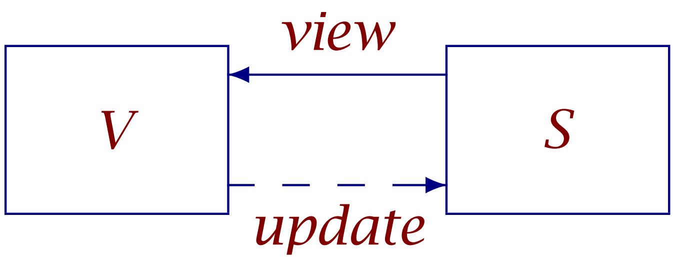
Notions of Bidirectional Computation and Entangled State Monads

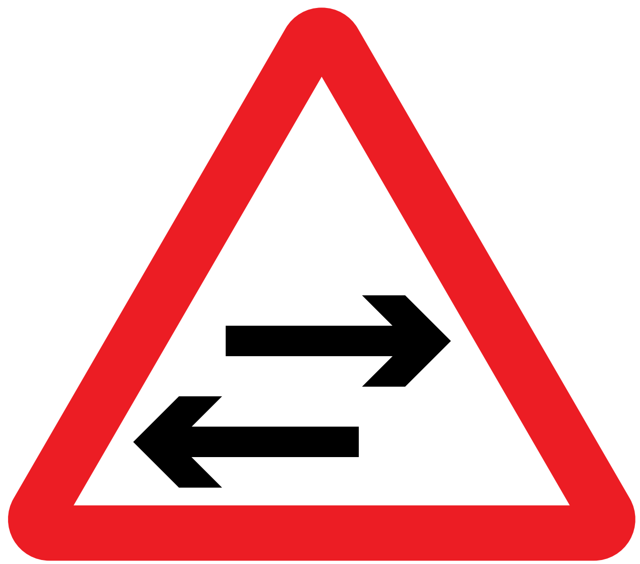
Faris Abou-Saleh, James Cheney, Jeremy Gibbons,
James McKinna, Perdita Stevens
SSBX, Oxford, July 2016

1. Bidirectional transformations (BX)

- *view–update* problem in databases

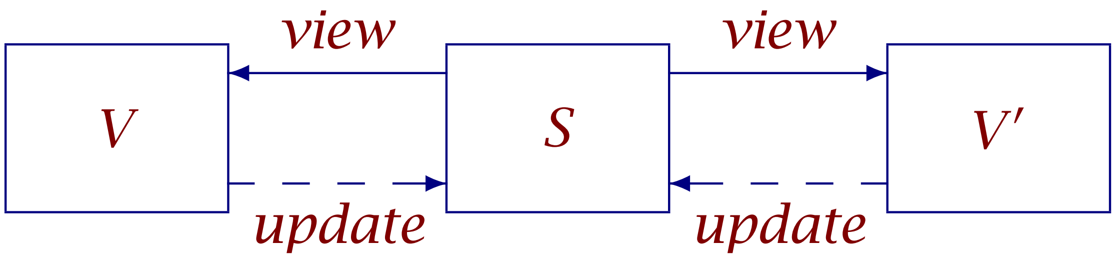


- *round-tripping* laws for consistency

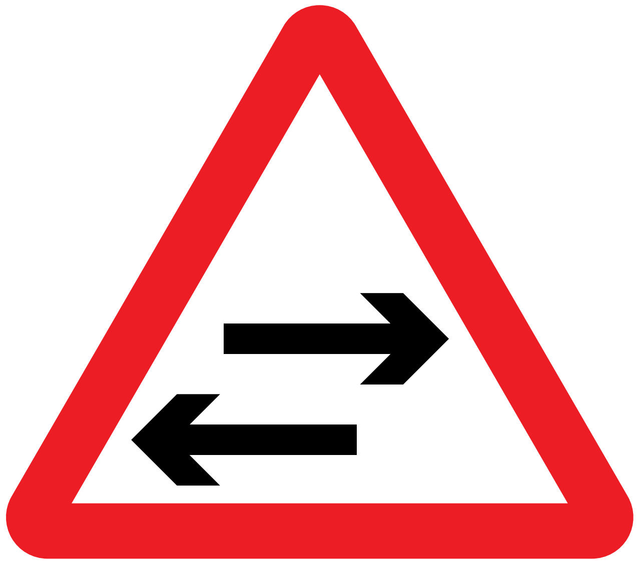


1.1. Symmetrize

- *view-update* problem in databases

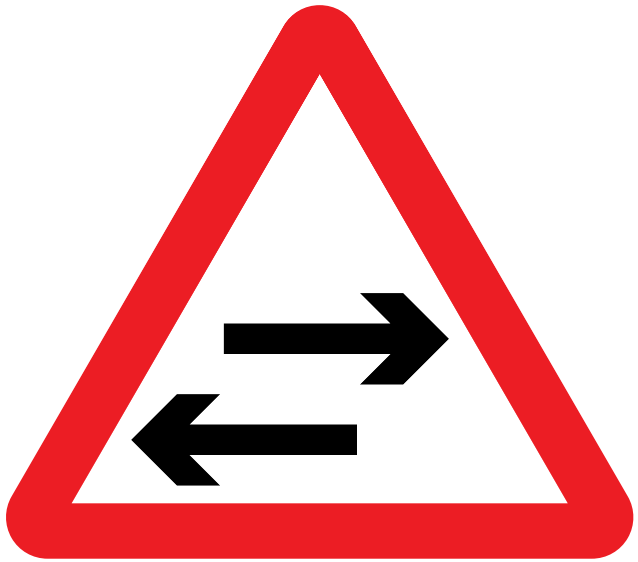


- *round-tripping* laws for consistency
- *symmetrize*—neither data source definitive
- *applications* in interactive programs, model-driven engineering. . .



1.2. Overview of talk

- *lenses* for BX
- BX is *inherently stateful*
- consistency maintenance implies *entangled state*
- combining with *other effects*, eg exceptions, non-determinism, I/O
- *composing* BX



2. Lenses for BX (Foster, Pierce, et al.)

An asymmetric *lens* $l : A \rightsquigarrow B$ from source A to view B is captured by

$$\begin{aligned} \textbf{data}\ Lens\ \alpha\ \beta = Lens\ \{ & view\ \ \ \ :: \alpha \to \beta, \\ & update :: \alpha \to \beta \to \alpha \} \end{aligned}$$

Round-tripping: say that $l :: Lens\ A\ B$ is *well-behaved* if

$$\begin{aligned} l.view\ (l.update\ s\ v) &= v \\ l.update\ s\ (l.view\ s) &= s \end{aligned}$$

and *very well-behaved* (rather a strong condition) if

$$l.update\ (l.update\ s\ v)\ v' = l.update\ s\ v'$$

2.1. Symmetric lenses (Hofmann, Pierce, Wagner)

More generally, neither data source need determine the other.

A *symmetric lens* $s : A \Leftrightarrow_C B$ between A and B, with *complements* of type C, is captured by

$$\mathbf{data}\ \mathit{SLens}\ \alpha\ \beta\ \gamma = \mathit{SLens}\ \{\ \mathit{putlr} :: (\alpha, \gamma) \to (\beta, \gamma),\ \mathit{putrl} :: (\beta, \gamma) \to (\alpha, \gamma)\ \}$$

Say that $s :: \mathit{SLens}\ A\ B\ C$ is *well-behaved* if

$$s.\mathit{putlr}\ (a, c) = (b, c') \Rightarrow s.\mathit{putrl}\ (b, c') = (a, c')$$
$$s.\mathit{putrl}\ (b, c) = (a, c') \Rightarrow s.\mathit{putlr}\ (a, c') = (b, c')$$

and in addition, *very well-behaved* (‘strong’) if

$$s.\mathit{putlr}\ (a, c) = (b, c') \Rightarrow s.\mathit{putlr}\ (a', c') = s.\mathit{putlr}\ (a', c)$$
$$s.\mathit{putrl}\ (b, c) = (a, c') \Rightarrow s.\mathit{putrl}\ (b', c') = s.\mathit{putrl}\ (b', c)$$

3. BX is effectful

Lenses involve 'reading' and 'writing': *impure*, with *computational effects*.

So let's look at the *state monad*:

$$\mathbf{data}\ State\ \sigma\ \alpha = State\ \{\ runState :: \sigma \to (\alpha, \sigma)\ \}$$

$$\begin{aligned}
&\mathbf{instance}\ Monad\ (State\ \sigma)\ \mathbf{where}\\
&\quad return\ a = State\ (\lambda s \to (a, s))\\
&\quad x \gg\!= k\ \ = State\ (\lambda s \to \mathbf{let}\ (a, s') = runState\ x\ s\\
&\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \ runState\ (k\ a)\ s')
\end{aligned}$$

with two additional operations, to *read* and *write* the state:

$$\begin{aligned}
&get\ \ \ :: State\ \sigma\ \sigma\\
&get\ \ \ = State\ (\lambda s \to (s, s))\\
&set\ \ \ :: \sigma \to State\ \sigma\ ()\\
&set\ s' = State\ (\lambda s \to ((), s'))
\end{aligned}$$

3.1. Equational theory of state

The *get* and *set* operations of the state monad satisfy four laws:

$$
\begin{aligned}
&\mathbf{do}\ \{ s \leftarrow \mathit{get}; s' \leftarrow \mathit{get}; \mathit{return}\ (s, s') \} &&= \mathbf{do}\ \{ s \leftarrow \mathit{get}; \mathit{return}\ (s, s) \} \\
&\mathbf{do}\ \{ \mathit{set}\ s; \mathit{get} \} &&= \mathbf{do}\ \{ \mathit{set}\ s; \mathit{return}\ s \} \\
&\mathbf{do}\ \{ s \leftarrow \mathit{get}; \mathit{set}\ s \} &&= \mathbf{do}\ \{ \mathit{return}\ () \} \\
&\mathbf{do}\ \{ \mathit{set}\ s; \mathit{set}\ s' \} &&= \mathbf{do}\ \{ \mathit{set}\ s' \}
\end{aligned}
$$

Indeed, the state monad is the *initial* model of this equational theory.

3.2. State with multiple components

One can generalise to several components; say, ‘left’ and ‘right’:

$$get_L :: State\ (\alpha, \beta)\ \alpha$$
$$get_R :: State\ (\alpha, \beta)\ \beta$$

$$set_L :: \alpha \rightarrow State\ (\alpha, \beta)\ ()$$
$$set_R :: \beta \rightarrow State\ (\alpha, \beta)\ ()$$

3.2. State with multiple components

One can generalise to several components; say, 'left' and 'right'...

The corresponding equational theory has four state laws on left:

$$
\begin{aligned}
&\mathbf{do}\ \{ s \leftarrow get_L; s' \leftarrow get_L; return\ (s, s') \} &&= \mathbf{do}\ \{ s \leftarrow get_L; return\ (s, s) \} \\
&\mathbf{do}\ \{ set_L\ s; get_L \} &&= \mathbf{do}\ \{ set_L\ s; return\ s \} \\
&\mathbf{do}\ \{ s \leftarrow get_L; set_L\ s \} &&= \mathbf{do}\ \{ return\ () \} \\
&\mathbf{do}\ \{ set_L\ s; set_L\ s' \} &&= \mathbf{do}\ \{ set_L\ s' \}
\end{aligned}
$$

another four on right:

$$
\begin{aligned}
&\mathbf{do}\ \{ s \leftarrow get_R; s' \leftarrow get_R; return\ (s, s') \} &&= \mathbf{do}\ \{ s \leftarrow get_R; return\ (s, s) \} \\
&\mathbf{do}\ \{ set_R\ s; get_R \} &&= \mathbf{do}\ \{ set_R\ s; return\ s \} \\
&\mathbf{do}\ \{ s \leftarrow get_R; set_R\ s \} &&= \mathbf{do}\ \{ return\ () \} \\
&\mathbf{do}\ \{ set_R\ s; set_R\ s' \} &&= \mathbf{do}\ \{ set_R\ s' \}
\end{aligned}
$$

and...

3.2. State with multiple components

One can generalise to several components; say, 'left' and 'right'...

The corresponding equational theory has four state laws on left:

$$
\begin{aligned}
&\mathbf{do}\ \{\, s \leftarrow get_L; s' \leftarrow get_L; return\ (s, s')\,\} &&= \mathbf{do}\ \{\, s \leftarrow get_L; return\ (s, s)\,\} \\
&\mathbf{do}\ \{\, set_L\ s; get_L\,\} &&= \mathbf{do}\ \{\, set_L\ s; return\ s\,\} \\
&\mathbf{do}\ \{\, s \leftarrow get_L; set_L\ s\,\} &&= \mathbf{do}\ \{\, return\ ()\,\} \\
&\mathbf{do}\ \{\, set_L\ s; set_L\ s'\,\} &&= \mathbf{do}\ \{\, set_L\ s'\,\}
\end{aligned}
$$

another four on right, and four stating that left and right are independent:

$$
\begin{aligned}
&\mathbf{do}\ \{\, a \leftarrow get_L; b \leftarrow get_R; return\ (a, b)\,\} \\
&\qquad\qquad = \mathbf{do}\ \{\, b \leftarrow get_R; a \leftarrow get_L; return\ (a, b)\,\} \\
&\mathbf{do}\ \{\, set_L\ a; b \leftarrow get_R; return\ b\,\} = \mathbf{do}\ \{\, b \leftarrow get_R; set_L\ a; return\ b\,\} \\
&\mathbf{do}\ \{\, set_R\ b; a \leftarrow get_L; return\ a\,\} = \mathbf{do}\ \{\, a \leftarrow get_L; set_R\ b; return\ a\,\} \\
&\mathbf{do}\ \{\, set_L\ a; set_R\ b\,\} = \mathbf{do}\ \{\, set_R\ b; set_L\ a\,\}
\end{aligned}
$$

3.3. Equational theory of entangled state

Those pair-state laws are too strong for interesting BX:

- *set–set* laws on either side imply very well-behavedness
- left–right independence precludes any interaction

We want a weaker theory. Say that BX is *well-behaved* if

$$
\begin{aligned}
\mathbf{do}\ \{\, a \leftarrow get_L; a' \leftarrow get_L; return\ (a, a')\,\} &= \mathbf{do}\ \{\, a \leftarrow get_L; return\ (a, a)\,\} \\
\mathbf{do}\ \{\, set_L\ a; a' \leftarrow get_L; return\ a'\,\} &= \mathbf{do}\ \{\, set_L\ a; return\ a\,\} \\
\mathbf{do}\ \{\, a \leftarrow get_L; set_L\ a\,\} &= \mathbf{do}\ \{\, return\ ()\,\} \\
\mathbf{do}\ \{\, b \leftarrow get_R; b' \leftarrow get_R; return\ (b, b')\,\} &= \mathbf{do}\ \{\, b \leftarrow get_R; return\ (b, b)\,\} \\
\mathbf{do}\ \{\, set_R\ b; b' \leftarrow get_R; return\ b'\,\} &= \mathbf{do}\ \{\, set_R\ b; return\ b\,\} \\
\mathbf{do}\ \{\, b \leftarrow get_R; set_R\ b\,\} &= \mathbf{do}\ \{\, return\ ()\,\} \\
\mathbf{do}\ \{\, a \leftarrow get_L; b \leftarrow get_R; return\ (a, b)\,\} &= \mathbf{do}\ \{\, b \leftarrow get_R; a \leftarrow get_L; return\ (a, b)\,\}
\end{aligned}
$$

(and *very well-behaved* if in addition set–set holds on each side).

3.4. Entanglement

Having introduced the state effect, it is natural to generalise, to allow other effects too.

We define a BX $A \rightleftharpoons_T B$ in monad T between A and B by

$$\begin{aligned} \textbf{data}\ BX\ \tau\ \alpha\ \beta = BX\ \{ & get_L :: \tau\ \alpha, \\ & get_R :: \tau\ \beta, \\ & set_L\ :: \alpha \to \tau\ (), \\ & set_R\ :: \beta \to \tau\ ()\ \} \end{aligned}$$

Say that BX is *well-behaved* if it satisfies the seven laws above.

Our earlier definitions were a special case, with $T = State\ (\alpha, \beta)$.

3.5. Really a generalization

Asymmetric lenses as entangled state:

$$
\begin{aligned}
&\mathit{lens2bx} :: \mathit{Lens}\ \alpha\ \beta \to \mathit{BX}\ (\mathit{State}\ \alpha)\ \alpha\ \beta \\
&\mathit{lens2bx}\ l = \mathit{BX}\ \mathit{get}\ \mathit{get}_V\ \mathit{set}\ \mathit{set}_V\ \mathbf{where} \\
&\quad \mathit{get}_V \quad\ = \mathbf{do}\ \{ s \leftarrow \mathit{get}; \mathit{return}\ (l.\mathit{view}\ s) \} \\
&\quad \mathit{set}_V\ v' = \mathbf{do}\ \{ s \leftarrow \mathit{get}; \mathit{set}\ (l.\mathit{update}\ s\ v') \}
\end{aligned}
$$

Symmetric lenses as entangled state:

$$
\begin{aligned}
&\mathit{slens2bx} :: \mathit{SLens}\ \alpha\ \beta\ \gamma \to \mathit{BX}\ (\mathit{State}\ (\alpha, \beta, \gamma))\ \alpha\ \beta \\
&\mathit{slens2bx}\ l = \mathit{BX}\ \mathit{get}_L\ \mathit{get}_R\ \mathit{set}_L\ \mathit{set}_R\ \mathbf{where} \\
&\quad \mathit{get}_L \quad = \mathbf{do}\ \{ (a, b, c) \leftarrow \mathit{get}; \mathit{return}\ a \} \\
&\quad \mathit{get}_R \quad = \mathbf{do}\ \{ (a, b, c) \leftarrow \mathit{get}; \mathit{return}\ b \} \\
&\quad \mathit{set}_L\ a' = \mathbf{do}\ \{ (a, b, c) \leftarrow \mathit{get}; \mathbf{let}\ (b', c') = l.\mathit{putlr}\ (a', c); \mathit{set}\ (a', b', c') \} \\
&\quad \mathit{set}_R\ b' = \mathbf{do}\ \{ (a, b, c) \leftarrow \mathit{get}; \mathbf{let}\ (a', c') = l.\mathit{putrl}\ (b', c); \mathit{set}\ (a', b', c') \}
\end{aligned}
$$

4. Combining effects

Now BX can use *other effects* in addition to state:

$$\mathbf{newtype}\ \mathit{StateT}\ \sigma\ \tau\ \alpha = \mathit{StateT}\ \{\mathit{runStateT} :: \sigma \rightarrow \tau\ (\alpha, \sigma)\}$$

$$\begin{aligned}
&\mathbf{instance}\ \mathit{Monad}\ \tau \Rightarrow \mathit{Monad}\ (\mathit{StateT}\ \sigma\ \tau)\ \mathbf{where} \\
&\quad \mathit{return}\ a = \mathit{StateT}\ (\lambda s \rightarrow \mathit{return}\ (a, s)) \\
&\quad m \gg= k\ \ = \mathit{StateT}\ (\lambda s \rightarrow \mathbf{do}\ \{(a, s') \leftarrow \mathit{runStateT}\ m\ s; \mathit{runStateT}\ (k\ a)\ s'\})
\end{aligned}$$

This too provides get and set operations (satisfying the same four laws):

$$\begin{aligned}
&\mathit{get} :: \mathit{Monad}\ \tau \Rightarrow \mathit{StateT}\ \sigma\ \tau\ \sigma \\
&\mathit{get} = \mathit{StateT}\ (\lambda s \rightarrow \mathit{return}\ (s, s))
\end{aligned}$$

$$\begin{aligned}
&\mathit{set} :: \mathit{Monad}\ \tau \Rightarrow \sigma \rightarrow \mathit{StateT}\ \sigma\ \tau\ () \\
&\mathit{set}\ s' = \mathit{StateT}\ (\lambda s \rightarrow \mathit{return}\ ((), s'))
\end{aligned}$$

but also supports *lifting* computations from the underlying monad:

$$\begin{aligned}
&\mathit{lift} :: \mathit{Monad}\ \tau \Rightarrow \tau\ \alpha \rightarrow \mathit{StateT}\ \sigma\ \tau\ \alpha \\
&\mathit{lift}\ m = \mathit{StateT}\ (\lambda s \rightarrow \mathbf{do}\ \{a \leftarrow m; \mathit{return}\ (a, s)\})
\end{aligned}$$

4.1. Example: environment

BX may be parametrised by some configuration data (eg Voigtländer's *bias*).

$$
\begin{aligned}
&\mathit{switch} :: (\gamma \to \mathit{BX}\ (\mathit{State}\ \sigma)\ \alpha\ \beta) \to \mathit{BX}\ (\mathit{StateT}\ \sigma\ (\mathit{Reader}\ \gamma))\ \alpha\ \beta \\
&\mathit{switch}\ \mathit{bx} = \mathit{BX}\ \mathit{gl}\ \mathit{gr}\ \mathit{sl}\ \mathit{sr}\ \textbf{where} \\
&\quad \mathit{gl} \quad\ = \textbf{do}\ \{ c \leftarrow \mathit{lift}\ \mathit{ask}; \mathit{inject}\ ((\mathit{bx}\ c).\mathit{get}_L) \} \\
&\quad \mathit{gr} \quad\ = \textbf{do}\ \{ c \leftarrow \mathit{lift}\ \mathit{ask}; \mathit{inject}\ ((\mathit{bx}\ c).\mathit{get}_R) \} \\
&\quad \mathit{sl}\ a\ = \textbf{do}\ \{ c \leftarrow \mathit{lift}\ \mathit{ask}; \mathit{inject}\ ((\mathit{bx}\ c).\mathit{set}_L\ a) \} \\
&\quad \mathit{sr}\ b = \textbf{do}\ \{ c \leftarrow \mathit{lift}\ \mathit{ask}; \mathit{inject}\ ((\mathit{bx}\ c).\mathit{set}_R\ b) \}
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathit{inject} :: \mathit{Monad}\ \tau \Rightarrow \mathit{State}\ \sigma\ \alpha \to \mathit{StateT}\ \sigma\ \tau\ \alpha \\
&\mathit{inject}\ m = \mathit{StateT}\ (\lambda s \to \mathit{return}\ (\mathit{runState}\ m\ s))
\end{aligned}
$$

4.2. Example: nondeterminism

When setting a new a', if it's not already consistent with existing b then nondeterministically select a new b' amongst those consistent with a'.

$$
\begin{array}{l}
\mathit{nondetBX} :: (\alpha \to \beta \to \mathit{Bool}) \to (\alpha \to [\,\beta\,]) \to (\beta \to [\,\alpha\,]) \to \\
\qquad\qquad \mathit{BX}\ (\mathit{StateT}\ (\alpha,\beta)\ [\,])\ \alpha\ \beta \\
\mathit{nondetBX}\ \mathit{ok}\ \mathit{bs}\ \mathit{as} = \mathit{BX}\ (\mathit{gets}\ \mathit{fst})\ (\mathit{gets}\ \mathit{snd})\ \mathit{set}_L\ \mathit{set}_R\ \mathbf{where} \\
\quad \mathit{set}_L\ a' = \mathbf{do}\ \{(a,b) \leftarrow \mathit{get}; \\
\qquad\qquad\qquad \mathbf{if}\ \mathit{ok}\ a'\ b\ \mathbf{then}\ \mathit{set}\ (a',b)\ \mathbf{else} \\
\qquad\qquad\qquad\quad \mathbf{do}\ \{b' \leftarrow \mathit{lift}\ (\mathit{bs}\ a'); \mathit{set}\ (a',b')\}\} \\
\quad \mathit{set}_R\ b' = \mathbf{do}\ \{(a,b) \leftarrow \mathit{get}; \\
\qquad\qquad\qquad \mathbf{if}\ \mathit{ok}\ a\ b'\ \mathbf{then}\ \mathit{set}\ (a,b')\ \mathbf{else} \\
\qquad\qquad\qquad\quad \mathbf{do}\ \{a' \leftarrow \mathit{lift}\ (\mathit{as}\ b'); \mathit{set}\ (a',b')\}\}
\end{array}
$$

where

$$
\begin{array}{l}
\mathit{gets} :: \mathit{Monad}\ \tau \Rightarrow (\sigma \to \alpha) \to \mathit{StateT}\ \sigma\ \tau\ \alpha \\
\mathit{gets}\ f = \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathit{return}\ (f\ s)\}
\end{array}
$$

4.3. Example: interaction—"transformation by example"

Maintain a collection of known ways to restore consistency. Use these when you can; when you can't, ask, and remember the answer.

$$
\begin{aligned}
&\mathit{dynamicBX} :: (\mathit{Eq}\ \alpha, \mathit{Eq}\ \beta, \mathit{Monad}\ \tau) \Rightarrow \\
&\qquad (\alpha \to \alpha \to \beta \to \tau\ \beta) \to (\alpha \to \beta \to \beta \to \tau\ \alpha) \to \\
&\qquad \mathit{BX}\ (\mathit{StateT}\ ((\alpha, \beta), [\,((\alpha, \alpha, \beta), \beta)\,], [\,((\alpha, \beta, \beta), \alpha)\,])\ \tau)\ \alpha\ \beta \\
&\mathit{dynamicBX}\ f\ g = \mathit{BX}\ (\mathit{gets}\ (\mathit{fst} \circ \mathit{fst3}))\ (\mathit{gets}\ (\mathit{snd} \circ \mathit{fst3}))\ \mathit{set}_L\ \mathit{set}_R\ \mathbf{where} \\
&\quad \mathit{set}_L\ a' = \mathbf{do}\ \{((a, b), \mathit{fs}, \mathit{bs}) \leftarrow \mathit{get}; \\
&\qquad\qquad \mathbf{if}\ a \equiv a'\ \mathbf{then}\ \mathit{return}\ ()\ \mathbf{else\ case}\ \mathit{lookup}\ (a, a', b)\ \mathit{fs}\ \mathbf{of} \\
&\qquad\qquad\quad \mathit{Just}\ b' \quad \to \mathit{set}\ ((a', b'), \mathit{fs}, \mathit{bs}) \\
&\qquad\qquad\quad \mathit{Nothing} \to \mathbf{do}\ \{b' \leftarrow \mathit{lift}\ (f\ a\ a'\ b); \\
&\qquad\qquad\qquad\qquad \mathit{set}\ ((a', b'), ((a, a', b), b') : \mathit{fs}, \mathit{bs})\}\} \\
&\quad \mathit{set}_R\ b' = \ldots \quad \text{-- dual}
\end{aligned}
$$

Eg ask the user ($\tau = \mathit{IO}$), or search exhaustively ($\tau = [\,]$).

5. Necessarily *StateT*?

All those examples instantiate the monad τ to *StateT S T* for some *S*, *T*.

It's *no (great) loss of generality* to stick to *StateT S T* rather than some more general *T*.

Here's why — and why 'great'.

5.1. Consistency and stability

Evidently a $bx :: BX\ T\ A\ B$ stores an (A, B) pair.

But not just any such pair: a *consistent* pair, ie one returnable via

$$\mathbf{do}\ \{ a \leftarrow bx.get_L; b \leftarrow bx.get_R; return\ (a, b) \}$$

This set of pairs is the consistency relation $A \bowtie B$ maintained by bx.

Note that this is not the same as a *stable* pair, an (a, b) such that

$$\mathbf{do}\ \{ bx.set_L\ a; bx.set_R\ b; bx.get_L \} = \mathbf{do}\ \{ bx.set_L\ a; bx.set_R\ b; return\ a \}$$
$$\mathbf{do}\ \{ bx.set_R\ b; bx.set_L\ a; bx.get_R \} = \mathbf{do}\ \{ bx.set_R\ b; bx.set_L\ a; return\ b \}$$

Stable pairs are consistent (for a well-behaved BX),
but consistent pairs are not necessarily stable.

Call a BX *stable* if all its consistent pairs are stable.

5.2. Data refinement

For stable *bx*, we have get and set operations on $A \bowtie B$ pairs:

$$\begin{aligned} get_{LR} &= \textbf{do}\ \{ a \leftarrow bx.get_L; b \leftarrow bx.get_R; return\ (a, b) \} \\ set_{LR}\ (a', b') &= \textbf{do}\ \{ bx.set_L\ a'; bx.set_R\ b' \} \end{aligned}$$

(but this is only well-behaved on $A \bowtie B$!).

From these, we can construct a data refinement $T \sqsubseteq StateT\ (A \bowtie B)\ T$:

$$abs\ m = \textbf{do}\ \{ ab \leftarrow get_{LR}; (c, ab') \leftarrow runStateT\ m\ ab; set_{LR}\ ab'; return\ c \}$$

So let's abbreviate

$$\textbf{type}\ StateTBX\ \tau\ \sigma\ \alpha\ \beta = BX\ (StateT\ \sigma\ \tau)\ \alpha\ \beta$$

6. Composition

It's crucial that BX should compose.

They do; but it's more delicate than you might expect—in particular, the interaction between well-behavedness and other effects.

We can't expect to compose arbitrary BX, because we can't compose arbitrary monads. So we consider only *StateTBX T S*, for different *S* but the same *T*.

6.1. Transparency

For many $StateTBX\ T\ S\ A\ B$, the get functions incur no additional effects: get_L is of the form $gets\ r$ for some $r :: S \to A$ (and similarly for get_R).

Call such a function *T-pure*.
(Not just 'pure': although it has no T-effects, it depends on the state.)

Call a BX *transparent* if its get_L and get_R are T-pure.

(Note that the *switch* example is not transparent, because the gets are not $(Reader\ \gamma)$-pure.)

6.2. Embeddings of stateful computations

A lens between state spaces induces a monad morphism:

$$\begin{aligned}
&\mathit{embed} :: \mathit{Monad}\ \tau \Rightarrow \mathit{Lens}\ \alpha\ \beta \to \mathit{StateT}\ \beta\ \tau\ \gamma \to \mathit{StateT}\ \alpha\ \tau\ \gamma \\
&\mathit{embed}\ l\ m = \mathbf{do}\ \{a \leftarrow \mathit{get}; \mathbf{let}\ b = l.\mathit{view}\ a; (c, b') \leftarrow \mathit{lift}\ (\mathit{runStateT}\ m\ b); \\
&\qquad \mathbf{let}\ a' = l.\mathit{update}\ a\ b'; \mathit{set}\ a'; \mathit{return}\ c\}
\end{aligned}$$

In particular, we can run stateful computations on compound states:

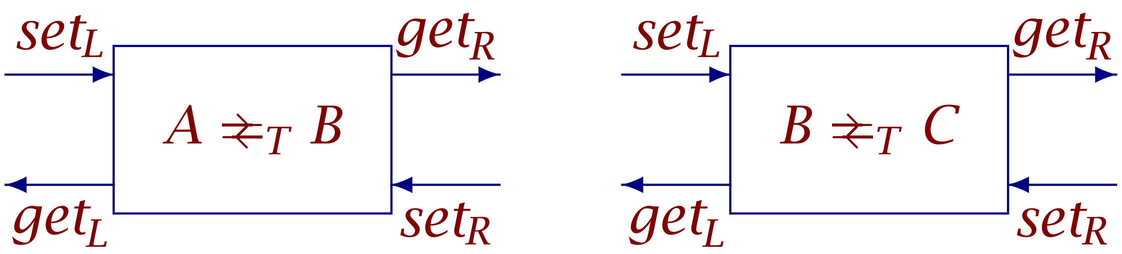
$$\begin{aligned}
&\mathit{left} :: \mathit{Monad}\ \tau \Rightarrow \mathit{StateT}\ \sigma_1\ \tau\ \alpha \to \mathit{StateT}\ (\sigma_1, \sigma_2)\ \tau\ \alpha \\
&\mathit{left} = \mathit{embed}\ (\mathit{Lens}\ \mathit{view}_L\ \mathit{update}_L)\ \mathbf{where} \\
&\quad \mathit{view}_L\ (s_1, s_2) = s_1 \\
&\quad \mathit{update}_L\ (s_1, s_2)\ s_1' = (s_1', s_2) \\
&\mathit{right} :: \mathit{Monad}\ \tau \Rightarrow \mathit{StateT}\ \sigma_2\ \tau\ \alpha \to \mathit{StateT}\ (\sigma_1, \sigma_2)\ \tau\ \alpha \\
&\mathit{right} = \mathit{embed}\ (\mathit{Lens}\ \mathit{view}_R\ \mathit{update}_R)\ \mathbf{where} \\
&\quad \mathit{view}_R\ (s_1, s_2) = s_2 \\
&\quad \mathit{update}_R\ (s_1, s_2)\ s_2' = (s_1, s_2')
\end{aligned}$$

6.3. Chaining together

Using *left* and *right*, we can define composition by:

$$
\begin{aligned}
&(\fatsemi) :: Monad\ \tau \Rightarrow \\
&\qquad StateTBX\ \tau\ \sigma_1\ \alpha\ \beta \rightarrow StateTBX\ \tau\ \sigma_2\ \beta\ \gamma \rightarrow StateTBX\ \tau\ (\sigma_1, \sigma_2)\ \alpha\ \gamma \\
&x \fatsemi y = BX\ gl\ gr\ sl\ sr\ \mathbf{where} \\
&\quad gl \quad = \mathbf{do}\ \{ left\ (get_L\ x) \} \\
&\quad gr \quad = \mathbf{do}\ \{ right\ (get_R\ y) \} \\
&\quad sl\ a = \mathbf{do}\ \{ left\ (set_L\ x\ a); b \leftarrow left\ (get_R\ x); right\ (set_L\ y\ b) \} \\
&\quad sr\ c = \mathbf{do}\ \{ right\ (set_R\ y\ c); b \leftarrow right\ (get_L\ y); left\ (set_R\ x\ b) \}
\end{aligned}
$$

The set operations carry the middle value across the gap:



The compound state consists only of the *consistent* pairs (s_1, s_2).

6.4. Equivalence

Here's an identity BX:

$$\mathit{identity} :: \mathit{Monad}\ \tau \Rightarrow \mathit{StateTBX}\ \tau\ \alpha\ \alpha\ \alpha$$
$$\mathit{identity} = \mathit{BX}\ \mathit{get}\ \mathit{get}\ \mathit{set}\ \mathit{set}$$

One might expect that $\mathit{identity} \fatsemi x = x = x \fatsemi \mathit{identity}$ for any x. But these don't even have the same types! We have to resort to equality 'up to'.

We say that $x :: \mathit{BX}\ T_1\ A\ B$ and $y :: \mathit{BX}\ T_2\ A\ B$ are *equivalent* (and write $x \equiv y$) if there exists an isomorphism $\varphi :: T_1\ \alpha \to T_2\ \alpha$ that preserves the operations (ie $\varphi\ (\mathit{get}_L\ x) = \mathit{get}_L\ y$ etc).

When $T_1 = \mathit{StateT}\ S_1\ T$ and $T_2 = \mathit{StateT}\ S_2\ T$, we can construct φ from an isomorphism between S_1 and S_2.

6.5. Composition is monoidal

Composition of transparent BX is associative, with *identity* as unit, modulo ≡.

$$\mathit{identity} \,{}_{9}^{\circ}\, x \equiv x \equiv x \,{}_{9}^{\circ}\, \mathit{identity}$$
$$x \,{}_{9}^{\circ}\, (y \,{}_{9}^{\circ}\, z) \equiv (x \,{}_{9}^{\circ}\, y) \,{}_{9}^{\circ}\, z$$

But note that transparency is important (or the underlying monad has to be commutative).

Note also that equivalence of state spaces is rather strong; bisimulation-based equivalences may be more appropriate.

7. Conclusions

- BX is inherently stateful
- in fact, that state is *entangled*
- having introduced state, we might as well introduce other effects too
- cleanly incorporates partiality, nondeterminism, I/O, ...
- but the conditions for preserving well-behavedness are subtle

- supported by EPSRC grant *A Theory of Least Change for BX*

- scaffolding for a unified study
- joint work with Faris Abou-Saleh, James Cheney, James McKinna, Perdita Stevens