; (\text{res} \; zs)
\end{align*}
\]

Syntactic pairing:

\[
\begin{align*}
\text{paired} \; [] & = ([], C_1) \\
\text{paired} \; [x] & = ([], C_2 \; x) \\
\text{paired} \; (x : y : zs) & = (y : v, C_3 \; x \; c) \\
\text{where} \; (v, c) & = \text{paired} \; zs
\end{align*}
\]
Automatic Bidirectionalization by Example

Syntactic pairing:

\[
\begin{align*}
\text{paired} & \, [\,] & = (\, [], \, C_1) \\
\text{paired} & \, [\, x] & = (\, [], \, C_2 \, x) \\
\text{paired} & \, (x : y : z) & = (\, y : v, \, C_3 \, x \, c) \\
\end{align*}
\]

where \((v, c) = \text{paired} \, z\)

Syntactic inversion:

\[
\begin{align*}
\text{inv} & \, (\, [], \, C_1) & = [\,] \\
\text{inv} & \, (\, [], \, C_2 \, x) & = [\, x] \\
\text{inv} & \, (y : v, \, C_3 \, x \, c) & = x : y : z \\
\end{align*}
\]

where \(z = \text{inv} \, (v, c)\)
Automatic Bidirectionalization by Example

Syntactic inversion:

\[
\begin{align*}
\text{inv} ([], C_1) &= [] \\
\text{inv} ([], C_2 x) &= [x] \\
\text{inv} (y : v, C_3 x c) &= x : y : zs \\
\text{where } &zs = \text{inv} (v, c)
\end{align*}
\]

Then,

\[
\text{put } v' s = \text{inv} (v', \text{res } s)
\]
Automatic Bidirectionalization by Example

Syntactic inversion:

\[
\begin{align*}
\text{inv} ([] \ , \ C_1) & = [] \\
\text{inv} ([] \ , \ C_2 \ x) & = [x] \\
\text{inv} (y : v, \ C_3 \ x \ c) & = x : y : zs \\
\text{where} \ zs & = \text{inv} (v, c)
\end{align*}
\]

Then,

\[
\text{put} \ v' \ s = \text{inv} (v', \ \text{res} \ s)
\]

corresponds to (the earlier seen):

\[
\begin{align*}
\text{put} \ [] & = [] \\
\text{put} \ [] \ [x] & = [x] \\
\text{put} (y' : v') \ (x : y : zs) & = x : y' : (\text{put} \ v' \ zs)
\end{align*}
\]
Automatic Bidirectionalization

Syntactic Bidirectionalization

[Matsuda et al., ICFP’07]
Automatic Bidirectionalization

source

\[ s \]

\[ s' \]

\[ \text{get} \]

\[ v \]

\[ v' \]

\[ \text{put} \]

view

Semantic Bidirectionalization

[V., POPL’09]
Semantic Bidirectionalization

Aim: Write higher-order function $\text{bff}^\dagger$ such that any $\text{get}$ and $\text{bff get}$ satisfy $\text{GetPut}$, $\text{PutGet}$, . . . .

$\dagger$ “Bidirectionalization for free!”
**Semantic Bidirectionalization**

**Aim:** Write higher-order function \( \text{bff} \) such that any \( \text{get} \) and \( \text{bff get} \) satisfy GetPut, PutGet, ....

**Examples:**

```
"abc"    tail    "bc"
```

\( \text{† "Bidirectionalization for free!"} \)
Semantic Bidirectionalization

**Aim:** Write higher-order function $\text{bff}^\dagger$ such that any $\text{get}$ and $\text{bff get}$ satisfy $\text{GetPut}$, $\text{PutGet}$, . . . .

**Examples:**

```
"abc"  tail  "bc"
```

```
"ade"  tail  update  "de"
```
Semantic Bidirectionalization

Aim: Write higher-order function $\text{bff}^\dagger$ such that any get and bff get satisfy GetPut, PutGet, . . . .

Examples:

```
"abc"  ---\overrightarrow{\text{tail}}---  "bc"
      \overleftarrow{\text{update}}

"ade"  \overleftarrow{\text{bff\ tail}}\overrightarrow{\text{tail}}\overrightarrow{\text{update}}\overleftarrow{\text{update}}
```
Semantic Bidirectionalization

Aim: Write higher-order function $\text{bff}^\dagger$ such that any $\text{get}$ and $\text{bff get}$ satisfy GetPut, PutGet, ....

Examples:

```
'abac'

flatten

"abac"
```

$\dagger$ “Bidirectionalization for free!”
Semantic Bidirectionalization

Aim: Write higher-order function $\text{bff}^\dagger$ such that any $\text{get}$ and $\text{bff get}$ satisfy $\text{GetPut, PutGet, ...}$

Examples:

- 'a' 'b' 'a' 'c'
- 'a' 'b' 'x' 'c'
- "abac"
- "abxc"

$\dagger$ "Bidirectionalization for free!"
Semantic Bidirectionalization

**Aim:** Write higher-order function $bff^\dagger$ such that any `get` and `bff get` satisfy GetPut, PutGet, ....

**Examples:**

```
'a' 'b' 'a' 'c'
```

```
'a' 'b' 'x' 'c'
```

```
'abac'
```

```
'abxc'
```

\[ ^\dagger \text{ "Bidirectionalization for free!" } \]
Analyzing Specific Instances

Assume we are given some

\[
\text{get} :: [\alpha] \rightarrow [\alpha]
\]

How can we, or \text{bff}, analyze it without access to its source code?
Analyzing Specific Instances

Assume we are given some

\[
\text{get} :: [\alpha] \rightarrow [\alpha]
\]

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

**Idea:** How about applying \text{get} to some input?
Analyzing Specific Instances

Assume we are given some

\[
\text{get} :: [\alpha] \rightarrow [\alpha]
\]

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

Idea: How about applying get to some input?

Like:

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

Indeed, this gives us traceability for free:
Analyzing Specific Instances

Like:

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

Indeed, this gives us traceability for free:

![Diagram showing list operations]
Analyzing Specific Instances

Like:

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

Indeed, this gives us traceability for free:
Analyzing Specific Instances

Like:

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

Indeed, this gives us traceability for free:

![Diagram showing tail and init operations on lists](image)
Analyzing Specific Instances

Like:

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

Indeed, this gives us traceability for free:

\[
\text{tail} \quad \text{and} \quad \text{init}
\]

and
Analyzing Specific Instances

Like:

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

Indeed, this gives us traceability for free:

\[
\text{tail}
\]

and

\[
\text{init}
\]

and

\[
\text{get}
\]
Analyzing Specific Instances

Like:

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

Indeed, this gives us traceability for free:

Then transfer the gained insights to arbitrary lists!
Analyzing Specific Instances

Like:

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

Indeed, this gives us traceability for free:

Then transfer the gained insights to arbitrary lists!
Semantic Bidirectionalization by Example

'\textbf{b}' 'a' 'c' 'a' \xrightarrow{\text{\texttt{tail}} \circ \text{\texttt{flatten}}} \text{"aca"}

\text{update}

bff (\text{\texttt{tail}} \circ \text{\texttt{flatten}}) \xrightarrow{?} \text{"xca"}
Semantic Bidirectionalization by Example

?- bff (tail ◦ flatten) v' "xca"
Semantic Bidirectionalization by Example

```
  t
 / \      \\
1   2   3   4
      \    \\
     s    \\
      \  \\
     'b' 'a' 'c' 'a'
```

```
1 → 'b'
2 → 'a'
3 → 'c'
4 → 'a'
```

```
1 → 'b'
2 → 'x'
3 → 'c'
4 → 'a'
```

```
2 → 'x'
3 → 'c'
4 → 'a'
```

```
"xca"'
```
Semantic Bidirectionalization by Example

```
1 → 'b'
2 → 'a'
3 → 'c'
4 → 'a'
```

```
1 → 'b'
2 → 'x'
3 → 'c'
4 → 'a'
```

```
"xca" \[\text{v'}\]
```
Semantic Bidirectionalization by Example

\[
\begin{align*}
1 & \rightarrow 'b' \\
2 & \rightarrow 'a' \\
3 & \rightarrow 'c' \\
4 & \rightarrow 'a'
\end{align*}
\]

\[
\begin{align*}
2 & \rightarrow 'x' \\
3 & \rightarrow 'c' \\
4 & \rightarrow 'a'
\end{align*}
\]

\[
\text{"xca"}_\neg
\]

\[
\text{tail} \circ \text{flatten}
\]

\[
[2,3,4] \begin{array}{c}
\text{get t}
\end{array}
\]
Semantic Bidirectionalization by Example

\[
\begin{align*}
1 & \rightarrow 'b' \\
2 & \rightarrow 'a' \\
3 & \rightarrow 'c' \\
4 & \rightarrow 'a'
\end{align*}
\]

\[
\begin{align*}
2 & \rightarrow 'x' \\
3 & \rightarrow 'c' \\
4 & \rightarrow 'a'
\end{align*}
\]

"xc\text{a}"
Semantic Bidirectionalization by Example

\[
\begin{array}{c}
\text{tail} \circ \text{flatten} \\
[2,3,4]
\end{array}
\]
Semantic Bidirectionalization by Example

(1 \rightarrow 'b', 2 \rightarrow 'a', 3 \rightarrow 'c', 4 \rightarrow 'a')

2 \rightarrow 'x'
3 \rightarrow 'c'
4 \rightarrow 'a'

"xca"  \rightarrow \text{flatten}

\text{tail} \circ \text{flatten}

[2, 3, 4] \text{get } t
Semantic Bidirectionalization by Example

\[ S \]

\[ \text{'b' 'a' 'c' 'a'} \]

\[ \text{'b' 'x' 'c' 'a'} \]

\[ \text{bff (tail \circ flatten)} \]

\[ \text{''xca''} \]
Semantic Bidirectionalization by Example

In the diagram, the tree structure is shown with the following labels:

- The tree is labeled with the sequence 'b' 'a' 'c' 'a'.
- The tree includes a node labeled 'x'.

The diagram illustrates the process of semantic bidirectionalization, which involves transforming the tree and its labels according to specific rules. The transformed sequence is shown as 'aca'.

The notation 'tail ◦ flatten' refers to the process of flattening the tree and then taking the tail of the resulting sequence.

The final output is the sequence [2, 3, 4], which represents the transformed version of the original input.
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
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

F. Bancilhon and N. Spyratos.
Update semantics of relational views.

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.

S. Katayama.
Systematic search for lambda expressions.


J. Voigtländer.
Bidirectionalization for free!