On the Laws of Bidirectional Transformations

Sebastian Fischer

joint work with Weijiang Hu
& Hugo Pacheco
The solutions are all simple...

after you have arrived at them.

— Robert H. 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)

get :: Source -> View

put :: Source -> View -> Source

retrieves some information
restores (possibly updated) information
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 spiciness.

data Sauce = Sauce
{ color :: Color,
  spiciness :: Spiciness }
- get function retrieves color:
  
  ```
  get Color :: Sauce → Color
  get Color s = color s
  ```

- put function updates color:

  ```
  put Color :: Sauce → Color → Sauce
  put Color s c = s { color = c }
  ```
sauce = Sauce { color = "red", spiciness = 5 }

getColor sauce \rightarrow "red"

putColor sauce "green"

\rightarrow Sauce { color = "green", spiciness = 5 }
Well-behaved BX

**PutGet law:**
\[
\text{get}(\text{put } s \nu) = \nu
\]
(for all sources \(s\), views \(\nu\))

**GetPut law:**
\[
\text{put } s \ (\text{get } s) = s
\]
(for all sources \(s\))
Put Get

\[ \text{get (put s r)} = r \]
\begin{align*}
\text{get Color} & \quad (\text{put Color sauce "green"}) \\
& = \\
\text{color} & \quad (\text{sauce} \{ \text{color = "green"} \}) \\
& = \\
& \quad "\text{green}" 
\end{align*}
Get Put

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

putColor sauce (getColor sauce)

= sauce { color = getColor sauce }

= Sauce
Put Twice

implied by PutGet and GetPut

\[ \text{put} (\text{put } s \text{ v}) \text{ } \checkmark \text{ same view } \]

= 

because of PutGet

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

= 

because of GetPut

\[ \text{put } s \text{ v} \]
Put Twice

\[ \text{putColor} \left( \text{putColor} \text{ sauce "green"} \right) \text{ "green"} \]

\[ = \left( \text{sauce} \{ \text{color} = \text{"green"} \} \right) \]
\[ \{ \text{color} = \text{"green"} \} \]

\[ = \text{sauce} \{ \text{color} = \text{"green"} \} \]

\[ = \text{putColor} \text{ sauce "green"} \]
Injectivity

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

\[ g \cdot f = \text{id} \]
Example: \( \text{get} \circ (\text{put} \circ \text{get}) = \text{id} \)

- \( \text{put} \circ \text{s} \) maps different views to different sources.
- \( \text{get} \circ \text{put} \circ \text{s} = \text{id} \)

\( \Rightarrow \) \( \text{put} \circ \text{get} \) implies that \( \text{put} \circ \text{s} \) is injective for all sources \( \text{s} \).
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 \( g \):

\[ f \circ g = \text{id} \]
Surjectivity

Example: \( \text{get} \circ \text{put} \circ \text{r} = \text{id} \)

\( \text{get} \) can yield every view

\( \Rightarrow \) \( \text{PutGet} \) implies that \( \text{get} \) 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

\[ \text{put} :: \text{Source} \rightarrow (\text{View} \rightarrow \text{Source}) \]

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

\[ \text{put takes source and yields function on views} \]

\[ \text{put} \ s :: \text{View} \rightarrow \text{Source} \]
Currying

curly f \times 7 = f (x, 7)

uncurry f (x, 7) = f \times 7

uncurry put :: (Source, View) \rightarrow Source

uncurry put takes a 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 → (View → 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 \( \text{GetPut} \) law holds, then

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

is right inverse of uncurry put:

\[
\text{uncurry put } ((\lambda s \rightarrow (s, \text{get}s)) \ s)
\]

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

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

\[
= \text{GetPut}
\]

\[
= s
\]
Characterization

The GetPut and PutGet laws hold

\[ \Rightarrow \]

1. The GetPut law holds and

2. The PutTwice law holds \((\text{put} \, (\text{put} \, s \, v) \, v = \text{put} \, s \, v)\)

3. put \, s \, is injective for all \, s

(only mention put function)
Characterization

The GetPut and PutGet laws hold

\[ \iff \]

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

\[ \text{put} : \text{Source} \rightarrow \text{View} \rightarrow \text{Source} \]

such that

1. The Put twice law holds
2. put $s$ is injective for all $s$
3. uncurry put is surjective

there is exactly one function get

such that $\exists X$ is well-behaved
Unique Specification

The necessary conditions on put are sufficient to define get uniquely.

Definition of get is redundant
Unique Specification

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

such that

\[ \text{put } s \sqrt{ } = s \]
Existence

Because uncurry put is surjective, there is for all $s$ a source $s'$ and a view $\nu$ such that:

$$s = \text{put } s' \circ \nu$$
Existence

\[ \text{put } s \land \phi = (\text{choice of } s' \land \phi) \]
\[ \text{put } (\text{put } s' \land \phi) \land \phi = (\text{Put Twice law}) \]
\[ \text{put } s' \land \phi \]
\[ = (\text{choice of } s' \land \phi) \]
\[ s' \land \phi \]
Uniqueness

Because \( \text{put } s \) is injective

\[ \text{put } s \cdot v = s = \text{put } s \cdot v' \]

implies \( v = v' \)
Functional-Logic Programming

g s | put s v =:= s = v

where v free

valid Curry
Ambiguity of \texttt{get}

Another possible \texttt{put} function for \texttt{getColor} :: \texttt{Sauce} \to \texttt{Color} :

\begin{verbatim}
getColorSpicy :: \texttt{Sauce} \to \texttt{Color} \to \texttt{Sauce}
getColorSpicy s c
  | color s == c = s
  | otherwise = s \{ color = c, spicyness = spicyness s + 1 \}
\end{verbatim}
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 $S_X$ are completely determined by put function with

1. Put Twice law
2. $\text{put } s$ injective for all $s$
3. Uncurry $\text{put}$ surjective

(all conditions are implied by well-behavedness $\Rightarrow$ no restriction)