Notions of Bidirectional Computation and Entangled State Monads

Faris Abou-Saleh[1], James Cheney[2], Jeremy Gibbons[1], James McKinna[2], and Perdita Stevens[2]

[1] Department of Computer Science, University of Oxford
firstname.lastname@cs.ox.ac.uk
[2] School of Informatics, University of Edinburgh
firstname.lastname@ed.ac.uk

Abstract. Bidirectional transformations (bx) support principled consistency maintenance among data sources. Each data source corresponds to one perspective on a composite system, manifested by operations to 'get' and 'set' a view of the whole from that particular perspective. Bx are important in a wide range of settings, including databases, interactive applications, and model-driven development. We show that bx are naturally modelled in terms of mutable state; in particular, the 'set' operations are stateful functions. This leads naturally to considering bx that exploit other computational effects too, such as I/O, nondeterminism, and failure, all largely ignored in the bx literature to date. We present a semantic foundation for symmetric bidirectional transformations with effects. We build on the mature theory of monadic encapsulation of effects in functional programming, develop the equational theory and important combinators for effectful bx, and provide a prototype implementation in Haskell along with several illustrative examples.

1 Introduction

Bidirectional transformations (bx) arise when synchronising data in different data sources: updates to one source entail corresponding updates to the others, in order to maintain consistency. When a data source represents the complete information, this is a straightforward task; an update can be matched by discarding and regenerating the other sources. It becomes more interesting when one data representation lacks some information that is recorded by another; then the corresponding update has to merge new information on one side with old information on the other side. Such bidirectional transformations have been the focus of a flurry of recent activity—in databases, in programming languages, and in software engineering, among other fields—giving rise to a flourishing series of BX Workshops (see http://bx-community.wikidot.com/) and BX Seminars (in Japan, Germany, and Canada so far: see [6] for an early report on the state of the art).

The different branches of the bx community have come up with a variety of different formalisations of bx with conflicting definitions and incompatible

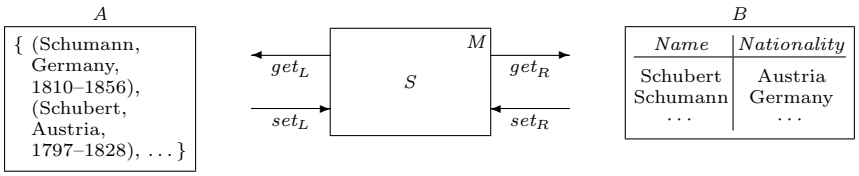

Fig. 1. Effectful bx between sources A and B and with effects in monad M, with hidden state S, illustrated using the Composers example

extensions, such as *lenses* [10], *relational bx* [32], *symmetric lenses* [13], *putback-based lenses* [28], and *profunctors* [21]. We have been seeking a unification of the varying approaches. It turns out that quite a satisfying unifying formalism can be obtained from the perspective of the state monad. More specifically, we are thinking about two data sources, and stateful computations on acting on pairs representing these two sources. However, the two components of the pair are not independent, as two distinct memory cells would be, but are *entangled*—a change to one component generally entails a consequent change to the other.

This stateful perspective suggests using monads for bx, much as Moggi showed that monads unify many computational effects [25]. But not only that; it suggests a way to *generalise* bx to encompass other features that monads can handle. In fact, several approaches to lenses do in practice allow for monadic operations [21, 28]. But there are natural concerns about such an extension: Are "monadic lenses" legitimate bidirectional transformations? Do they satisfy laws analogous to the roundtripping ('GetPut' and 'PutGet') laws of traditional lenses? Can we compose such transformations? We show that bidirectional computations can be encapsulated in monads, and be combined with other standard monads to accommodate effects, while still satisfying appropriate equational laws and supporting composition.

To illustrate our approach informally, consider Figure 1, which is based on the *Composers* example [33]. This example relates two data sources A and B: on the left, an $a :: A$ consists of a set of triples $(name, nationality, dates)$, and on the right, a $b :: B$ consists of an ordered list of pairs $(name, nationality)$. The two sources a and b are consistent when they contain the name–nationality pairs, ignoring dates and ordering. The centre of the figure illustrates the interface and typical operations of our (effectful) bx, including a monad M, operations $get_L :: M\ A$ and $get_R :: M\ B$ that return the current values of the left and right sides, and operations $set_L :: A \to M\ ()$ and $set_R :: B \to M\ ()$ that accept a new value for the left-hand or right-hand side, possibly performing side-effects in M.

In this example, neither A nor B is obtainable from the other; A omits the ordering information from B, while B omits the date information from A. This means that there may be multiple ways to change one side to match a change to the other. For example, the monadic computation $\mathbf{do}\ \{b \leftarrow get_R; set_R\ (b +\!\!+ [(\texttt{"Bach"}, \texttt{"Germany"})])\}$ looks at the current value of the B-side, and modifies it to include a new pair at the end. Of course, this addition is ambiguous: do we mean J. S. Bach (1685–1750), J. C. Bach (1735–1782), or another Bach? There

is no way to give the dates through the B interface; typically, pure bx would initialise the unspecified part to some default value such as `"????-????"`. Conversely, had we inserted the triple (`"Bach"`, `"Germany"`, `"1685-1750"`) on the left, then there may be several consistent ways to change the right-hand side to match: for example, inserting at the beginning, the end, or somewhere in the middle of the list. Again, conventional pure bx must fix some strategy in advance.

A conventional (pure) bx corresponds (roughly) to taking $M = \mathit{State}\ S$, where S is some type of states from which we can obtain both A and B; for example, S could consist of *lists* of triples $(\mathit{name}, \mathit{nationality}, \mathit{dates})$ from which we can easily extract both A (by forgetting the order) and B (by forgetting the dates). However, our approach to effectful bx allows many other choices for M that allow us to use side-effects when restoring consistency. Consider the following two scenarios, taken from the model-driven development domain, where the entities being synchronised are 'model states' a, b such as UML models or RDBMS schemas, drawn from suitable 'model spaces' A, B. We will revisit them among the concrete examples in Section 5.

Scenario 1.1 (nondeterminism). As mentioned above, most formal notions of bx require that the transformation (or programmer) decide on a consistency-restoration strategy in advance. By contrast, in the Janus Transformation Language (JTL) [5], programmers need only specify a consistency relation, allowing the bx engine to resolve the underspecification nondeterministically. Given a change to one source, JTL uses an external constraint solver to find a consistent choice for the other source; there might be multiple choices.

Our effectful bx can handle this by combining state with a nondeterministic choice monad, taking $M = \mathit{StateT}\ S\ [\,]$. For example, an attempt to add Bach to the right-hand side could result in several possibilities for the left-hand side (perhaps using an external source such as Wikipedia to find the dates for candidate matches). Conversely, adding Bach to the left-hand side could result in a nondeterministic choice of all possible positions to add the matching record in the right-hand list. No previous formalism permits such *nondeterministic bx* to be composed with conventional deterministic transformations, or characterises the laws that such transformations ought to satisfy. ♢

Scenario 1.2 (interaction). Alternatively, instead of automatically trying to find a single (or all possible) dates for Bach, why not ask the user for help? In unidirectional model transformation settings, Varró [36] proposed "model transformation by example", where the transformation system 'learns' gradually, by prompting its users over time, the desired way to restore consistency in various situations. One can see this as an (interactive) instance of *memoisation*.

Our effectful bx can also handle this, using the IO monad in concert with some additional state to remember past questions and their answers. For example, when Bach is added to the right-hand side, the bx can check to see whether it already knows how to restore consistency. If so, it does so without further ado. If not, it queries the user to determine how to fill in the missing values needed

for the right-hand side, perhaps having first appealed to some external knowledge base to generate helpful suggestions. It then records the new version of the right-hand side and updates its state so that the same question is not asked again. No previous formalism allows for I/O during consistency restoration. ♢

The paper is structured as follows. Section 2 reviews monads as a foundation for effectful programming, fixing (idealised) Haskell notation used throughout the paper, and recaps definitions of lenses. Our contributions start in Section 3, with a presentation of our monadic approach to bx. Section 4 considers a definition of composition for effectful bx. In Section 5 we discuss initialisation, and formalise the motivating examples above, along with other combinators and examples of effectful bx. These examples are the first formal treatments of effects such as nondeterminism or interaction for symmetric bidirectional transformations, and they illustrate the generality of our approach. Finally we discuss related work and conclude. All proofs and code for examples can be found in an extended version of this paper [1].

2 Background

Our approach to bx is semantics-driven, so we here provide some preliminaries on semantics of effectful computation – focusing on monads, Haskell's use of type classes for them, and some key instances that we exploit heavily in what follows. We also briefly recap the definitions of asymmetric and symmetric lenses.

2.1 Effectful computation

Moggi's seminal work on the semantics of effectful computation [25], and much continued investigation, shows how computational effects can be described using monads. Building on this, we assume that computations are represented as Kleisli arrows for a strong monad T defined on a cartesian closed category $\mathbb{C}$ of 'value' types and 'pure' functions. The reader uncomfortable with such generality can safely consider our definitions in terms of the category of sets and total functions, with T encapsulating the 'ambient' programming language effects: none in a total functional programming language like Agda, partiality in Haskell, global state in Pascal, network access in Java, etc.

2.2 Notational conventions

We write in Haskell notation, except for the following few idealisations. We assume a cartesian closed category $\mathbb{C}$, avoiding niceties about lifted types and undefined values in Haskell; we further restrict attention to terminating programs. We use lowercase (Greek) letters for polymorphic type variables in code, and uppercase (Roman) letters for monomorphic instantiations of those variables in accompanying prose. We elide constructors and destructors for a **newtype**, the explicit witnesses to the isomorphism between the defined type and its structure,

and use instead a **type** synonym that equates the defined type and its structure; e.g., in Section 2.4 we omit the function *runStateT* from $StateT\ S\ T\ A$ to $S \to T\ (A, S)$. Except where expressly noted, we assume a kind of Barendregt convention, that bound variables are chosen not to clash with free variables; for example, in the definition below of a commutative monad, we elide the explicit proviso "for x, y distinct variables not free in m, n" (one might take the view that m, n are themselves variables, rather than possibly open terms that might capture x, y). We use a tightest-binding lowered dot for field access in records; e.g., in Definition 3.8 we write $bx.get_L$ rather than $get_L\ bx$; we therefore write function composition using a centred dot, $f \cdot g$. The code online expands these conventions into real Haskell. We also make extensive use of equational reasoning over monads in **do** notation [11]. Different branches of the bx community have conflicting naming conventions for various operations, so we have renamed some of them, favouring internal over external consistency.

2.3 Monads

Definition 2.1 (monad type class). Type constructors representing notions of effectful computation are represented as instances of the Haskell type class *Monad*:

```
class Monad τ where
  return :: α → τ α
  (>>=)  :: τ α → (α → τ β) → τ β   -- pronounced 'bind'
```

A monad instance should satisfy the following laws:

$$\begin{aligned} &return\ x \gg\!= f &&= f\ x \\ &m \gg\!= return &&= m \\ &(m \gg\!= f) \gg\!= g &&= m \gg\!= \lambda x.\, (f\ x \gg\!= g) \end{aligned}$$ ♢

Common examples in Haskell (with which we assume familiarity) include:

```
type Id α          = α                  -- no effects
data Maybe α       = Just α | Nothing   -- failure/exceptions
data [α]           = [] | α : [α]       -- choice
type State σ α     = σ → (α, σ)         -- state
type Reader σ α    = σ → α              -- environment
type Writer σ α    = (α, σ)             -- logging
```

as well as the (in)famous *IO* monad, which encapsulates interaction with the outside world. We need a *Monoid* σ instance for the *Writer* σ monad, in order to support empty and composite logs.

Definition 2.2. In Haskell, monadic expressions may be written using **do** notation, which is defined by translation into applications of bind:

$$\begin{aligned}
&\mathbf{do}\,\{\mathbf{let}\ decls; ms\} &&= \mathbf{let}\ decls\ \mathbf{in}\ \mathbf{do}\,\{ms\}\\
&\mathbf{do}\,\{a \leftarrow m; ms\} &&= m \gg\!= \lambda a.\,\mathbf{do}\,\{ms\}\\
&\mathbf{do}\,\{m\} &&= m
\end{aligned}$$

The *body* ms of a **do** expression consists of zero or more 'qualifiers', and a final expression m of monadic type; qualifiers are either 'declarations' **let** *decls* (with *decls* a collection of bindings $a = e$ of patterns a to expressions e) or 'generators' $a \leftarrow m$ (with pattern a and monadic expression m). Variables bound in pattern a may appear free in the subsequent body ms; in contrast to Haskell, we assume that the pattern cannot fail to match. When the return value of m is not used – e.g., when void – we write $\mathbf{do}\,\{m; ms\}$ as shorthand for $\mathbf{do}\,\{\_ \leftarrow m; ms\}$ with its wildcard pattern. ♢

Definition 2.3 (commutative monad). We say that $m :: T\ A$ *commutes in* T if the following holds for all $n :: T\ B$:

$$\mathbf{do}\,\{x \leftarrow m; y \leftarrow n; return\ (x, y)\} = \mathbf{do}\,\{y \leftarrow n; x \leftarrow m; return\ (x, y)\}$$

A monad T is *commutative* if all $m :: T\ A$ commute, for all A. ♢

Definition 2.4. An element z of a monad is called a *zero element* if it satisfies:

$$\mathbf{do}\,\{x \leftarrow z; f\ x\} = z = \mathbf{do}\,\{x \leftarrow m; z\}$$ ♢

Among monads discussed so far, *Id*, *Reader* and *Maybe* are commutative; if σ is a commutative monoid, *Writer* σ is commutative; but many interesting monads, such as *IO* and *State*, are not. The *Maybe* monad has zero element *Nothing*, and *List* has zero *Nil*; the zero element is unique if it exists.

Definition 2.5 (monad morphism). Given monads T and T', a *monad morphism* is a polymorphic function $\varphi :: \forall\alpha.\, T\ \alpha \to T'\ \alpha$ satisfying

$$\begin{aligned}
&\varphi\ (\mathbf{do}_T\,\{return\ a\}) &&= \mathbf{do}_{T'}\,\{return\ a\}\\
&\varphi\ (\mathbf{do}_T\,\{a \leftarrow m; k\ a\}) &&= \mathbf{do}_{T'}\,\{a \leftarrow \varphi\ m; \varphi\ (k\ a)\}
\end{aligned}$$

(subscripting to make clear which monad is used where). ♢

2.4 Combining state and other effects

We recall the *state monad transformer* (see e.g. Liang *et al.* [22]).

Definition 2.6 (state monad transformer). State can be combined with effects arising from an arbitrary monad T using the *StateT* monad transformer:

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

This provides *get* and *set* operations for the state type:

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

which satisfy the following four laws [29]:

$$
\begin{array}{lll}
\text{(GG)} & \mathbf{do}\ \{\, s \leftarrow \mathit{get}; s' \leftarrow \mathit{get}; \mathit{return}\ (s, s')\,\} & = \mathbf{do}\ \{\, s \leftarrow \mathit{get}; \mathit{return}\ (s, s)\,\} \\
\text{(SG)} & \mathbf{do}\ \{\, \mathit{set}\ s; \mathit{get}\,\} & = \mathbf{do}\ \{\, \mathit{set}\ s; \mathit{return}\ s\,\} \\
\text{(GS)} & \mathbf{do}\ \{\, s \leftarrow \mathit{get}; \mathit{set}\ s\,\} & = \mathbf{do}\ \{\, \mathit{return}\ ()\,\} \\
\text{(SS)} & \mathbf{do}\ \{\, \mathit{set}\ s; \mathit{set}\ s'\,\} & = \mathbf{do}\ \{\, \mathit{set}\ s'\,\}
\end{array}
$$

Computations in T embed into $\mathit{StateT}\ S\ T$ via the monad morphism *lift*:

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

Lemma 2.7. Unused *get*s are discardable:

$$\mathbf{do}\ \{\, \_ \leftarrow \mathit{get}; m\,\} = \mathbf{do}\ \{\, m\,\} \qquad \diamondsuit$$

Lemma 2.8 (liftings commute with *get* and *set*). We have:

$$
\begin{aligned}
&\mathbf{do}\ \{\, a \leftarrow \mathit{get}; b \leftarrow \mathit{lift}\ m; \mathit{return}\ (a, b)\,\} \\
&\qquad\qquad = \mathbf{do}\ \{\, b \leftarrow \mathit{lift}\ m; a \leftarrow \mathit{get}; \mathit{return}\ (a, b)\,\} \\
&\mathbf{do}\ \{\, \mathit{set}\ a; b \leftarrow \mathit{lift}\ m; \mathit{return}\ b\,\} = \mathbf{do}\ \{\, b \leftarrow \mathit{lift}\ m; \mathit{set}\ a; \mathit{return}\ b\,\}
\end{aligned}
\qquad \diamondsuit
$$

Definition 2.9. Some convenient shorthands:

$$
\begin{aligned}
&\mathit{gets} :: \mathit{Monad}\ \tau \Rightarrow (\sigma \to \alpha) \to \mathit{StateT}\ \sigma\ \tau\ \alpha \\
&\mathit{gets}\ f = \mathbf{do}\ \{\, s \leftarrow \mathit{get}; \mathit{return}\ (f\ s)\,\} \\
&\mathit{eval} :: \mathit{Monad}\ \tau \Rightarrow \mathit{StateT}\ \sigma\ \tau\ \alpha \to \sigma \to \tau\ \alpha \\
&\mathit{eval}\ m\ s = \mathbf{do}\ \{\, (a, s') \leftarrow m\ s; \mathit{return}\ a\,\} \\
&\mathit{exec} :: \mathit{Monad}\ \tau \Rightarrow \mathit{StateT}\ \sigma\ \tau\ \alpha \to \sigma \to \tau\ \sigma \\
&\mathit{exec}\ m\ s = \mathbf{do}\ \{\, (a, s') \leftarrow m\ s; \mathit{return}\ s'\,\}
\end{aligned}
\qquad \diamondsuit
$$

Definition 2.10. We say that a computation $m :: \mathit{StateT}\ S\ T\ A$ is a *T-pure query* if it cannot change the state, and is pure with respect to the base monad T; that is, $m = \mathit{gets}\ h$ for some $h :: S \to A$. Note that a T-pure query need not be pure with respect to $\mathit{StateT}\ S\ T$; in particular, it will typically read the state. $\diamondsuit$

Definition 2.11 (data refinement). Given monads M of 'abstract computations' and M' of 'concrete computations', various 'abstract operations' $\mathit{op} :: A \to M\ B$ with corresponding 'concrete operations' $\mathit{op}' :: A \to M'\ B$, an 'abstraction function' $\mathit{abs} :: M'\ \alpha \to M\ \alpha$ and a 'reification function' $\mathit{conc} :: M\ \alpha \to M'\ \alpha$, we say that *conc* is a *data refinement* from (M, op) to (M', op') if:

- $conc$ distributes over $(\gg\!=)$
- $abs \cdot conc = id$, and
- $op' = conc \cdot op$ for each of the operations. ♢

Remark 2.12. Given such a data refinement, a composite abstract computation can be faithfully simulated by a concrete one:

$$\begin{aligned}
& \mathbf{do}\,\{\, a \leftarrow op_1\ (); b \leftarrow op_2\ (a); op_3\ (a, b)\,\} \\
= \quad & [\![\ abs \cdot conc = id\ ]\!] \\
& abs\ (conc\ (\mathbf{do}\,\{\, a \leftarrow op_1\ (); b \leftarrow op_2\ (a); op_3\ (a, b)\,\})) \\
= \quad & [\![\ conc \text{ distributes over } (\gg\!=)\ ]\!] \\
& abs\ (\mathbf{do}\,\{\, a \leftarrow conc\ (op_1\ ()); b \leftarrow conc\ (op_2\ (a)); conc\ (op_3\ (a, b))\,\}) \\
= \quad & [\![\ \text{concrete operations}\ ]\!] \\
& abs\ (\mathbf{do}\,\{\, a \leftarrow op'_1\ (); b \leftarrow op'_2\ (a); op'_3\ (a, b)\,\})
\end{aligned}$$

If $conc$ also preserves $return$ (so $conc$ is a monad morphism), then we would have a similar result for 'empty' abstract computations too; but we don't need that stronger property in this paper, and it does not hold for our main example (Remark 3.7). ♢

Lemma 2.13. Given an arbitrary monad T, not assumed to be an instance of $StateT$, with operations $get_T :: T\ S$ and $set_T :: S \to T\ ()$ for a type S, such that get_T and set_T satisfy the laws (GG), (GS), and (SG) of Definition 2.6, then there is a data refinement from T to $StateT\ S\ T$. ♢

Proof (sketch). The abstraction function abs from $StateT\ S\ T$ to T and the reification function $conc$ in the opposite direction are given by

$$\begin{aligned}
abs\ m &= \mathbf{do}\,\{\, s \leftarrow get_T; (a, s') \leftarrow m\ s; set_T\ s'; return\ a\,\} \\
conc\ m &= \lambda s.\, \mathbf{do}\,\{\, a \leftarrow m; s' \leftarrow get_T; return\ (a, s')\,\} \\
&= \mathbf{do}\,\{\, a \leftarrow lift\ m; s' \leftarrow lift\ get_T; set\ s'; return\ a\,\}
\end{aligned}$$

□

Remark 2.14. Informally, if T provides suitable get and set operations, we can without loss of generality assume it to be an instance of $StateT$. The essence of the data refinement is for concrete computations to maintain a shadow copy of the implicit state; $conc\ m$ synchronises the outer copy of the state with the inner copy after executing m, and $abs\ m$ runs the $StateT$ computation m on an initial state extracted from T, and stores the final state back there. ♢

2.5 Lenses

The notion of an (asymmetric) 'lens' between a source and a view was introduced by Foster *et al.* [10]. We adapt their notation, as follows.

Definition 2.15. A lens $l :: Lens\ S\ V$ from source type S to view type V consists of a pair of functions which get a *view* of the source, and *update* an old source with a modified view:

$$\textbf{data}\ \mathit{Lens}\ \alpha\ \beta = \mathit{Lens}\ \{\mathit{view} :: \alpha \to \beta, \mathit{update} :: \alpha \to \beta \to \alpha\}$$

We say that a lens $l :: \mathit{Lens}\ S\ V$ is *well behaved* if it satisfies the two round-tripping laws

$$\begin{array}{ll} \text{(UV)} & l.\mathit{view}\ (l.\mathit{update}\ s\ v) = v \\ \text{(VU)} & l.\mathit{update}\ s\ (l.\mathit{view}\ s) = s \end{array}$$

and *very well-behaved* or *overwritable* if in addition

$$\text{(UU)} \quad l.\mathit{update}\ (l.\mathit{update}\ s\ v)\ v' = l.\mathit{update}\ s\ v' \qquad \diamondsuit$$

Remark 2.16. Very well-behavedness captures the idea that, after two successive updates, the second update completely *overwrites* the first. It turns out to be a rather strong condition, and many natural lenses do not satisfy it. Those that do generally have the special property that source S factorises cleanly into $V \times C$ for some type C of 'complements' independent of V, and so the view is a projection. For example:

$$\begin{array}{l} \mathit{fstLens} :: \mathit{Lens}\ (a, b)\ a \\ \mathit{fstLens} = \mathit{Lens}\ \mathit{fst}\ u\ \textbf{where}\ u\ (a, b)\ a' = (a', b) \\ \mathit{sndLens} :: \mathit{Lens}\ (a, b)\ b \\ \mathit{sndLens} = \mathit{Lens}\ \mathit{snd}\ u\ \textbf{where}\ u\ (a, b)\ b' = (a, b') \end{array}$$

But in general, the V may be computed from and therefore depend on all of the S value, and there is no clean factorisation of S into $V \times C$. $\diamondsuit$

Asymmetric lenses are constrained, in the sense that they relate two types S and V in which the view V is completely determined by the source S. Hofmann *et al.* [13] relaxed this constraint, introducing *symmetric lenses* between two types A and B, neither of which need determine the other:

Definition 2.17. A *symmetric lens* from A to B with complement type C consists of two functions converting to and from A and B, each also operating on C.

$$\textbf{data}\ \mathit{SLens}\ \gamma\ \alpha\ \beta = \mathit{SLens}\ \{\mathit{put}_R :: (\alpha, \gamma) \to (\beta, \gamma), \mathit{put}_L :: (\beta, \gamma) \to (\alpha, \gamma)\}$$

We say that symmetric lens l is *well-behaved* if it satisfies the following two laws:

$$\begin{array}{lllll} \text{(PutRL)} & l.\mathit{put}_R\ (a, c) = (b, c') & \Rightarrow & l.\mathit{put}_L\ (b, c') = (a, c') \\ \text{(PutLR)} & l.\mathit{put}_L\ (b, c) = (a, c') & \Rightarrow & l.\mathit{put}_R\ (a, c') = (b, c') \end{array}$$

(There is also a stronger notion of very well-behavedness, but we do not need it for this paper.) $\diamondsuit$

Remark 2.18. The idea is that A and B represent two overlapping but distinct views of some common underlying data, and the so-called complement C represents their amalgamation (*not* necessarily containing all the information

from both: rather, one view plus the complement together contain enough information to reconstruct the other view). Each function takes a new view and the old complement, and returns a new opposite view and a new complement. The two well-behavedness properties each say that after one update operation, the complement c' is fully consistent with the current views, and so a subsequent opposite update with the same view has no further effect on the complement. ♢

3 Monadic bidirectional transformations

We have seen that the state monad provides a pair *get*, *set* of operations on the state. A symmetric bx should provide two such pairs, one for each data source; these four operations should be effectful, not least because they should operate on some shared consistent state. We therefore introduce the following general notion of *monadic bx* (which we sometimes call 'mbx', for short).

Definition 3.1. We say that a data structure $t :: BX\ T\ A\ B$ is a *bx between A and B in monad T* when it provides appropriately typed functions:

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

3.1 Entangled state

The *get* and *set* operations of the state monad satisfy the four laws (GG), (SG), (GS), (SS) of Definition 2.6. More generally, one can give an equational theory of state with multiple memory locations; in particular, with just two locations 'left' (L) and 'right' (R), the equational theory has four operations get_L, set_L, get_R, set_R that match the BX interface. This theory has four laws for L analogous to those of Definition 2.6, another four such laws for R, and a final four laws stating that the L-operations commute with the R-operations. But this equational theory of two memory locations is too strong for interesting bx, because of the commutativity requirement: the whole point of the exercise is that invoking set_L should indeed affect the behaviour of a subsequent get_R, and symmetrically. We therefore impose only a subset of those twelve laws on the BX interface.

Definition 3.2. A *well-behaved BX* is one satisfying the following seven laws:

$$\begin{aligned}
&(\mathrm{G_L G_L}) && \textbf{do}\,\{\, a \leftarrow get_L; a' \leftarrow get_L; return\ (a, a')\,\} \\
&&& \quad = \textbf{do}\,\{\, a \leftarrow get_L; return\ (a, a)\,\} \\
&(\mathrm{S_L G_L}) && \textbf{do}\,\{\, set_L\ a; get_L\,\} \quad = \textbf{do}\,\{\, set_L\ a; return\ a\,\} \\
&(\mathrm{G_L S_L}) && \textbf{do}\,\{\, a \leftarrow get_L; set_L\ a\,\} = \textbf{do}\,\{\, return\ ()\,\} \\
&(\mathrm{G_R G_R}) && \textbf{do}\,\{\, a \leftarrow get_R; a' \leftarrow get_R; return\ (a, a')\,\} \\
&&& \quad = \textbf{do}\,\{\, a \leftarrow get_R; return\ (a, a)\,\} \\
&(\mathrm{S_R G_R}) && \textbf{do}\,\{\, set_R\ a; get_R\,\} \quad = \textbf{do}\,\{\, set_R\ a; return\ a\,\} \\
&(\mathrm{G_R S_R}) && \textbf{do}\,\{\, a \leftarrow get_R; set_R\ a\,\} = \textbf{do}\,\{\, return\ ()\,\}
\end{aligned}$$

$$(\mathrm{G_L G_R}) \quad \mathbf{do}\,\{\, a \leftarrow get_L; b \leftarrow get_R; return\ (a,b)\,\} = \mathbf{do}\,\{\, b \leftarrow get_R; a \leftarrow get_L; return\ (a,b)\,\}$$

We further say that a BX is *overwritable* if it satisfies

$$(\mathrm{S_L S_L}) \quad \mathbf{do}\,\{\, set_L\ a; set_L\ a'\,\} = \mathbf{do}\,\{\, set_L\ a'\,\}$$
$$(\mathrm{S_R S_R}) \quad \mathbf{do}\,\{\, set_R\ a; set_R\ a'\,\} = \mathbf{do}\,\{\, set_R\ a'\,\} \qquad \diamondsuit$$

We might think of the A and B views as being *entangled*; in particular, we call the monad arising as the initial model of the theory with the four operations $get_L, set_L, get_R, set_R$ and the seven laws $(\mathrm{G_L G_L}) \ldots (\mathrm{G_L G_R})$ the *entangled state monad*.

Remark 3.3. Overwritability is a strong condition, corresponding to very well-behavedness of lenses [10], history-ignorance of relational bx [32] etc.; many interesting bx fail to satisfy it. Indeed, in an effectful setting, a law such as $(\mathrm{S_L S_L})$ demands that $set_L\ a'$ be able to undo (or overwrite) any effects arising from $set_L\ a$; such behaviour is plausible in the pure state-based setting, but not in general. Consequently, we do not demand overwritability in what follows. $\diamondsuit$

Definition 3.4. A *bx morphism* from $bx_1 :: BX\ T_1\ A\ B$ to $bx_2 :: BX\ T_2\ A\ B$ is a monad morphism $\varphi : \forall \alpha.\, T_1\ \alpha \to T_2\ \alpha$ that preserves the bx operations, in the sense that $\varphi\ (bx_1.get_L) = bx_2.get_L$ and so on. A *bx isomorphism* is an invertible bx morphism, i.e. a pair of monad morphisms $\iota :: \forall \alpha.\, T_1\ \alpha \to T_2\ \alpha$ and $\iota^{-1} : \forall \alpha.\, T_2\ \alpha \to T_1\ \alpha$ which are mutually inverse, and which also preserve the operations. We say that bx_1 and bx_2 are *equivalent* (and write $bx_1 \equiv bx_2$) if there is a bx isomorphism between them. $\diamondsuit$

3.2 Stateful BX

The get and set operations of a BX, and the relationship via entanglement with the equational theory of the state monad, strongly suggest that there is something inherently stateful about bx; that will be a crucial observation in what follows. In particular, the get_L and get_R operations of a $BX\ T\ A\ B$ reveal that it is in some sense storing an $A \times B$ pair; conversely, the set_L and set_R operations allow that pair to be updated. We therefore focus on monads of the form $StateT\ S\ T$, where S is the 'state' of the bx itself, capable of recording an A and a B, and T is a monad encapsulating any other ambient effects that can be performed by the bx.

Definition 3.5. We introduce the following instance of the BX signature (note the inverted argument order):

$$\mathbf{type}\ StateTBX\ \tau\ \sigma\ \alpha\ \beta = BX\ (StateT\ \sigma\ \tau)\ \alpha\ \beta \qquad \diamondsuit$$

The intuition is that the state S of a $StateTBX\ T\ S\ A\ B$ includes (at least) the current A and B values, which we may 'get' and 'set' via the BX interface, possibly incurring effects in T in the process.

Remark 3.6. In fact, we can say more about the pair inside a $bx :: BX\ T\ A\ B$: it will generally be the case that only certain such pairs are observable. Specifically, we can define the subset $R \subseteq A \times B$ of *consistent pairs* according to bx, namely those pairs (a, b) that may be returned by

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

We can see this subset R as the *consistency relation* between A and B maintained by bx. We sometimes write $A \bowtie B$ for this relation, when the bx in question is clear from context. ♢

Remark 3.7. Note that restricting attention to instances of $StateT$ is not as great a loss of generality as might at first appear. Consider a well-behaved bx of type $BX\ T\ A\ B$, over some monad T not assumed to be an instance of $StateT$. We say that a consistent pair $(a, b) :: A \bowtie B$ is *stable* if, when setting the components in either order, the later one does not disturb the earlier:

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

We say that the bx itself is stable if all its consistent pairs are stable. Stability does not follow from the laws, but many bx do satisfy this stronger condition. And given a stable bx, we can construct get and set operations for $A \bowtie B$ pairs, satisfying the three laws (GG), (GS), (SG) of Definition 2.6. By Lemma 2.13, this gives a data refinement from T to $StateT\ S\ T$, and so we lose nothing by using $StateT\ S\ T$ instead of T. Despite this, we do not impose stability as a requirement in the following, because some interesting bx are not stable. ♢

We have not found convincing examples of $StateTBX$ in which the two get functions have effects from T, rather than being T-pure queries. When the get functions are T-pure queries, we obtain the get/get commutation laws ($\mathrm{G_L G_L}$), ($\mathrm{G_R G_R}$), ($\mathrm{G_L G_R}$) for free [11], motivating the following:

Definition 3.8. We say that a well-behaved $bx :: StateTBX\ T\ S\ A\ B$ in the monad $StateT\ S\ T$ is *transparent* if get_L, get_R are T-pure queries, *i.e.* there exist $read_L :: S \to A$ and $read_R :: S \to B$ such that $bx.get_L = gets\ read_L$ and $bx.get_R = gets\ read_R$. ♢

Remark 3.9. Under the mild condition (Moggi's *monomorphism condition* [25]) on T that *return* be injective, $read_L$ and $read_R$ are *uniquely* determined for a transparent bx; so informally, we refer to $bx.read_L$ and $bx.read_R$, regarding them as part of the signature of bx. The monomorphism condition holds for the various monads we consider here (provided we have non-empty types σ for *State*, *Reader*, *Writer*). ♢

Now, transparent $StateTBX$ compose (Section 4), while general bx with effectful gets do not. So, in what follows, we confine our attention to transparent bx.

3.3 Subsuming lenses

Asymmetric lenses, as in Definition 2.15, are subsumed by *StateTBX*. To simulate $l :: Lens\ A\ B$, one uses a *StateTBX* on state A and underlying monad Id:

$$\begin{aligned}&BX\ get\ set\ get_R\ set_R\ \mathbf{where}\\ &\quad get_R\ \ \ \ = \mathbf{do}\ \{\, a \leftarrow get; return\ (l.view\ a)\,\}\\ &\quad set_R\ b' = \mathbf{do}\ \{\, a \leftarrow get; set\ (l.update\ a\ b')\,\}\end{aligned}$$

Symmetric lenses, as in Definition 2.17, are subsumed by our effectful bx too. In a nutshell, to simulate $sl :: SLens\ C\ A\ B$ one uses $StateTBX\ Id\ S$ where $S \subseteq A \times B \times C$ is the set of 'consistent triples' (a, b, c), in the sense that $sl.put_R\ (a, c) = (b, c)$ and $sl.put_L\ (b, c) = (a, c)$:

$$\begin{aligned}&BX\ get_L\ set_L\ get_L\ set_R\ \mathbf{where}\\ &\quad get_L\ \ \ \ = \mathbf{do}\ \{\, (a, b, c) \leftarrow get; return\ a\,\}\\ &\quad get_R\ \ \ \ = \mathbf{do}\ \{\, (a, b, c) \leftarrow get; return\ b\,\}\\ &\quad set_L\ a' = \mathbf{do}\ \{\, (a, b, c) \leftarrow get; \mathbf{let}\ (b', c') = sl.put_R\ (a, c); set\ (a', b', c')\,\}\\ &\quad set_R\ b' = \mathbf{do}\ \{\, (a, b, c) \leftarrow get; \mathbf{let}\ (a', c') = sl.put_L\ (b, c); set\ (a', b', c')\,\}\end{aligned}$$

Asymmetric lenses generalise straightforwardly to accommodate effects in an underlying monad too. One can define

$$\begin{aligned}\mathbf{data}\ MLens\ \tau\ \alpha\ \beta = MLens\ \{\, &mview\ \ \ :: \alpha \to \beta,\\ &mupdate :: \alpha \to \beta \to \tau\ \alpha\,\}\end{aligned}$$

with corresponding notions of well-behaved and very-well-behaved monadic lens. (Diviánszky [8] and Pacheco et al. [28], among others, have proposed similar notions.) However, it turns out not to be straightforward to establish a corresponding notion of 'monadic symmetric lens' incorporating other effects. In this paper, we take a different approach to combining symmetry and effects; we defer further discussion of the different approaches to *MLens*es and the complications involved in extending symmetric lenses with effects to a future paper.

4 Composition

An obviously crucial question is whether well-behaved monadic bx compose. They do, but the issue is more delicate than might at first be expected. Of course, we cannot expect arbitrary BX to compose, because arbitrary monads do not. Here, we present one successful approach for *StateTBX*, based on lifting the component operations on different state types (but the same underlying monad of effects) into a common compound state.

Definition 4.1 (*StateT* embeddings from lenses). Given a lens from A to B, we can embed a *StateT* computation on the narrower type B into another computation on the wider type A, wrt the same underlying monad T:

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

Essentially, $\vartheta\ l\ m$ uses l to get a view b of the source a, runs m to get a return value c and updated view b', uses l to update the view yielding an updated source a', and returns c.

Lemma 4.2. If l::$\mathit{Lens}\ A\ B$ is very well-behaved, then $\vartheta\ l$ is a monad morphism. ♢

Definition 4.3. By Lemma 4.2, and since *fstLens* and *sndLens* are very well-behaved, we have the following monad morphisms lifting stateful computations to a product state space:

$$
\begin{aligned}
&\mathit{left} \ :: \mathit{Monad}\ \tau \Rightarrow \mathit{StateT}\ \sigma_1\ \tau\ \alpha \to \mathit{StateT}\ (\sigma_1, \sigma_2)\ \tau\ \alpha \\
&\mathit{left} \ = \vartheta\ \mathit{fstLens} \\
&\mathit{right} :: \mathit{Monad}\ \tau \Rightarrow \mathit{StateT}\ \sigma_2\ \tau\ \alpha \to \mathit{StateT}\ (\sigma_1, \sigma_2)\ \tau\ \alpha \\
&\mathit{right} = \vartheta\ \mathit{sndLens}
\end{aligned}
$$ ♢

Definition 4.4. For $bx_1 :: \mathit{StateTBX}\ \ T\ S_1\ A\ B$, $bx_2 :: \mathit{StateTBX}\ \ T\ S_2\ B\ C$, define the join $S_1\ {}_{bx_1}\!\bowtie_{bx_2} S_2$ informally as the subset of $S_1 \times S_2$ consisting of the pairs (s_1, s_2) in which observing the middle component of type B in state s_1 yields the same result as in state s_2. We might express this set-theoretically as follows:

$$S_1\ {}_{bx_1}\!\bowtie_{bx_2} S_2 = \{\, (s_1, s_2) \mid \mathit{eval}\ (bx_1.\mathit{get}_R)\ s_1 = \mathit{eval}\ (bx_2.\mathit{get}_L)\ s_2 \,\}$$

More generally, one could state a categorical definition in terms of pullbacks. In Haskell, we can only work with the coarser type of raw pairs (S_1, S_2). Note that the equation in the set comprehension compares two computations of type $T\ B$; but if the bx are transparent, and *return* injective as per Remark 3.9, then the definition simplifies to:

$$S_1\ {}_{bx_1}\!\bowtie_{bx_2} S_2 = \{\, (s_1, s_2) \mid bx_1.\mathit{read}_R\ s_1 = bx_2.\mathit{read}_L\ s_2 \,\}$$

The notation $S_1\ {}_{bx_1}\!\bowtie_{bx_2} S_2$ explicitly mentions bx_1 and bx_2, but we usually just write $S_1 \bowtie S_2$. No confusion should arise from using the same symbol to denote the consistent pairs of a single bx, as we did in Remark 3.6. ♢

Definition 4.5. Using *left* and *right*, we can define composition by:

$$
\begin{aligned}
&(\fatsemi) :: \mathit{Monad}\ \tau \Rightarrow \\
&\qquad \mathit{StateTBX}\ \sigma_1\ \tau\ \alpha\ \beta \to \mathit{StateTBX}\ \sigma_2\ \tau\ \beta\ \gamma \to \mathit{StateTBX}\ (\sigma_1 \bowtie \sigma_2)\ \tau\ \alpha\ \gamma \\
&bx_1 \fatsemi bx_2 = \mathit{BX}\ \mathit{get}_L\ \mathit{set}_L\ \mathit{get}_R\ \mathit{set}_R\ \mathbf{where} \\
&\quad \mathit{get}_L \ = \mathbf{do}\ \{\, \mathit{left}\ (bx_1.\mathit{get}_L) \,\}
\end{aligned}
$$

$$
\begin{aligned}
&get_R \quad = \mathbf{do}\ \{\, right\ (bx_2.get_R)\,\} \\
&set_L\ a = \mathbf{do}\ \{\, left\ (bx_1.set_L\ a);\ b \leftarrow left\ (bx_1.get_R);\ right\ (bx_2.set_L\ b)\,\} \\
&set_R\ c = \mathbf{do}\ \{\, right\ (bx_2.set_R\ c);\ b \leftarrow right\ (bx_2.get_L);\ left\ (bx_1.set_R\ b)\,\}
\end{aligned}
$$

Essentially, to set the left-hand side of the composed bx, we first set the left-hand side of the left component bx_1, then get bx_1's b-value, and set the left-hand side of bx_2 to this value; and similarly on the right. Note that the composition maintains the invariant that the compound state is in the subset $\sigma_1 \bowtie \sigma_2$ of $\sigma_1 \times \sigma_2$. ♢

Theorem 4.6 (transparent composition). Given transparent (and hence well-behaved) $bx_1 :: StateTBX\ S_1\ T\ A\ B$ and $bx_2 :: StateTBX\ S_2\ T\ B\ C$, their composition $bx_1 \fatsemi bx_2 :: StateTBX\ (S_1 \bowtie S_2)\ T\ A\ C$ is also transparent. ♢

Remark 4.7. Unpacking and simplifying the definitions, we have:

$$
\begin{aligned}
&bx_1 \fatsemi bx_2 = BX\ get_L\ set_L\ get_R\ set_R\ \mathbf{where} \\
&\quad get_L \quad = \mathbf{do}\ \{\, (s_1, \_) \leftarrow get;\ return\ (bx_1.read_L\ s_1)\,\} \\
&\quad get_R \quad = \mathbf{do}\ \{\, (\_, s_2) \leftarrow get;\ return\ (bx_2.read_R\ s_2)\,\} \\
&\quad set_L\ a' = \mathbf{do}\ \{\, (s_1, s_2) \leftarrow get; \\
&\qquad\qquad\qquad ((), s_1') \leftarrow lift\ (bx_1.set_L\ a'\ s_1); \\
&\qquad\qquad\qquad \mathbf{let}\ b = bx_1.read_R\ s_1'; \\
&\qquad\qquad\qquad ((), s_2') \leftarrow lift\ (bx_2.set_L\ b\ s_2); \\
&\qquad\qquad\qquad set\ (s_1', s_2')\,\} \\
&\quad set_R\ c' = \mathbf{do}\ \{\, (s_1, s_2) \leftarrow get; \\
&\qquad\qquad\qquad ((), s_2') \leftarrow lift\ (bx_2.set_R\ c'\ s_2); \\
&\qquad\qquad\qquad \mathbf{let}\ b = bx_2.read_L\ s_2'; \\
&\qquad\qquad\qquad ((), s_1') \leftarrow lift\ (bx_1.set_R\ b\ s_1); \\
&\qquad\qquad\qquad set\ (s_1', s_2')\,\}
\end{aligned}
$$
♢

Remark 4.8. Allowing effectful *get*s turns out to impose appreciable extra technical difficulty. In particular, while it still appears possible to prove that composition preserves well-behavedness, the identity laws of composition do not appear to hold in general. At the same time, we currently lack compelling examples that motivate effectful *get*s; the only example we have considered that requires this capability is Example 5.11 in Section 5. This is why we mostly limit attention to transparent bx. ♢

Composition is usually expected to be associative and to satisfy identity laws. We can define a family of identity bx as follows:

Definition 4.9 (identity). For any underlying monad instance, we can form the *identity* bx as follows:

$$
\begin{aligned}
&identity :: Monad\ \tau \Rightarrow StateTBX\ \tau\ \alpha\ \alpha\ \alpha \\
&identity = BX\ get\ set\ get\ set
\end{aligned}
$$

Clearly, this bx is well-behaved, indeed transparent, and overwritable. ♢

However, if we ask whether $bx = \mathit{identity} \fatsemi bx$, we are immediately faced with a problem: the two bx do not even have the same state types. Similarly when we ask about associativity of composition. Apparently, therefore, as for symmetric lenses [13], we must satisfy ourselves with equality only up to some notion of equivalence, such as the one introduced in Definition 3.4.

Theorem 4.10. Composition of transparent bx satisfies the identity and associativity laws, modulo $\equiv$.

$$\begin{array}{ll} \text{(Identity)} & \mathit{identity} \fatsemi bx \equiv bx \equiv bx \fatsemi \mathit{identity} \\ \text{(Assoc)} & bx_1 \fatsemi (bx_2 \fatsemi bx_3) \equiv (bx_1 \fatsemi bx_2) \fatsemi bx_3 \end{array}$$ ♢

5 Examples

We now show how to use and combine bx, and discuss how to extend our approach to support initialisation. We adapt some standard constructions on symmetric lenses, involving pairs, sums and lists. Finally we investigate some effectful bx primitives and combinators, culminating with the two examples from Section 1.

5.1 Initialisation

Readers familiar with bx will have noticed that so far we have not mentioned mechanisms for initialisation, e.g. 'create' for asymmetric lenses [10], 'missing' in symmetric lenses [13], or Ω in relational bx terminology [32]. Moreover, as we shall see in Section 5.2, initialisation is also needed for certain combinators.

Definition 5.1. An *initialisable StateTBX* is a *StateTBX* with two additional operations for initialisation:

$$\begin{array}{l} \mathbf{data}\ \mathit{InitStateTBX}\ \tau\ \sigma\ \alpha\ \beta = \mathit{InitStateTBX}\ \{ \\ \quad \mathit{get}_L :: \mathit{StateT}\ \sigma\ \tau\ \alpha, \quad \mathit{set}_L :: \alpha \to \mathit{StateT}\ \sigma\ \tau\ (), \quad \mathit{init}_L :: \alpha \to \tau\ \sigma, \\ \quad \mathit{get}_R :: \mathit{StateT}\ \sigma\ \tau\ \beta, \quad \mathit{set}_R :: \beta \to \mathit{StateT}\ \sigma\ \tau\ (), \quad \mathit{init}_R :: \beta \to \tau\ \sigma\} \end{array}$$

The init_L and init_R operations build an initial state from one view or the other, possibly incurring effects in the underlying monad. Well-behavedness of the bx requires in addition:

$$\begin{array}{ll} (\mathrm{I_L G_L}) & \mathbf{do}\ \{s \leftarrow bx.\mathit{init}_L\ a; bx.\mathit{get}_L\ s\} \\ & \quad = \mathbf{do}\ \{s \leftarrow bx.\mathit{init}_L\ a; \mathit{return}\ (a, s)\} \\ (\mathrm{I_R G_R}) & \mathbf{do}\ \{s \leftarrow bx.\mathit{init}_R\ b; bx.\mathit{get}_R\ s\} \\ & \quad = \mathbf{do}\ \{s \leftarrow bx.\mathit{init}_R\ b; \mathit{return}\ (b, s)\} \end{array}$$

stating informally that initialising then getting yields the initialised value. There are no laws concerning initialising then setting. ♢

We can extend composition to handle initialisation as follows:

$$(bx_1 \fatsemi bx_2).init_L\ a = \mathbf{do}\ \{ s_1 \leftarrow bx_1.init_L\ a; b \leftarrow bx_1.get_R\ s_1; \\ s_2 \leftarrow bx_2.init_L\ b; return\ (s_1, s_2)\}$$

and symmetrically for $init_R$. We refine the notions of bx isomorphism and equivalence to *InitStateTBX* as follows. A monad isomorphism $\iota :: StateT\ S_1\ T \to StateT\ S_2\ T$ amounts to a bijection $h :: S_1 \to S_2$ on the state spaces. An isomorphism of *InitStateTBX*s consists of such an ι and h satisfying the following equations (and their duals):

$$\begin{aligned} &\iota\ (bx_1.get_L) &&= bx_2.get_L \\ &\iota\ (bx_1.set_L\ a) &&= bx_2.set_L\ a \\ &\mathbf{do}\ \{ s \leftarrow bx_1.init_L\ a; return\ (h\ s)\} &&= bx_2.init_L\ a \end{aligned}$$

Note that the first two equations (and their duals) imply that ι is a conventional isomorphism between the underlying bx structures of bx_1 and bx_2 if we ignore the initialisation operations. The third equation simply says that h maps the state obtained by initialising bx_1 with a to the state obtained by initialising bx_2 with a. Equivalence of *InitStateTBX*s amounts to the existence of such an isomorphism.

Remark 5.2. Of course, there may be situations where these operations are not what is desired. We might prefer to provide both view values and ask the bx system to find a suitable hidden state consistent with both at once. This can be accommodated, by providing a third initialisation function:

$$initBoth :: \alpha \to \beta \to \tau\ (Maybe\ \sigma)$$

However, *initBoth* and $init_L$, $init_R$ are not interdefinable: *initBoth* requires both initial values, so is no help in defining a function that has access only to one; and conversely, given both initial values, there are in general two different ways to initialise from one of them (and two more to initialise from one and then set with the other). Furthermore, it is not clear how to define *initBoth* for the composition of two bx equipped with *initBoth*. ♢

5.2 Basic constructions and combinators

It is obviously desirable – and essential in the design of any future bx programming language – to be able to build up bx from components using combinators that preserve interesting properties, and therefore avoid having to prove well-behavedness from scratch for each bx. Symmetric lenses [13] admit several standard constructions, involving constants, duality, pairing, sum types, and lists. We show that these constructions can be generalised to *StateTBX*, and establish that they preserve well-behavedness. For most combinators, the initialisation operations are straightforward; in the interests of brevity, they and obvious duals are omitted in what follows.

Definition 5.3 (duality). Trivially, we can dualise any bx:

$$\begin{aligned}&\mathit{dual} :: \mathit{StateTBX}\ \tau\ \sigma\ \alpha\ \beta \to \mathit{StateTBX}\ \tau\ \sigma\ \beta\ \alpha\\ &\mathit{dual}\ bx = \mathit{BX}\ bx.\mathit{get}_R\ bx.\mathit{set}_R\ bx.\mathit{get}_L\ bx.\mathit{set}_L\end{aligned}$$

This simply exchanges the left and right operations; it preserves transparency and overwritability of the underlying bx. ♢

Definition 5.4 (constant and pair combinators). *StateTBX* also admits constant, pairing and projection operations:

$$\begin{aligned}&\mathit{constBX} :: \mathit{Monad}\ \tau \Rightarrow \alpha \to \mathit{StateTBX}\ \tau\ \alpha\ ()\ \alpha\\ &\mathit{fstBX} \quad :: \mathit{Monad}\ \tau \Rightarrow \mathit{StateTBX}\ \tau\ (\alpha,\beta)\ (\alpha,\beta)\ \alpha\\ &\mathit{sndBX} \quad :: \mathit{Monad}\ \tau \Rightarrow \mathit{StateTBX}\ \tau\ (\alpha,\beta)\ (\alpha,\beta)\ \beta\end{aligned}$$

These straightforwardly generalise to bx the corresponding operations for symmetric lenses. If they are to be initialisable, *fstBX* and *sndBX* also have to take a parameter for the initial value of the opposite side:

$$\begin{aligned}&\mathit{fstIBX} \ :: \mathit{Monad}\ \tau \Rightarrow \beta \to \mathit{InitStateTBX}\ \tau\ (\alpha,\beta)\ (\alpha,\beta)\ \alpha\\ &\mathit{sndIBX} :: \mathit{Monad}\ \tau \Rightarrow \alpha \to \mathit{InitStateTBX}\ \tau\ (\alpha,\beta)\ (\alpha,\beta)\ \beta\end{aligned}$$

Pairing is defined as follows:

$$\begin{aligned}&\mathit{pairBX} :: \mathit{Monad}\ \tau \Rightarrow \mathit{StateTBX}\ \tau\ \sigma_1\ \alpha_1\ \beta_1 \to \mathit{StateTBX}\ \tau\ \sigma_2\ \alpha_2\ \beta_2 \to\\ &\qquad \mathit{StateTBX}\ \tau\ (\sigma_1,\sigma_2)\ (\alpha_1,\alpha_2)\ (\beta_1,\beta_2)\\ &\mathit{pairBX}\ bx_1\ bx_2 = \mathit{BX}\ \mathit{gl}\ \mathit{sl}\ \mathit{gr}\ \mathit{sr}\ \textbf{where}\\ &\quad \mathit{gl} = \textbf{do}\ \{\, a_1 \leftarrow \mathit{left}\ (bx_1.\mathit{get}_L); a_2 \leftarrow \mathit{right}\ (bx_2.\mathit{get}_L); \mathit{return}\ (a_1,a_2)\,\}\\ &\quad \mathit{sl}\ (a_1,a_2) = \textbf{do}\ \{\, \mathit{left}\ (bx_1.\mathit{set}_L\ a_1); \mathit{right}\ (bx_2.\mathit{set}_L\ a_2)\,\}\\ &\quad \mathit{gr} \qquad = \ldots \quad \text{-- dual}\\ &\quad \mathit{sr} \qquad = \ldots \quad \text{-- dual}\end{aligned}$$

♢

Other operations based on isomorphisms, such as associativity of pairs, can be lifted to *StateTBX*s without problems. Well-behavedness is immediate for *constBX*, *fstBX*, *sndBX* and for any other bx that can be obtained from an asymmetric or symmetric lens. For the *pairBX* combinator we need to verify preservation of transparency:

Proposition 5.5. If bx_1 and bx_2 are transparent (and hence well-behaved), then so is $\mathit{pairBX}\ bx_1\ bx_2$. ♢

Remark 5.6. The pair combinator does not necessarily preserve overwritability. For this to be the case, we need to be able to commute the *set* operations of the component bx, including any effects in T. Moreover, the pairing combinator is not in general uniquely determined for non-commutative T, because the effects of bx_1 and bx_2 can be applied in different orders. ♢

Definition 5.7 (sum combinators). Similarly, we can define combinators analogous to the 'retentive sum' symmetric lenses and injection operations [13]. The injection operations relate an α and either the same α or some unrelated β; the old α value of the left side is retained when the right side is a β.

$$\begin{aligned}
&\mathit{inlBX} :: \mathit{Monad}\ \tau \Rightarrow \alpha \to \mathit{StateTBX}\ \tau\ (\alpha, \mathit{Maybe}\ \beta)\ \alpha\ (\mathit{Either}\ \alpha\ \beta)\\
&\mathit{inrBX} :: \mathit{Monad}\ \tau \Rightarrow \beta \to \mathit{StateTBX}\ \tau\ (\beta, \mathit{Maybe}\ \alpha)\ \beta\ (\mathit{Either}\ \alpha\ \beta)
\end{aligned}$$

The *sumBX* combinator combines two underlying bx and allows switching between them; the state of both (including that of the bx that is not currently in focus) is retained.

$$\begin{aligned}
&\mathit{sumBX} :: \mathit{Monad}\ \tau \Rightarrow \mathit{StateTBX}\ \tau\ \sigma_1\ \alpha_1\ \beta_1 \to \mathit{StateTBX}\ \tau\ \sigma_2\ \alpha_2\ \beta_2 \to\\
&\qquad \mathit{StateTBX}\ \tau\ (\mathit{Bool}, \sigma_1, \sigma_2)\ (\mathit{Either}\ \alpha_1\ \alpha_2)\ (\mathit{Either}\ \beta_1\ \beta_2)\\
&\mathit{sumBX}\ bx_1\ bx_2 = \mathit{BX}\ gl\ sl\ gr\ sr\ \mathbf{where}\\
&\quad gl = \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get};\\
&\qquad\qquad \mathbf{if}\ b\ \mathbf{then}\ \mathbf{do}\ \{(a_1, _) \leftarrow \mathit{lift}\ (bx_1.\mathit{get}_L\ s_1); \mathit{return}\ (\mathit{Left}\ a_1)\}\\
&\qquad\qquad\quad \mathbf{else}\ \ \mathbf{do}\ \{(a_2, _) \leftarrow \mathit{lift}\ (bx_2.\mathit{get}_L\ s_2); \mathit{return}\ (\mathit{Right}\ a_2)\}\}\\
&\quad sl\ (\mathit{Left}\ a_1)\ \ = \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get};\\
&\qquad\qquad\qquad\quad ((), s_1') \leftarrow \mathit{lift}\ ((bx_1.\mathit{set}_L\ a_1)\ s_1);\\
&\qquad\qquad\qquad\quad \mathit{set}\ (\mathit{True}, s_1', s_2)\}\\
&\quad sl\ (\mathit{Right}\ a_2) = \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get};\\
&\qquad\qquad\qquad\quad ((), s_2') \leftarrow \mathit{lift}\ ((bx_2.\mathit{set}_L\ a_2)\ s_2);\\
&\qquad\qquad\qquad\quad \mathit{set}\ (\mathit{False}, s_1, s_2')\}\\
&\quad gr = \ldots \quad \text{-- dual}\\
&\quad sr = \ldots \quad \text{-- dual}
\end{aligned}$$

♢

Proposition 5.8. If bx_1 and bx_2 are transparent, then so is $\mathit{sumBX}\ bx_1\ bx_2$.

♢

Finally, we turn to building a bx that operates on lists from one that operates on elements. The symmetric lens list combinators [13] implicitly regard the length of the list as data that is shared between the two views. The forgetful list combinator forgets all data beyond the current length. The retentive version maintains list elements beyond the current length, so that they can be restored if the list is lengthened again. We demonstrate the (more interesting) retentive version, making the shared list length explicit. Several other variants are possible.

Definition 5.9 (retentive list combinator). This combinator relies on the initialisation functions to deal with the case where the new values are inserted into the list, because in this case we need the capability to create new values on the other side (and new states linking them).

$$\begin{aligned}
&\mathit{listIBX} :: \mathit{Monad}\ \tau \Rightarrow\\
&\qquad \mathit{InitStateTBX}\ \tau\ \sigma\ \alpha\ \beta \to \mathit{InitStateTBX}\ \tau\ (\mathit{Int}, [\sigma])\ [\alpha]\ [\beta]\\
&\mathit{listIBX}\ bx = \mathit{InitStateTBX}\ gl\ sl\ il\ gr\ sr\ ir\ \mathbf{where}
\end{aligned}$$

$$
\begin{array}{l}
gl \quad = \mathbf{do}\ \{(n, cs) \leftarrow get; mapM\ (lift \cdot eval\ bx.get_L)\ (take\ n\ cs)\} \\
sl\ as = \mathbf{do}\ \{(\_, cs) \leftarrow get; \\
\qquad\qquad cs' \leftarrow lift\ (sets\ (exec \cdot bx.set_L)\ bx.init_L\ as\ cs); \\
\qquad\qquad set\ (length\ as, cs')\} \\
il\ as = \mathbf{do}\ \{cs \leftarrow mapM\ (bx.init_L)\ as; return\ (length\ as, cs)\} \\
gr \quad = \ldots \quad \text{-- dual} \\
sr\ bs = \ldots \quad \text{-- dual} \\
ir\ bs = \ldots \quad \text{-- dual}
\end{array}
$$

Here, the standard Haskell function *mapM* sequences a list of computations, and *sets* sequentially updates a list of states from a list of views, retaining any leftover states if the view list is shorter:

$$
\begin{array}{l}
sets :: Monad\ \tau \Rightarrow (\alpha \to \gamma \to \tau\ \gamma) \to (\alpha \to \tau\ \gamma) \to [\alpha] \to [\gamma] \to \tau\ [\gamma] \\
sets\ set\ init\ [\,]\ \quad cs\ \quad = return\ cs \\
sets\ set\ init\ (x : xs)\ (c : cs) = \mathbf{do}\ \{c' \leftarrow set\ x\ c; cs' \leftarrow sets\ set\ init\ xs\ cs; \\
\qquad\qquad\qquad\qquad\qquad\qquad return\ (c' : cs')\} \\
sets\ set\ init\ xs\ \quad [\,]\ \quad = mapM\ init\ xs
\end{array}
$$

♢

Proposition 5.10. If bx is transparent, then so is $listIBX\ bx$. ♢

5.3 Effectful bx

We now consider examples of bx that make nontrivial use of monadic effects. The careful consideration we paid earlier to the requirements for composability give rise to some interesting and non-obvious constraints on the definitions, which we highlight as we go.

For accessibility, we use specific monads in the examples in order to state and prove properties; for generality, the accompanying code abstracts from specific monads using Haskell type class constraints instead. In the interests of brevity, we omit dual cases and initialisation functions, but these are defined in the online code supplement.

Example 5.11 (environment). The *Reader* or environment monad is useful for modelling global parameters. Some classes of bidirectional transformations are naturally parametrised; for example, Voigtländer *et al.*'s approach [37] uses a *bias* parameter to determine how to merge changes back into lists.

Suppose we have a family of bx indexed by some parameter γ, over a monad $Reader\ \gamma$. Then we can define

$$
\begin{array}{l}
switch :: (\gamma \to StateTBX\ (Reader\ \gamma)\ \sigma\ \alpha\ \beta) \to StateTBX\ (Reader\ \gamma)\ \sigma\ \alpha\ \beta \\
switch\ f = BX\ gl\ sl\ gr\ sr\ \mathbf{where} \\
\quad gl \quad = \mathbf{do}\ \{c \leftarrow lift\ ask; (f\ c).get_L\} \\
\quad sl\ a = \mathbf{do}\ \{c \leftarrow lift\ ask; (f\ c).set_L\ a\} \\
\quad gr \quad = \ldots \quad \text{-- dual} \\
\quad sr \quad = \ldots \quad \text{-- dual}
\end{array}
$$

where the standard $ask :: Reader\ \gamma$ operation reads the γ value. ♢

Proposition 5.12. If $f\ c :: \mathit{StateTBX}\ (\mathit{Reader}\ C)\ S\ A\ B$ is transparent for any $c :: C$, then $\mathit{switch}\ f$ is a well-behaved, but not necessarily transparent, $\mathit{StateTBX}\ (\mathit{Reader}\ C)\ S\ A\ B$. ♢

Remark 5.13. Note that *switch f* is well-behaved but not necessarily transparent. This is because the *get* operations read not only from the designated state of the *StateTBX* but also from the *Reader* environment, and so they are not (*Reader C*)-pure. This turns out not to be a big problem in this case, because *Reader C* is a commutative monad. But suppose that the underlying monad were not *Reader C* but a non-commutative monad such as *State C*, maintaining some flag that may be changed by the *set* operations; in this scenario, it is not difficult to construct a counterexample to the identity laws for composition. Such counterexamples are why we have largely restricted attention in this paper to transparent bx. (Besides, one what argue that it is never necessary for the get operations to depend on the context; any such dependencies could be handled entirely by the set operations.) ♢

Example 5.14 (exceptions). We turn next to the possibility of failure. Conventionally, the functions defining a bx are required to be total, but often it is not possible to constrain the source and view types enough to make this literally true; for example, consider a bx relating two *Rational* views whose consistency relation is $\{(x, {}^1\!/_x) \mid x \neq 0\}$. A principled approach to failure is to use the *Maybe* (exception) monad, so that an attempt to divide by zero yields *Nothing*.

$$
\begin{array}{l}
\mathit{invBX} :: \mathit{StateTBX}\ \mathit{Maybe}\ \mathit{Rational}\ \mathit{Rational}\ \mathit{Rational} \\
\mathit{invBX} = \mathit{BX}\ \mathit{get}\ \mathit{set}_L\ (\mathit{gets}\ (\lambda a.\, {}^1\!/_a))\ \mathit{set}_R\ \mathbf{where} \\
\quad \mathit{set}_L\ a = \mathbf{do}\ \{\mathit{lift}\ (\mathit{guard}\ (a \not\equiv 0)); \mathit{set}\ a\} \\
\quad \mathit{set}_R\ b = \mathbf{do}\ \{\mathit{lift}\ (\mathit{guard}\ (b \not\equiv 0)); \mathit{set}\ ({}^1\!/_b)\}
\end{array}
$$

where $\mathit{guard}\ b = \mathbf{do}\ \{\mathbf{if}\ b\ \mathbf{then}\ \mathit{Just}\ ()\ \mathbf{else}\ \mathit{Nothing}\}$ is a standard operation in the *Maybe* monad. As another example, suppose we know that A is in the *Read* and *Show* type classes, so each A value can be printed to and possibly read from a string. We can define:

$$
\begin{array}{l}
\mathit{readSomeBX} :: (\mathit{Read}\ \alpha, \mathit{Show}\ \alpha) \Rightarrow \mathit{StateTBX}\ \mathit{Maybe}\ (\alpha, \mathit{String})\ \alpha\ \mathit{String} \\
\mathit{readSomeBX} = \mathit{BX}\ (\mathit{gets}\ \mathit{fst})\ \mathit{set}_L\ (\mathit{gets}\ \mathit{snd})\ \mathit{set}_R\ \mathbf{where} \\
\quad \mathit{set}_L\ a' = \mathit{set}\ (a', \mathit{show}\ a') \\
\quad \mathit{set}_R\ b' = \mathbf{do}\ \{(\_, b) \leftarrow \mathit{get}; \\
\qquad\qquad \mathbf{if}\ b == b'\ \mathbf{then}\ \mathit{return}\ ()\ \mathbf{else\ case}\ \mathit{reads}\ b'\ \mathbf{of} \\
\qquad\qquad\quad ((a', \texttt{""}) : \_) \rightarrow \mathit{set}\ (a', b') \\
\qquad\qquad\quad \_ \qquad\qquad \rightarrow \mathit{lift}\ \mathit{Nothing}\}
\end{array}
$$

(The function *reads* returns a list of possible parses with remaining text.) Note that the get operations are *Maybe*-pure: if there is a *Read* error, it is raised instead by the set operations.

The same approach can be generalised to any monad T having a polymorphic error value $\mathit{err} :: \forall \alpha.\, T\ \alpha$ and any pair of partial inverse functions $f :: A \rightarrow \mathit{Maybe}\ B$ and $g :: B \rightarrow \mathit{Maybe}\ A$ (*i.e.*, $f\ a = \mathit{Just}\ b$ if and only if $g\ b = \mathit{Just}\ a$, for all a, b):

$$
\begin{aligned}
&\mathit{partialBX} :: \mathit{Monad}\ \tau \Rightarrow (\forall\alpha.\tau\ \alpha) \to (\alpha \to \mathit{Maybe}\ \beta) \to (\beta \to \mathit{Maybe}\ \alpha) \to \\
&\qquad \mathit{StateTBX}\ \tau\ (\alpha,\beta)\ \alpha\ \beta \\
&\mathit{partialBX}\ \mathit{err}\ f\ g = \mathit{BX}\ (\mathit{gets}\ \mathit{fst})\ \mathit{set}_L\ (\mathit{gets}\ \mathit{snd})\ \mathit{set}_R\ \mathbf{where} \\
&\quad \mathit{set}_L\ a' = \mathbf{case}\ f\ a'\ \mathbf{of}\ \mathit{Just}\ b' \to \mathit{set}\ (a', b') \\
&\qquad\qquad\qquad\qquad\qquad\ \mathit{Nothing} \to \mathit{lift}\ \mathit{err} \\
&\quad \mathit{set}_R\ b' = \mathbf{case}\ g\ b'\ \mathbf{of}\ \mathit{Just}\ a' \to \mathit{set}\ (a', b') \\
&\qquad\qquad\qquad\qquad\qquad\ \mathit{Nothing} \to \mathit{lift}\ \mathit{err}
\end{aligned}
$$

Then we could define *invBX* and a stricter variation of *readSomeBX* (one that will *read* only a string that it *show*s—rejecting alternative renderings, whitespace, and so on) as instances of *partialBX*. ♢

Proposition 5.15. Let $f :: A \to \mathit{Maybe}\ B$ and $g :: B \to \mathit{Maybe}\ A$ be partial inverses and let *err* be a zero element for T. Then $\mathit{partialBX}\ \mathit{err}\ f\ g :: \mathit{StateTBX}\ T\ S\ A\ B$ is well-behaved, where $S = \{\,(a, b) \mid f\ a = \mathit{Just}\ b\,\}$. ♢

Example 5.16 (nondeterminism—Scenario 1.1 revisited). For simplicity's sake, we model nondeterminism via the list monad: a 'nondeterministic function' from A to B is represented as a pure function of type $A \to [B]$. The following bx is parametrised on a predicate *ok* that checks consistency of two states, a fix-up function *bs* that returns the B values consistent with a given A, and symmetrically a fix-up function *as*.

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

Proposition 5.17. Given *ok*, $S = \{\,(a, b) \mid \mathit{ok}\ a\ b\,\}$, and *as* and *bs* satisfying

$$
\begin{aligned}
a \in \mathit{as}\ b &\Rightarrow \mathit{ok}\ a\ b \\
b \in \mathit{bs}\ a &\Rightarrow \mathit{ok}\ a\ b
\end{aligned}
$$

then $\mathit{nondetBX}\ \mathit{ok}\ \mathit{bs}\ \mathit{as} :: \mathit{StateTBX}\ [\,]\ S\ A\ B$ is well-behaved (indeed, it is clearly transparent). It is not necessary for the two conditions to be equivalences. ♢

Remark 5.18. Note that, in addition to choice, the list monad also allows for failure: the fix-up functions can return the empty list. From a semantic point of view, nondeterminism is usually modelled using the monad of finite nonempty sets. If we had used the nonempty set monad instead of lists, then failure would not be possible. ♢

Example 5.19 (signalling). We can define a bx that sends a signal every time either side changes:

$$
\begin{aligned}
&\mathit{signalBX} :: (\mathit{Eq}\ \alpha, \mathit{Eq}\ \beta, \mathit{Monad}\ \tau) \Rightarrow (\alpha \to \tau\ ()) \to (\beta \to \tau\ ()) \to \\
&\qquad \mathit{StateTBX}\ \tau\ \sigma\ \alpha\ \beta \to \mathit{StateTBX}\ \tau\ \sigma\ \alpha\ \beta \\
&\mathit{signalBX}\ \mathit{sigA}\ \mathit{sigB}\ \mathit{bx} = \mathit{BX}\ (\mathit{bx}.\mathit{get}_L)\ \mathit{sl}\ (\mathit{bx}.\mathit{get}_R)\ \mathit{sr}\ \mathbf{where} \\
&\quad \mathit{sl}\ a' = \mathbf{do}\ \{\, a \leftarrow \mathit{bx}.\mathit{get}_L; \mathit{bx}.\mathit{set}_L\ a'; \\
&\qquad\qquad \mathit{lift}\ (\mathbf{if}\ a \not\equiv a'\ \mathbf{then}\ \mathit{sigA}\ a'\ \mathbf{else}\ \mathit{return}\ ())\,\} \\
&\quad \mathit{sr}\ b' = \mathbf{do}\ \{\, b \leftarrow \mathit{bx}.\mathit{get}_R; \mathit{bx}.\mathit{set}_R\ b'; \\
&\qquad\qquad \mathit{lift}\ (\mathbf{if}\ b \not\equiv b'\ \mathbf{then}\ \mathit{sigB}\ b'\ \mathbf{else}\ \mathit{return}\ ())\,\}
\end{aligned}
$$

Note that *sl* checks to see whether the new value a' equals the old value a, and does nothing if so; only if they are different does it perform $\mathit{sigA}\ a'$. If the bx is to be well-behaved, then no action can be performed in the case that $a == a'$.

For example, instantiating the underlying monad to *IO* we have:

$$
\begin{aligned}
&\mathit{alertBX} :: (\mathit{Eq}\ \alpha, \mathit{Eq}\ \beta) \Rightarrow \mathit{StateTBX}\ \mathit{IO}\ \sigma\ \alpha\ \beta \to \mathit{StateTBX}\ \mathit{IO}\ \sigma\ \alpha\ \beta \\
&\mathit{alertBX} = \mathit{signalBX}\ (\lambda\_.\, \mathit{putStrLn}\ \texttt{"Left"})\ (\lambda\_.\, \mathit{putStrLn}\ \texttt{"Right"})
\end{aligned}
$$

which prints a message whenever one side changes. This is well-behaved; the *set* operations are side-effecting, but the side-effects only occur when the state is changed. It is not overwritable, because multiple changes may lead to different signals from a single change.

As another example, we can define a logging bx as follows:

$$
\begin{aligned}
&\mathit{logBX} :: (\mathit{Eq}\ \alpha, \mathit{Eq}\ \beta) \Rightarrow \mathit{StateTBX}\ (\mathit{Writer}\ [\mathit{Either}\ \alpha\ \beta])\ \sigma\ \alpha\ \beta \to \\
&\qquad\qquad \mathit{StateTBX}\ (\mathit{Writer}\ [\mathit{Either}\ \alpha\ \beta])\ \sigma\ \alpha\ \beta \\
&\mathit{logBX} = \mathit{signalBX}\ (\lambda a.\, \mathit{tell}\ [\mathit{Left}\ a])\ (\lambda b.\, \mathit{tell}\ [\mathit{Right}\ b])
\end{aligned}
$$

where $\mathit{tell} :: \sigma \to \mathit{Writer}\ \sigma\ ()$ is a standard operation in the *Writer* monad that writes a value to the output. This bx logs a list of all of the views as they are changed. Wrapping a component of a chain of composed bx with *log* can provide insight into how changes at the ends of the chain propagate through that component. If memory use is a concern, then we could limit the length of the list to record only the most recent updates – lists of bounded length also form a monoid. ♢

Proposition 5.20. If A and B are types equipped with a well-behaved notion of equality (in the sense that $(a == b) = \mathit{True}$ if and only if $a = b$), and $\mathit{bx} :: \mathit{StateTBX}\ T\ S\ A\ B$ is well-behaved, then $\mathit{signalBX}\ \mathit{sigA}\ \mathit{sigB}\ \mathit{bx} :: \mathit{StateTBX}\ T\ S\ A\ B$ is well-behaved. Moreover, *signalBX* preserves transparency. ♢

Example 5.21 (interaction—Scenario 1.2 revisited). For this example, we need to record both the current state (an A and a B) and the learned collection of consistency restorations. The latter is represented as two lists; the first list contains a tuple $((a', b), b')$ for each invocation of $\mathit{set}_L\ a'$ on a state $(\_, b)$ resulting in an updated state (a', b'); the second is symmetric, for $\mathit{set}_R\ b'$ invocations.

The types A and B must each support equality, so that we can check for previously asked questions. We abstract from the base monad; we parametrise the bx on two monadic functions, each somehow determining a consistent match for one state.

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

where $\mathit{fst3}\ (a,b,c) = a$. For example, the bx below finds matching states by asking the user, writing to and reading from the terminal.

$$
\begin{aligned}
&\mathit{dynamicIOBX} :: (\mathit{Eq}\ \alpha, \mathit{Eq}\ \beta, \mathit{Show}\ \alpha, \mathit{Show}\ \beta, \mathit{Read}\ \alpha, \mathit{Read}\ \beta) \Rightarrow \\
&\qquad \mathit{StateTBX}\ \mathit{IO}\ ((\alpha,\beta),[((\alpha,\beta),\beta)],[((\alpha,\beta),\alpha)])\ \alpha\ \beta \\
&\mathit{dynamicIOBX} = \mathit{dynamicBX}\ \mathit{matchIO}\ (\mathit{flip}\ \mathit{matchIO}) \\
&\mathit{matchIO} :: (\mathit{Show}\ \alpha, \mathit{Show}\ \beta, \mathit{Read}\ \beta) \Rightarrow \alpha \to \beta \to \mathit{IO}\ \beta \\
&\mathit{matchIO}\ a\ b = \mathbf{do}\ \{\mathit{putStrLn}\ (\texttt{"Setting "} +\!\!+ \mathit{show}\ a); \\
&\qquad\qquad\quad \mathit{putStr}\ (\texttt{"Replacement for "} +\!\!+ \mathit{show}\ b +\!\!+ \texttt{"?"}); \\
&\qquad\qquad\quad s \leftarrow \mathit{getLine}; \mathit{return}\ (\mathit{read}\ s)\}
\end{aligned}
$$

An alternative way to find matching states, for a finite state space, would be to search an enumeration $[\mathit{minBound}..\mathit{maxBound}]$ of the possible values, checking against a fixed oracle p:

$$
\begin{aligned}
&\mathit{dynamicSearchBX} :: \\
&\quad (\mathit{Eq}\ \alpha, \mathit{Eq}\ \beta, \mathit{Enum}\ \alpha, \mathit{Bounded}\ \alpha, \mathit{Enum}\ \beta, \mathit{Bounded}\ \beta) \Rightarrow \\
&\quad (\alpha \to \beta \to \mathit{Bool}) \to \\
&\quad \mathit{StateTBX}\ \mathit{Maybe}\ ((\alpha,\beta),[((\alpha,\beta),\beta)],[((\alpha,\beta),\alpha)])\ \alpha\ \beta \\
&\mathit{dynamicSearchBX}\ p = \mathit{dynamicBX}\ (\mathit{search}\ p)\ (\mathit{flip}\ (\mathit{search}\ (\mathit{flip}\ p))) \\
&\mathit{search} :: (\mathit{Enum}\ \beta, \mathit{Bounded}\ \beta) \Rightarrow (\alpha \to \beta \to \mathit{Bool}) \to \alpha \to \beta \to \mathit{Maybe}\ \beta \\
&\mathit{search}\ p\ a\ \_ = \mathit{find}\ (p\ a)\ [\mathit{minBound}..\mathit{maxBound}]
\end{aligned}
$$

♢

Proposition 5.22. For any f, g, the bx $\mathit{dynamicBX}\ f\ g$ is well-behaved (it is clearly transparent). ♢

6 Related work

Bidirectional programming This has a large literature; work on view update flourished in the early 1980s, and the term 'lens' was coined in 2005 [9]. The

GRACE report [6] surveys work since. We mention here only the closest related work.

Pacheco *et al.* [28] present 'putback-style' asymmetric lenses; *i.e.* their laws and combinators focus only on the 'put' functions, of type $Maybe\ s \to v \to m\ s$, for some monad m. This allows for effects, and they include a combinator *effect* that applies a monad morphism to a lens. Their laws assume that the monad m admits a membership operation $(\in) :: a \to m\ a \to Bool$. For monads such as *List* or *Maybe* that support such an operation, their laws are similar to ours, but their approach does not appear to work for other important monads such as *IO* or *State*. In Diviánsky's monadic lens proposal [8], the *get* function is monadic, so in principle it too can have side-effects; as we have seen, this possibility significantly complicates composition.

Johnson and Rosebrugh [16] analyse symmetric lenses in a general setting of categories with finite products, showing that they correspond to pairs of (asymmetric) lenses with a common source. Our composition for *StateTBX*s uses a similar idea; however, their construction does not apply directly to monadic lenses, because the Kleisli category of a monad does not necessarily have finite products. They also identify a different notion of equivalence of symmetric lenses.

Elsewhere, we have considered a coalgebraic approach to bx [2]. Relating such an approach to the one presented here, and investigating their associated equivalences, is an interesting future direction of research.

Macedo *et al.* [24] observe that most bx research deals with just two models, but many tools and specifications, such as QVT-R [27], allow relating multiple models. Our notion of bx generalises straightforwardly to such multidirectional transformations, provided we only update one source value at a time.

Monads and algebraic effects The vast literature on combining and reasoning about monads [17, 22, 23, 26] stems from Moggi's work [25]; we have shown that bidirectionality can be viewed as another kind of computational effect, so results about monads can be applied to bidirectional computation.

A promising area to investigate is the *algebraic* treatment of effects [29], particularly recent work on combining effects using operations such as sum and tensor [15] and *handlers* of algebraic effects [3, 19, 30]. It appears straightforward to view entangled state as generated by operations and equations analogous to the bx laws. What is less clear is whether operations such as composition can be defined in terms of effect handlers: so far, the theory underlying handlers [30] does not support 'tensor-like' combinations of computations. We therefore leave this investigation for future work.

The relationship between lenses and state monad morphisms is intriguing, and hints of it appear in previous work on *compositional references* by Kagawa [18]. The fact that lenses determine state monad morphisms (Definition 4.1) appears to be folklore; Shkaravska [31] stated this result in a talk, and it is implicit in the design of the Haskell `Data.Lens` library [20], but we are not aware of any previous published proof.

7 Conclusions and further work

We have presented a semantic framework for effectful bidirectional transformations (bx). Our framework encompasses symmetric lenses, which (as is well-known) in turn encompass other approaches to bx such as asymmetric lenses [10] and relational bx [32]; we have also given examples of other monadic effects. This is an advance on the state of the art of bidirectional transformations: ours is the first formalism to reconcile the stateful behavior of bx with other effects such as nondeterminism, I/O or exceptions with due attention paid to the corresponding laws. We have defined composition for effectful bx and shown that composition is associative and satisfies identity laws, up to a suitable notion of equivalence based on monad isomorphisms. We have also demonstrated some combinators suitable for grounding the design of future bx languages based on our approach.

In future we plan to investigate equivalence, and the relationship with the work of Johnson and Rosebrugh [16], further. The equivalence we present here is finer than theirs, and also finer than the equivalence for symmetric lenses presented by Hofmann *et al.* [13]. Early investigations, guided by an alternative coalgebraic presentation [2] of our framework, suggest that the situation for bx may be similar to that for processes given as labelled transition systems: it is possible to give many different equivalences which are 'right' according to different criteria. We think the one we have given here is the finest reasonable, equating just enough bx to make composition work. Another interesting area for exploration is formalisation of our (on-paper) proofs.

Our framework provides a foundation for future languages, libraries, or tools for effectful bx, and there are several natural next steps in this direction. In this paper we explored only the case where the get and set operations read or write complete states, but our framework allows for generalisation beyond the category Set and hence, perhaps, into delta-based bx [7], edit lenses [14] and ordered updates [12], in which the operations record state changes rather than complete states. Another natural next step is to explore different *witness structures* encapsulating the dependencies between views, in order to formulate candidate principles of Least Change (informally, that "a bx should not change more than it has to in order to restore consistency") that are more practical and flexible than those that can be stated in terms of views alone.

Acknowledgements

Preliminary work on this topic was presented orally at the BIRS workshop 13w5115 in December 2013; a four-page abstract [4] of some of the ideas in this paper appeared at the Athens BX Workshop in March 2014; and a short presentation on an alternative coalgebraic approach [2] was made at CMCS 2014. We thank the organisers of and participants at those meetings and the anonymous reviewers for their helpful comments. The work was supported by the UK EPSRC-funded project *A Theory of Least Change for Bidirectional Transformations* [35] (EP/K020218/1, EP/K020919/1).

References

1. Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: Notions of bidirectional computation and entangled state monads. Tech. rep., TLCBX project (2015), extended version with proofs, available from `http://groups.inf.ed.ac.uk/bx/`
2. Abou-Saleh, F., McKinna, J.: A coalgebraic approach to bidirectional transformations (2014), short presentation at CMCS
3. Brady, E.: Programming and reasoning with algebraic effects and dependent types. In: ICFP. pp. 133–144. ACM (2013)
4. Cheney, J., McKinna, J., Stevens, P., Gibbons, J., Abou-Saleh, F.: Entangled state monads (abstract). In: [34]
5. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: JTL: A bidirectional and change propagating transformation language. In: SLE 2010. LNCS, vol. 6563, pp. 183–202. Springer (2010)
6. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: A cross-discipline perspective. In: ICMT. LNCS, vol. 5563, pp. 260–283. Springer (2009)
7. Diskin, Z., Xiong, Y., Czarnecki, K.: From state- to delta-based bidirectional model transformations: the asymmetric case. JOT 10, 6: 1–25 (2011)
8. Diviánszky, P.: LGtk API correction. `http://people.inf.elte.hu/divip/LGtk/CorrectedAPI.html` (April 2013)
9. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view update problem. In: popl. pp. 233–246. ACM (2005)
10. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM TOPLAS 29(3), 17 (2007), extended version of [9]
11. Gibbons, J., Hinze, R.: Just **do** it: Simple monadic equational reasoning. In: ICFP. pp. 2–14. ACM (2011)
12. Hegner, S.J.: An order-based theory of updates for closed database views. Ann. Math. Art. Int. 40, 63–125 (2004)
13. Hofmann, M., Pierce, B.C., Wagner, D.: Symmetric lenses. In: POPL. pp. 371–384. ACM (2011)
14. Hofmann, M., Pierce, B.C., Wagner, D.: Edit lenses. In: POPL. pp. 495–508. ACM (2012)
15. Hyland, M., Plotkin, G.D., Power, J.: Combining effects: Sum and tensor. TCS 357(1-3), 70–99 (2006)
16. Johnson, M., Rosebrugh, R.: Spans of lenses. In: [34]
17. Jones, M.P., Duponcheel, L.: Composing monads. Tech. Rep. RR-1004, DCS, Yale (1993)
18. Kagawa, K.: Compositional references for stateful functional programming. In: ICFP. pp. 217–226 (1997)
19. Kammar, O., Lindley, S., Oury, N.: Handlers in action. In: ICFP. pp. 145–158. ACM (2013)
20. Kmett, E.: `Data.Lens` library, `http://hackage.haskell.org/package/lenses-0.1.2/docs/Data-Lenses.html`
21. Kmett, E.: `lens-4.0.4` library, `http://hackage.haskell.org/package/lens`
22. Liang, S., Hudak, P., Jones, M.P.: Monad transformers and modular interpreters. In: POPL. pp. 333–343 (1995)

23. Lüth, C., Ghani, N.: Composing monads using coproducts. In: ICFP. pp. 133–144. ACM (2002)
24. Macedo, N., Cunha, A., Pacheco, H.: Toward a framework for multidirectional model transformations. In: [34]
25. Moggi, E.: Notions of computation and monads. Inf.&Comp. 93(1), 55–92 (1991)
26. Mossakowski, T., Schröder, L., Goncharov, S.: A generic complete dynamic logic for reasoning about purity and effects. FAC 22(3-4), 363–384 (2010)
27. OMG: MOF 2.0 Query/View/Transformation specification (QVT), version 1.1 (January 2011), http://www.omg.org/spec/QVT/1.1/
28. Pacheco, H., Hu, Z., Fischer, S.: Monadic combinators for "putback" style bidirectional programming. In: PEPM. pp. 39–50. ACM (2014), http://doi.acm.org/10.1145/2543728.2543737
29. Plotkin, G.D., Power, J.: Notions of computation determine monads. In: FOSSACS. LNCS, vol. 2303, pp. 342–356. Springer (2002)
30. Plotkin, G.D., Pretnar, M.: Handling algebraic effects. LMCS 9(4) (2013)
31. Shkaravska, O.: Side-effect monad, its equational theory and applications (2005), seminar slides available at: http://www.ioc.ee/~tarmo/tsem05/shkaravska1512-slides.pdf
32. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. SoSyM 9(1), 7–20 (2010)
33. Stevens, P., McKinna, J., Cheney, J.: 'Composers' example. http://bx-community.wikidot.com/examples:composers (2014)
34. Terwilliger, J., Hidaka, S. (eds.): BX Workshop. http://ceur-ws.org/Vol-1133/#bx (2014)
35. TLCBX Project: A theory of least change for bidirectional transformations. http://www.cs.ox.ac.uk/projects/tlcbx/, http://groups.inf.ed.ac.uk/bx/ (2013–2016)
36. Varró, D.: Model transformation by example. In: MoDELS 2006. LNCS, vol. 4199, pp. 410–424. Springer (2006)
37. Voigtländer, J., Hu, Z., Matsuda, K., Wang, M.: Enhancing semantic bidirectionalization via shape bidirectionalizer plug-ins. JFP 23(5), 515–551 (2013)