On the Laws of Bidirectional Transformations

Sebastian Fischer

joint work with Zhenjiang Hu
& Hugo Pacheco

The solutions are all simple...
after you have arrived at them.

— Robert M. Pirsig

in

Zen

and the Art of

Motorcycle Maintenance

(1974)

Question:

What is the essence of Bidirectional Transformations?

What **exactly** do we need to define in order to specify BX?

Answer:

It depends!

(on what we mean by BX)

Bidirectional Transformations
(or BX)

retrieves some information

get :: Source → View

put :: Source → View → Source

restores (possibly updated) information

BX

Not every combination of get and put functions is reasonable.

Different classes of BX are defined using different sets of laws.

BX

Laws introduce redundancy

What part, exactly, of the definitions of get and put is redundant?

Example

Say, sauce is characterized by color and spicyness.

```
data Sauce = Sauce
  { color :: Color,
    spicyness :: Spicyness }
```

- get function retrieves color:

```
getColor :: Sauce -> Color
getColor s = color s
```

- put function updates color:

```
putColor :: Sauce -> Color -> Sauce
putColor s c =
    s { color = c }
```

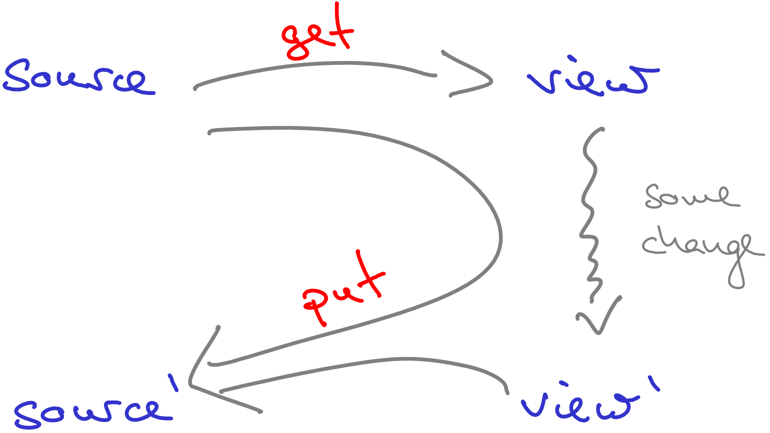
```
sauce = Sauce { color = "red",
                spicyness = 5 }

getColor sauce → "red"
putColor sauce "green"
  → Sauce { color = "green",
            spicyness = 5 }
```

spicyness stays the same

=

9

BX



Well-Behaved BX

PutGet law:

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

(for all sources s, views v)

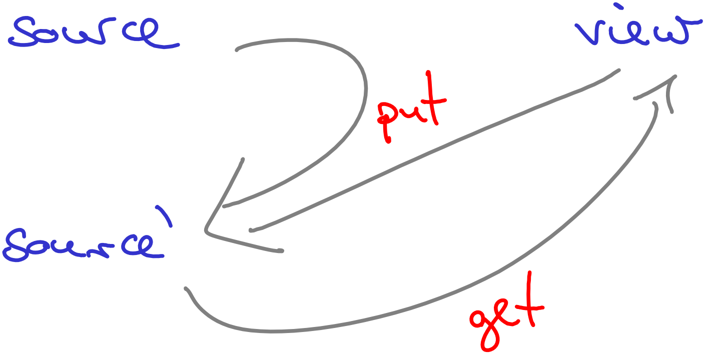
GetPut law:

$$\mathrm{put}\ s\ (\mathrm{get}\ s) = s$$

(for all sources s)

PutGet

$$get\ (put\ s\ v) = v$$



PutGet

getColor (putColour source "green")

=

color (source { color = "green" })

=

"green"

GetPut

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

GetPut

$$\text{putColor source (getColor source)}$$

$$=$$

$$\text{source \{color = color source\}}$$

$$=$$

$$\text{source}$$

PutTwice

implied by PutGet and GetPut

same view

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

$$=$$ because of PutGet

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

$$=$$ because of GetPut

$$\text{put}\ s\ v$$

PutTwice

putColor (putColor sauce "green") "green"

= (sauce {color = "green"})
{color = "green"}

= sauce {color = "green"}

= putColor sauce "green"

Injectivity

A function f is injective iff it has a left inverse g :

$$g \cdot f = id$$

left inverse

injective

Injectivity

Example: $\quad get\ (put\ s\ v) = v$

$\Updownarrow$

put s maps different views to different sources $\quad get \circ put\ s = id$

→ PutGet implies that put s is injective for all sources s

Injectivity

getColor is left inverse of putColor s

getColor (putColor s c)

=

color (s { color = c })

=

c

Surjectivity

A function f is surjective iff it has a right inverse g:

$$f \circ g = \mathrm{id}$$

surjective

right inverse

Surjectivity

Example:

$$\mathrm{get}\,(\mathrm{put}\ s\ v) = v$$

get can yield every view

$$\Updownarrow$$

$$\mathrm{get} \cdot \mathrm{put}\ s = \mathrm{id}$$

→ PutGet implies that get is surjective

Intuitively, these injectivity and surjectivity properties mean that:

- the source type is at least as big as the view type
- every view can be embedded into every source and can be retrieved without loss of information

Currying

$$put :: Source \to (View \to Source)$$

$$put\ s\ v = (put\ s)\ v$$

put takes source
and yields function on views

$$put\ s :: View \to Source$$

Currying

$$\text{curry } f\ x\ y = f\ (x, y)$$

$$\text{uncurry } f\ (x, y) = f\ x\ y$$

$$\text{uncurry put} :: (\text{Source}, \text{View}) \to \text{Source}$$

uncurry put takes
pair of source and view

Currying and Surjectivity

put is (usually) not surjective because function type View → Source is (usually) bigger than type Source

$$put :: Source \to (View \to Source)$$

put (usually) cannot yield every function from View to Source

Currying and Surjectivity

But: GetPut law

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

implies that uncurry put is surjective

$$\text{uncurry put } (s, v) = \text{put } s \ v$$

put can yield every source when applied to appropriate source and view

Currying and Surjectivity

if GetPut law holds, then

$$(\lambda s \to (s, \mathit{get}\ s))$$

is right inverse of uncurry put :

$$
\begin{aligned}
& \mathit{uncurry}\ \mathit{put}\ ((\lambda s \to (s, \mathit{get}\ s))\ s) \\
&= \mathit{uncurry}\ \mathit{put}\ (s, \mathit{get}\ s) \\
&= \mathit{put}\ s\ (\mathit{get}\ s) \qquad \text{GetPut} \\
&= s
\end{aligned}
$$

Characterization

The GetPut and PutGet laws hold

$\Updownarrow$

1. The GetPut law holds and
2. The PutTwice law holds ($put\ (put\ s\ v)\ v = put\ s\ v$)
3. $put\ s$ is injective for all s

only mention put function

Characterization

The GetPut and PutGet laws hold

$$\Updownarrow$$

1. The PutGet law holds and
2. The PutTwice law holds and
3. uncurry put is surjective

→ only mention put function

Unique Specification

For a given function

put :: Source → View → Source

such that

1. The PutTwice law holds
2. put s is injective for all s
3. uncurry put is surjective

there is exactly one function get such that BX is well-behaved

Unique Specification

The necessary conditions on put are sufficient to define get uniquely

Definition of get is redundant

Unique Specification

get s = v

such that

put s v = s

Existence

Because uncurry put is surjective there is for all s a source s' and a view v such that:

$$s = \text{put } s' \ v$$

Existence

put s v

= (choice of s' and v)

put (put s' v) v

= (PutTwice law)

put s' v

= (choice of s' and v)

s

Uniqueness

Because put s is injective

$$\text{put } s\ v = s = \text{put } s\ v'$$

implies $v = v'$

Functional-Logic Programming

get s | put s v =:= s

= v

where

v free

valid Curry

Ambiguity of get

Another possible put function for

getColor :: Sauce -> Color :

```
putColorSpicy :: Sauce -> Color -> Sauce
putColorSpicy s c
    | color s == c  = s
    | otherwise     = s { color = c,
          spicyness = spicyness s + 1 }
```

Ambiguity of get

Different well-behaved BX can have the same get function

The get function is not sufficient to specify well-behaved BX

Conclusion

Well-behaved BX are completely determined by put function with

1. PutTwice law
2. put s injective for all s
3. uncurry put surjective

(All conditions are implied by well-behavedness → no restriction)