Bidirectional Transformations
—
a PL perspective

BIRS meeting on BX, 2013
Bidirectional Transformations (BX)

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

\[ a_1 \rightarrow b_1 \]

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

\[ a_1 \xrightarrow{\text{to}} b_1 \]

Unless bijective, typically additional information needed/useful:

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

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

unless bijective, typically additional information needed/useful:

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

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

\[ a_1 \rightarrow b_1 \]
\[ a_2 \leftarrow b_2 \]
\[ a_3 \rightarrow b_3 \]

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
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
Objectives for this Talk

- get everybody into "BX mode" for the week
- set out basic premises of the PL approach, paradigmatic problems
- introduce terminology and semantic principles
- no details of specific solutions
- relate to what "we" think is solved and what not
- open discussion
What’s specific about “the PL approach”, anyway?

- focus on the transformations/functions themselves, not so much on the data
- focus on extensional semantics and laws
- correctness by construction/derivation (as opposed to a-posteriori verification)
- assuming a very clean setting (naive?)
- being driven by our favourite new PL techniques
- typically, algebraic data domains
What’s specific about “the PL approach”, anyway?

- focus on the transformations/functions themselves, not so much on the data
What’s specific about “the PL approach”, anyway?

- focus on the transformations/functions themselves, not so much on the data
- focus on extensional semantics and laws
What’s specific about “the PL approach”, anyway?

- focus on the transformations/functions themselves, not so much on the data
- focus on extensional semantics and laws
- correctness by construction/derivation (as opposed to a-posteriori verification)
What’s specific about “the PL approach”, anyway?

- focus on the transformations/functions themselves, not so much on the data
- focus on extensional semantics and laws
- correctness by construction/derivation (as opposed to a-posteriori verification)
- assuming a very clean setting (naive?)
What’s specific about “the PL approach”, anyway?

- focus on the transformations/functions themselves, not so much on the data
- focus on extensional semantics and laws
- correctness by construction/derivation (as opposed to a-posteriori verification)
- assuming a very clean setting (naive?)
- being driven by our favourite new PL techniques
What’s specific about “the PL approach”, anyway?

- focus on the transformations/functions themselves, not so much on the data
- focus on extensional semantics and laws
- correctness by construction/derivation (as opposed to a-posteriori verification)
- assuming a very clean setting (naive?)
- being driven by our favourite new PL techniques
- typically, algebraic data domains
Bidirectional Transformations

unless bijective, typically additional information needed/useful:

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

\[ a_1 \xrightarrow{\Delta} b_1 \xleftarrow{\Delta} a_2 \xrightarrow{\text{to}} b_2 \xleftarrow{\text{from}} \]

Focus on:
- single-side updates
- one-step updates
Bidirectional Transformations

also, focus on asymmetric setting:

- to usually non-injective, henceforth called get
- from then called put, definitely needs extra info
- for simplicity, state-based
- “sources” and “views”

focus on:

- single-side updates
- one-step updates
Bidirectional Transformations

also, focus on asymmetric setting:

- to usually non-injective, henceforth called get
- from then called put, definitely needs extra info
- for simplicity, state-based
- “sources” and “views”

focus on:

- single-side updates
- one-step updates
Bidirectional Transformations

A closer look at representing $S \rightarrow V$ connections. For example:

$$\text{get}$$

```
\begin{array}{c}
x \\
y \\
z \\
u \\
v
\end{array} \quad \begin{array}{c}
y \\
z \\
u \\
v
\end{array}
```
Bidirectional Transformations

A closer look at representing \( S \rightarrow V \) connections. For example:

```
get

x y z u v
```

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

Because of:

```
7 - \frac{25}{29}
```
Bidirectional Transformations

A closer look at representing connections. For example:

\[x \quad y \quad z \quad u \quad v\]

get

or

\[x \quad y \quad z \quad u \quad v\]

get

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

Because of:

\[x \quad y \quad z \quad u \quad v\]
Bidirectional Transformations

A closer look at representing connections. For example:

\[
\begin{align*}
\text{get} & \quad x & y & z & u & v \\
\end{align*}
\]

or

\[
\begin{align*}
\text{get} & \quad x & y & z & u & v \\
\end{align*}
\]

or

\[
\begin{align*}
\text{get} & \quad x & y & z & u & v \\
\end{align*}
\]

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

Because of:

\[
\begin{align*}
\end{align*}
\]
Bidirectional Transformations

A closer look at representing \( S \rightarrow V \) connections. For example:

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

A closer look at representing connections. For example:

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

Because of:
Bidirectional Transformations

Some further relevant aspects:

▶ What artifacts need to be specified?
  ▶ both get and put
  ▶ only one of them, the other derived
  ▶ a more abstract artifact, 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?
Properties / Laws
Bidirectional Transformations

Specific asymmetric setting, state-based:

\[ \text{source} \quad s \quad \text{view} \quad v \]

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

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

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

Specific asymmetric setting, state-based:

```
source
s

get :: S → V
put :: S → V → S

view
v

update

s'

v'
```
About Behavior under No-Change

To prevent this, the GetPut law:

\[ \text{put} \left( \text{get} \right) = \text{s} \]

NB: For this, put must be surjective.

Principle: If the view does not change, neither should the source.
To prevent this, the GetPut law:

put(s) = s

NB: For this, put must be surjective.

Principle: If the view does not change, neither should the source.
About Behavior under No-Change

To prevent this, the GetPut law:
\[
\text{put}(\text{get}) = s_{NB}
\]

NB: For this, put must be surjective.

Principle: If the view does not change, neither should the source.
About Behavior under No-Change

To prevent this, the GetPut law:

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

NB: For this, \text{put} must be surjective.
About Behavior under No-Change

Principle: If the view does not change, neither should the source.
About Behavior under No-Change

To prevent this, the GetPut law:

\[ \text{put } s \ (\text{get } s) = s \]
About Behavior under No-Change

To prevent this, the GetPut law:

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

NB: For this, \text{put} must be surjective.
About Preservation of Changes

Project out string component

| foo | 0 |

Principle: Updates should be translated exactly.
About Preservation of Changes

To prevent this, the PutGet law:

\[ \text{get}(\text{put}(s, v)) = v \]

NB: For this, \( \text{put}s \) must be injective for every \( s \).

Principle: Updates should be translated exactly.
About Preservation of Changes

To prevent this, the PutGet law:

\[
\text{get} \left( \text{put} s v \right) = v
\]

NB: For this, put \( s \) must be injective for every \( s \).

Principle: Updates should be translated exactly.

\[
\text{return a constant}
\]
About Preservation of Changes

To prevent this, the PutGet law:

\[ \text{get}(\text{put}s \, v) = v \]

NB: For this, \( \text{put}s \) must be injective for every \( s \).

Principle: Updates should be translated exactly.
To prevent this, the PutGet law:

\[
\text{get} \ (\text{put} \ s \ v) = v
\]
To prevent this, the PutGet law:

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

NB: For this, \text{put } s must be injective for every \text{ } s.
Somewhat more Challenging

If we want to allow such behavior, we need to weaken the PutGet law (and people have done so).

project out and duplicate string component
Somewhat more Challenging

If we want to allow such behavior, we need to weaken the PutGet law (and people have done so).
If we want to allow such behavior, we need to weaken the PutGet law (and people have done so).
If we want to allow such behavior, we need to weaken the PutGet law (and people have done so).
If we want to allow such behavior, we need to weaken the PutGet law (and people have done so).
About Consistent Composition

To prevent this, the PutPut law:

\[
\text{put}(\text{put}(s \ v)) = s \ v' = \text{put}(s \ v')
\]
About Consistent Composition

To prevent this, the PutPut law:

\[ \text{put}(\text{put}(v)) = v' \]
About Consistent Composition

To prevent this, the PutPut law:

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

increment numeric component
if string component has changed
To prevent this, the PutPut law:
\[
\text{put}(\text{put}(s, v)) = \text{put}(s, \text{put}(v, 1)) = \text{put}(\text{put}(s, 0), 1)
\]
About Consistent Composition

translated updates produce "side effects" on source
About Consistent Composition

To prevent this, the PutPut law:

\[\text{put}(\text{put}(s \; v)) \; v' = \text{put}(s \; v')\]

\[14 - \frac{57}{59}\]
About Consistent Composition

To prevent this, the PutPut law:

\[
\text{put}(\text{put}(s \, v) \, v) = \text{put}(s \, v')
\]
To prevent this, the PutPut law:

\[
\text{put} \ (\text{put} \ s \ v) \ v' = \text{put} \ s \ v'
\]
Less Debatable

Actually a consequence of GetPut and PutGet, the PutTwice law:

\[
\text{put } (\text{put } s \ v) \ v = \text{put } s \ v
\]
Actually a consequence of GetPut and PutGet, the PutTwice law:

\[
\text{put} \ (\text{put} \ s \ v) \ v = \text{put} \ s \ v
\]

We’ll get back to this property in a moment.
Ambiguity of put
How many puts are there?

Due to non-injectivity, get can map many objects from $S$ onto the same object from $V$. 
How many puts are there?

In essence, get projects out part of the information in the source object...
How many puts are there?

In essence, get projects out part of the information in the source object... and throws away the rest.
How many puts are there?

After an update,
How many puts are there?

After an update, the “view part” of the new source object is fixed by PutGet...
After an update, the “view part” of the new source object is fixed by PutGet... and if the lens obeys PutPut, the “projected away part” is fixed to be exactly the one from the original source.
How many puts are there?

After an update, the “view part” of the new source object is fixed by PutGet... and if the lens obeys PutPut, the “projected away part” is fixed to be exactly the one from the original source.

Even this doesn’t mean that there is only exactly one “very well-behaved” put per get!
How many puts are there?

Moreover, if the lens *doesn’t* need to obey PutPut, then the behavior of *put* is much less constrained... ...and there are even more puts to choose from!
How many puts are there?

Moreover, if the lens *doesn’t* need to obey PutPut, then the behavior of put is much less constrained. . .

. . . and there are even more puts to choose from!

So, definitely need extra information to select one.
On the Other Hand... 

... there is only one get per “well-behaving” put!
On the Other Hand . . .

. . . there is only one get per “well-behaving” put!

Specifically, if put is surjective, is injective for every s, and satisfies PutTwice, then there is exactly one get such that the two together satisfy GetPut and PutGet.
On the Other Hand... 

... there is only one get per “well-behaving” put!

Specifically, if put is surjective, is injective for every $s$, and satisfies PutTwice, then there is exactly one get such that the two together satisfy GetPut and PutGet. And, there are equivalent, even nicer conditions formulated just in terms of put as well.

[Fischer, Hu, Pacheco]
On the Other Hand...

... there is only one get per “well-behaving” put!

Specifically, if put is surjective, is injective for every s, and satisfies PutTwice, then there is exactly one get such that the two together satisfy GetPut and PutGet. And, there are equivalent, even nicer conditions formulated just in terms of put as well.

[Fischer, Hu, Pacheco]

There are even first concrete bidirectionalization techniques derived from this put-based approach!
Conclusion / Discussion (?)
“Solved”

- a lot of very nice definitive work on semantics
- successful methods for automatic derivation of reasonable put- from get-functions on strings, trees, and graphs (?)
  - combinator languages with powerful type systems
  - program transformations based on constant-complement
  - query languages with automatic tracing
  - grammar-based approaches
Open Problems

Leaving the academic niche:

► “How to deliver BX to the masses? Some effective way to integrate BX with existing general programming languages would be nice. Most tools/languages are very academic, and I don’t see them being used for industrial case studies...”

► “But I think to really achieve world domination, a BX framework will need to make substantial progress on having an attractive and intuitive front-end.”
Open Problems

Tackling ambiguities effectively:

▶ “Can we design a declarative language that can be used to describe any intentional bidirectional behavior (i.e., have full control of bidirectional behavior)?”

▶ “We still lack effective, intuitive (user-friendly) and generic mechanisms to tame the non-determinism of backwards transformation.”

▶ “Ability to control the choice between multiple valid backward transformation results. [...] clarify to what extent user can control by writing different get (forward) transformations.”
Open Problems

Handling richer semantic domains:

► “[. . . ] still no effective solution for non-tree shaped domains.”

► “Bx on ordered graphs (outgoing edges are ordered) and graphs in which ordered and unordered edges are mixed.”

► “Handling of constraints over the domains (that is, handling non CFG-like domains). DB people have some work on this (handling keys, functional dependencies, inclusion dependencies, etc), but the issue seems ignored by PL people.”
“Conclusion”

There is a lot of potential and possible inspiration from PL land for the general area of BX.

Challenges remain:

▶ scaling up in every way
▶ providing control over nondeterminism
▶ capturing user/programmer intentions
▶ handling richer structures/domains
▶ running efficiently