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 'put' a view of the whole from that 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 'put' operations are stateful functions. This leads naturally to considering bx that exploit other computational effects too, such as I/O, nondeterminism, and failure, which have largely been 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, and provide a prototype implementation in Haskell along with several illustrative examples.

1 Introduction

Bidirectional transformations (bx) arise when synchronising data in two different data sources: updates to one source entail corresponding updates to the other, in order to maintain consistency. When the two data sources have isomorphic representations, this is a straightforward task; an update on either source can be matched by discarding and regenerating the other source. It becomes more interesting when one data representation records some information that is missing from the other; 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. Even within more MPC-oriented circles, there are several conflicting definitions and incompatible extensions, such as *lenses* [10], *relational bx* [32], *symmetric lenses* [13], *putback-based lenses* [28], or *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 stateful computations acting on pairs, representing the two data 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.

By way of motivation, let us mention two examples, drawn 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. Each example illustrates the use of effects in its consistency restoration strategy, in response to an update to a new A-state a' (or B-state b', symmetrically). We revisit them in Section 6 as concrete examples of effectful bx.

Scenario 1 (nondeterminism). Our first example concerns nondeterminism. Most formal notions of bx require that the transformation choose one answer deterministically when restoring consistency. The developers of the Janus Transformation Language [4] (among others) have observed that this is unsatisfactory. Programmers may not wish to specify precisely how consistency should be restored in advance, but instead, may prefer to specify a consistency relation, and permit the bx engine to resolve the underspecification nondeterministically. For example, consistency may be restored by invoking an external constraint solver (JTL happens to use answer-set programming), but many variations of this idea are possible. No previous formalism permits such *nondeterministic bx* to be composed with conventional deterministic transformations, or characterises the laws that such transformations ought to satisfy.

Strategy: Given a', the bx internally records a', and checks whether a' is consistent with the current B-state b. If it is, nothing further need be done; if not, the bx replaces b with a new B-state b' chosen nondeterministically from the set of all those consistent with a'. Updating the B-state is symmetric. ◇

Scenario 2 (interaction). This is a bidirectional version of "model transformation by example" [35], where the bx 'learns' gradually, by prompting its users over time, the desired way to restore consistency in various situations.

Strategy: The bx maintains a collection of known ways to restore consistency. Given a', it internally records a', and checks whether it already knows how to restore consistency between a' and the current b. If so, it does so without further ado. If not, it queries the user for a new b', perhaps having first appealed to outside agency to generate helpful suggestions. It records b', and updates its collection of restorations so that the same query is not asked again. ◇

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 the related issue of equivalence, and prove associativity and identity laws for composition. In Section 6 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. This technical report version includes an appendix with all proofs; executable code can be found on our project web page at: `http://groups.inf.ed.ac.uk/bx/`.

A note to reviewers: *We presented a four-page abstract [3] of a preliminary version of this material in December at the BX Workshop in March 2014. The abstract contains some of the definitions from Section 2 and Section 3 (with some differences), and one very brief example of a bidirectional transformation performing I/O; but the remaining material here on composition (Section 4), on equivalence (Section 5), and on other effects (Section 6) is new, and of course the whole paper is completely rewritten.*

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 **newtype**s; e.g., in Section 2.4 we omit the explicit witnesses to isomorphisms such as the function *runStateT* from $\mathit{StateT}\ S\ T\ A$ to $S \rightarrow T\ (A, S)$. We use a tightest-binding lowered dot for field access in records; e.g., in Section 3.3 we write $l.\mathit{mview}$ rather than $\mathit{mview}\ l$;

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

$$\begin{array}{l}
\mathbf{class}\ \mathit{Monad}\ \tau\ \mathbf{where} \\
\quad \mathit{return} :: \alpha \to \tau\ \alpha \\
\quad (\gg\!=)\ \ :: \tau\ \alpha \to (\alpha \to \tau\ \beta) \to \tau\ \beta \quad \text{-- pronounced `bind'}
\end{array}$$

A monad instance should satisfy the following laws:

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

◊

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

$$\begin{array}{lll}
\mathbf{type}\ \mathit{Id}\ \alpha & = \alpha & \text{-- no effects} \\
\mathbf{data}\ \mathit{Maybe}\ \alpha & = \mathit{Just}\ \alpha \mid \mathit{Nothing} & \text{-- failure/exceptions} \\
\mathbf{data}\ \mathit{List}\ \alpha & = \mathit{Nil} \mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha) & \text{-- choice} \\
\mathbf{type}\ \mathit{State}\ \sigma\ \alpha & = \sigma \to (\alpha, \sigma) & \text{-- state} \\
\mathbf{type}\ \mathit{Reader}\ \sigma\ \alpha & = \sigma \to \alpha & \text{-- environment} \\
\mathbf{type}\ \mathit{Writer}\ \sigma\ \alpha & = (\alpha, \sigma) & \text{-- logging}
\end{array}$$

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

Definition 4. In Haskell, monadic expressions may be written using **do** notation, which is defined by translation into applications of bind:

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

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. When the return value of m is not used – e.g., when void – we write **do** $\{m; ms\}$ as shorthand for **do** $\{\_ \leftarrow m; ms\}$ with its wildcard pattern. ◊

Definition (commutative monad). We say that $m :: T\ A$ *commutes in* T if the following holds for any $n :: T\ B$, for x, y distinct variables not free in m, n:

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

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

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

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

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

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

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

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

which satisfy the following four laws [29]:

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

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

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

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

♢

Lemma 6. If laws (GG) and (GS) are satisfied, then unused *get*s are discardable:

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

♢

Definition. Some convenient shorthands:

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

♢

Lemma 7 (liftings commute with *get* and *set*). Suppose a, b are distinct variables not appearing in expression m. Then:

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

♢

Definition. 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 $h :: S \rightarrow A$. 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. ♢

Definition 8 (data refinement). Given monads M of 'abstract computations' and M' of 'concrete computations', various 'abstract operations' $\mathit{op} :: A \rightarrow M\ B$ with corresponding 'concrete operations' $\mathit{op}' :: A \rightarrow M'\ B$, an 'abstraction function' $\mathit{abs} :: M'\ \alpha \rightarrow M\ \alpha$ and a 'reification function' $\mathit{conc} :: M\ \alpha \rightarrow M'\ \alpha$, we say that *conc* is a *data refinement* from (M, op) to (M', op') if:

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

♢

Remark. The point is that given such a data refinement, a composite abstract computation can be faithfully simulated by a concrete one:

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

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. ♢

Lemma 9. Given an arbitrary monad T, not assumed to be an instance of $StateT$, with operations $get_T :: T\ S$ and $set_T :: S \rightarrow T\ ()$ for a type S, such that get_T and set_T satisfy the laws (GG), (GS), and (SG) of Definition 5, 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

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

□

Remark. 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 10. 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 'put' back a modified view into an old source to yield an updated source.

$$\mathbf{data}\ Lens\ \alpha\ \beta = Lens\ \{view :: \alpha \rightarrow \beta, update :: \alpha \rightarrow \beta \rightarrow \alpha\}$$

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. 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; but in general, the V may be computed from and therefore depend on all of the S value. ♢

Lenses and state monads are related by the following observation.

Lemma 11. A very well-behaved lens $l :: \mathit{Lens}\ S\ V$ induces a monad morphism $\varphi :: \forall \alpha. \mathit{State}\ V\ \alpha \to \mathit{State}\ S\ \alpha$, defined by

$$\varphi\ m = \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ (a, v') = m\ (l.\mathit{view}\ s); \mathit{set}\ (l.\mathit{update}\ s\ v'); \mathit{return}\ a\}$$

♢

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 12. 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.

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

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

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

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

Remark. 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 its complement contains 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 now introduce the general notion of monadic bx.

Definition. 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:

$$\begin{array}{l} \mathbf{data}\ BX\ \tau\ \alpha\ \beta = BX\ \{ get_L :: \tau\ \alpha, \quad set_L :: \alpha \to \tau\ (), \\ \qquad\qquad\qquad\qquad\qquad get_R :: \tau\ \beta, \quad set_R :: \beta \to \tau\ ()\} \end{array}$$

◇

3.1 Entangled state

We have seen that the *get* and *set* operations of the state monad satisfy the four laws (GG), (SG), (GS), (SS) of Definition 5. 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 5, 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. A *well-behaved BX* is one satisfying the following seven laws:

$$\begin{array}{ll}
(\mathrm{G_LG_L}) & \mathbf{do}\ \{a \leftarrow get_L; a' \leftarrow get_L; return\ (a, a')\} \\
& \qquad\qquad\qquad = \mathbf{do}\ \{a \leftarrow get_L; return\ (a, a)\} \\
(\mathrm{S_LG_L}) & \mathbf{do}\ \{set_L\ a; get_L\} \qquad = \mathbf{do}\ \{set_L\ a; return\ a\} \\
(\mathrm{G_LS_L}) & \mathbf{do}\ \{a \leftarrow get_L; set_L\ a\} = \mathbf{do}\ \{return\ ()\} \\
(\mathrm{G_RG_R}) & \mathbf{do}\ \{a \leftarrow get_R; a' \leftarrow get_R; return\ (a, a')\} \\
& \qquad\qquad\qquad = \mathbf{do}\ \{a \leftarrow get_R; return\ (a, a)\} \\
(\mathrm{S_RG_R}) & \mathbf{do}\ \{set_R\ a; get_R\} \qquad = \mathbf{do}\ \{set_R\ a; return\ a\} \\
(\mathrm{G_RS_R}) & \mathbf{do}\ \{a \leftarrow get_R; set_R\ a\} = \mathbf{do}\ \{return\ ()\} \\
(\mathrm{G_LG_R}) & \mathbf{do}\ \{a \leftarrow get_L; b \leftarrow get_R; return\ (a, b)\} \\
& \qquad\qquad\qquad = \mathbf{do}\ \{b \leftarrow get_R; a \leftarrow get_L; return\ (a, b)\}
\end{array}$$

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

$$\begin{array}{ll}
(\mathrm{S_LS_L}) & \mathbf{do}\ \{set_L\ a; set_L\ a'\} \quad = \mathbf{do}\ \{set_L\ a'\} \\
(\mathrm{S_RS_R}) & \mathbf{do}\ \{set_R\ a; set_R\ a'\} \quad = \mathbf{do}\ \{set_R\ a'\}
\end{array}$$

◇

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 $(\mathrm{G_LG_L})\ldots(\mathrm{G_LG_R})$ the *entangled state monad.*

Remark. 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. ♢

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

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

♢

The intuition is that state S 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 13. 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 14. 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 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 5. By Lemma 9, this gives a data refinement from T to $\mathit{StateT}\ S\ T$, and so we lose nothing by using $\mathit{StateT}\ S\ T$ instead of T. Despite this, we do not impose stability as a requirement in the following, because some interesting bx are not stable. $\diamondsuit$

We have not found convincing examples of $\mathit{StateTBX}$ in which the two get functions have effects from T, rather than being T-pure queries. In the latter case, we obtain the *get*/*get* commutation laws ($\mathrm{G_LG_L}$), ($\mathrm{G_RG_R}$), ($\mathrm{G_LG_R}$) for free [11], motivating the following:

Definition 15. We say that a well-behaved $\mathit{bx} :: \mathit{StateTBX}\ \ T\ \ S\ \ A\ \ B$ in the monad $\mathit{StateT}\ S\ T$ is *transparent* if get_L, get_R are T-pure queries, *i.e.* there exist $\mathit{read}_L :: S \to A$ and $\mathit{read}_R :: S \to B$ such that $\mathit{bx}.\mathit{get}_L = \mathit{gets}\ \mathit{read}_L$ and $\mathit{bx}.\mathit{get}_R = \mathit{gets}\ \mathit{read}_R$. $\diamondsuit$

Remark 16. 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 $\mathit{bx}.\mathit{read}_L$ and $\mathit{bx}.\mathit{read}_R$, regarding them as part of the signature of bx. The monomorphism condition holds for the various monads we consider in here (provided we have non-empty types σ for *State*, *Reader*, *Writer*). $\diamondsuit$

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

3.3 Monadic asymmetric lenses

To illustrate how BX generalises existing notions of (monadic) bx – and because it is a useful technical device when we define BX composition – consider:

Definition (monadic lens). A *monadic asymmetric lens* from source type A to view type B in which the update operation may have effects from monad T (or 'T-lens from A to B'), is represented by the type $\mathit{MLens}\ T\ A\ B$, where

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

We say that T-lens l is *well-behaved* if it satisfies

$$\begin{aligned}&(\mathrm{MVU})\quad \mathbf{do}\ \{l.\mathit{mupdate}\ s\ (l.\mathit{mview}\ s)\} = \mathbf{do}\ \{\mathit{return}\ s\}\\ &(\mathrm{MUV})\quad \mathbf{do}\ \{s' \leftarrow l.\mathit{mupdate}\ s\ v; \mathit{return}\ (s', l.\mathit{mview}\ s')\}\\ &\qquad\qquad = \mathbf{do}\ \{s' \leftarrow l.\mathit{mupdate}\ s\ v; \mathit{return}\ (s', v)\}\end{aligned}$$

and *very well-behaved* if in addition

$$\begin{aligned}&(\mathrm{MUU})\quad \mathbf{do}\ \{s' \leftarrow l.\mathit{mupdate}\ s\ v; l.\mathit{mupdate}\ s'\ v'\}\\ &\qquad\qquad = \mathbf{do}\ \{l.\mathit{mupdate}\ s\ v'\}\end{aligned}$$

$\diamondsuit$

An ordinary asymmetric lens as in Definition 10 is the special case *MLens Id*; the laws then specialise to the standard equational laws. Moreover, any asymmetric lens can be turned into a monadic lens that has no side-effects.

Definition. Here are a couple of useful examples:

$$
\begin{array}{l}
\mathit{fstMLens} :: \mathit{Monad}\ \tau \Rightarrow \mathit{MLens}\ \tau\ (\alpha, \beta)\ \alpha \\
\mathit{fstMLens} = \mathit{MLens}\ \mathit{mv}\ \mathit{mupd}\ \mathbf{where} \\
\quad \mathit{mv}\ (s_1, s_2) \qquad\quad = s_1 \\
\quad \mathit{mupd}\ (s_1, s_2)\ s_1' = \mathit{return}\ (s_1', s_2) \\
\mathit{sndMLens} :: \mathit{Monad}\ \tau \Rightarrow \mathit{MLens}\ \tau\ (\alpha, \beta)\ \beta \\
\mathit{sndMLens} = \mathit{MLens}\ \mathit{mv}\ \mathit{mupd}\ \mathbf{where} \\
\quad \mathit{mv}\ (s_1, s_2) \qquad\quad = s_2 \\
\quad \mathit{mupd}\ (s_1, s_2)\ s_2' = \mathit{return}\ (s_1, s_2')
\end{array}
$$

◇

Lemma 17. *fstMLens* and *sndMLens* are very well-behaved; moreover, their *mupdate* operations commute in T. ◇

Remark. Monadic generalisations of lenses have been considered by Pacheco *et al.* [28] and in online discussions [8]. The chief difference with the former [28] is that their laws appear to assume that the monad admits a membership operation $(\in) :: \alpha \to \tau\ \alpha \to \mathit{Bool}$; we make no such assumption, since it rules out important examples such as *State* and *IO*. In Diviánsky's monadic lens proposal [8], the *get* function has type $\alpha \to \tau\ \beta$, so in principle it too can have side-effects. As we shall see, this possibility significantly complicates composition. ◇

Remark. Symmetric lenses, as in Definition 12, are subsumed by our effectful bx too; in a nutshell, to simulate $\mathit{sl} :: \mathit{SLens}\ C\ A\ B$ 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 $\mathit{sl.put}_R\ (a, c) = (b, c)$ and $\mathit{sl.put}_L\ (b, c) = (a, c)$. But this turns out not to generalise straightforwardly to a corresponding notion of 'monadic symmetric lens' incorporating other effects as well. In the interests of brevity, we relegate to Appendix A the discussion as to why. ◇

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 states (but the same underlying monad of effects) into a common compound state.

Definition 18 (*StateT* embeddings from *T*-lenses). Given a T-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:

$$\begin{array}{l}
\vartheta :: \mathit{Monad}\ \tau \Rightarrow \mathit{MLens}\ \tau\ \alpha\ \beta \to \mathit{StateT}\ \beta\ \tau\ \gamma \to \mathit{StateT}\ \alpha\ \tau\ \gamma \\
\vartheta\ l\ m = \mathbf{do}\ a \leftarrow \mathit{get}; \mathbf{let}\ b = l.\mathit{mview}\ a; \\
\qquad\qquad\quad (c, b') \leftarrow \mathit{lift}\ (m\ b); \\
\qquad\qquad\quad a' \leftarrow \mathit{lift}\ (l.\mathit{mupdate}\ a\ b'); \\
\qquad\qquad\quad \mathit{set}\ a'; \mathit{return}\ c
\end{array}$$

♢

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 19. If $l :: \mathit{MLens}\ T\ A\ B$ is very well-behaved and $l.\mathit{mupdate}\ a\ b$ commutes in T for any a, b, then $\vartheta\ l$ is a monad morphism. ♢

Definition 20. By Lemmas 17 and 19, we have the following monad morphisms lifting stateful computations to a product state space:

$$\begin{array}{lcl}
\mathit{left} & :: & \mathit{Monad}\ \tau \Rightarrow \mathit{StateT}\ \sigma_1\ \tau\ \alpha \to \mathit{StateT}\ (\sigma_1, \sigma_2)\ \tau\ \alpha \\
\mathit{left} & = & \vartheta\ \mathit{fstMLens} \\
\mathit{right} & :: & \mathit{Monad}\ \tau \Rightarrow \mathit{StateT}\ \sigma_2\ \tau\ \alpha \to \mathit{StateT}\ (\sigma_1, \sigma_2)\ \tau\ \alpha \\
\mathit{right} & = & \vartheta\ \mathit{sndMLens}
\end{array}$$

♢

Definition. For $bx_1 :: \mathit{StateTBX}\ \ T\ \ S_1\ \ A\ \ B$, $bx_2 :: \mathit{StateTBX}\ \ T\ \ S_2\ \ B\ \ C$, define the join $S_1\ {}_{bx_1}\!\!\bowtie_{bx_2} S_2$ 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:

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

Note that the equation compares two computations of type $T\ B$; but if the bx are transparent, and *return* injective as per Remark 16, 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 13. ♢

Definition 21. Using *left* and *right*, we can define composition by:

$$\begin{array}{l}
(\fatsemi) :: \mathit{Monad}\ \tau \Rightarrow \\
\qquad \mathit{StateTBX}\ \sigma_1\ \tau\ \alpha\ \beta \to \mathit{StateTBX}\ \sigma_2\ \tau\ \beta\ \gamma \to \mathit{StateTBX}\ (\sigma_1 \bowtie \sigma_2)\ \tau\ \alpha\ \gamma \\
bx_1 \fatsemi bx_2 = \mathit{BX}\ \mathit{get}_L\ \mathit{set}_L\ \mathit{get}_R\ \mathit{set}_R\ \mathbf{where} \\
\quad \begin{array}{lcl}
\mathit{get}_L & = & \mathbf{do}\ \{ \mathit{left}\ (bx_1.\mathit{get}_L) \} \\
\mathit{get}_R & = & \mathbf{do}\ \{ \mathit{right}\ (bx_2.\mathit{get}_R) \} \\
\mathit{set}_L\ a & = & \mathbf{do}\ \{ \mathit{left}\ (bx_1.\mathit{set}_L\ a); b \leftarrow \mathit{left}\ (bx_1.\mathit{get}_R); \mathit{right}\ (bx_2.\mathit{set}_L\ b) \} \\
\mathit{set}_R\ c & = & \mathbf{do}\ \{ \mathit{right}\ (bx_2.\mathit{set}_R\ c); b \leftarrow \mathit{right}\ (bx_2.\mathit{get}_L); \mathit{left}\ (bx_1.\mathit{set}_R\ b) \}
\end{array}
\end{array}$$

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 operates on the compound state $\sigma_1 \bowtie \sigma_2$, not on arbitrary pairs $\sigma_1 \times \sigma_2$. ♢

Theorem 22 (transparent composition). Given transparent well-behaved $bx_1 :: \mathit{StateTBX}\ S_1\ T\ A\ B$ and $bx_2 :: \mathit{StateTBX}\ S_2\ T\ B\ C$, their composition $bx_1 \fatsemi bx_2 :: \mathit{StateTBX}\ (S_1 \bowtie S_2)\ T\ A\ C$ is transparent and well-behaved. $\Diamond$

Remark. Unpacking and simplifying the definitions, we have:

$$
\begin{array}{lll}
\multicolumn{3}{l}{bx_1 \fatsemi bx_2 = \mathit{BX}\ \mathit{get}_L\ \mathit{set}_L\ \mathit{get}_R\ \mathit{set}_R\ \mathbf{where}} \\
\quad \mathit{get}_L & = \mathbf{do}\ \{ & (s_1, \_) \leftarrow \mathit{get}; \mathit{return}\ (bx_1.\mathit{read}_L\ s_1)\} \\
\quad \mathit{get}_R & = \mathbf{do}\ \{ & (\_, s_2) \leftarrow \mathit{get}; \mathit{return}\ (bx_2.\mathit{read}_R\ s_2)\} \\
\quad \mathit{set}_L\ a' & = \mathbf{do}\ \{ & (s_1, s_2) \leftarrow \mathit{get}; \\
 & & ((), s_1') \leftarrow \mathit{lift}\ (bx_1.\mathit{set}_L\ a'\ s_1); \\
 & & \mathbf{let}\ b = bx_1.\mathit{read}_R\ s_1'; \\
 & & ((), s_2') \leftarrow \mathit{lift}\ (bx_2.\mathit{set}_L\ b\ s_2); \\
 & & \mathit{set}\ (s_1', s_2')\} \\
\quad \mathit{set}_R\ c' & = \mathbf{do}\ \{ & (s_1, s_2) \leftarrow \mathit{get}; \\
 & & ((), s_2') \leftarrow \mathit{lift}\ (bx_2.\mathit{set}_R\ c'\ s_2); \\
 & & \mathbf{let}\ b = bx_2.\mathit{read}_L\ s_2'; \\
 & & ((), s_1') \leftarrow \mathit{lift}\ (bx_1.\mathit{set}_R\ b\ s_1); \\
 & & \mathit{set}\ (s_1', s_2')\}
\end{array}
$$

$\Diamond$

Remark 23. 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 31 in Section 6. This is why we mostly limit attention to transparent bx. $\Diamond$

5 Equivalence

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

Definition 24 (identity). For any underlying monad instance, we can form the *identity* bx as follows:

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

Clearly, this bx is well-behaved, overwritable and transparent. $\Diamond$

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. Apparently, therefore, as for symmetric lenses [13], we must satisfy ourselves with equality up to some notion of *equivalence* of bx.

Definition. 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. $\diamondsuit$

Definition. In the case of *StateTBX*s, with $T_1 = StateT\ S_1\ T$ and $T_2 = StateT\ S_2\ T$ for some state types S_1, S_2, we can construct a monad isomorphism from T_1 to T_2 by lifting an isomorphism on the state spaces, using the following construction:

```
data Iso α β = Iso { to :: α → β, from :: β → α }
inv h = Iso (h.from) (h.to)
ι :: Monad τ ⇒ Iso σ₁ σ₂ → StateT σ₁ τ α → StateT σ₂ τ α
ι h m = do { s₂ ← get; (a, s₁) ← lift (m (h.from s₂));
             set (h.to s₁); return a }
ι⁻¹ h = ι (inv h)
```

$\diamondsuit$

Lemma 25. If $h :: S_1 \to S_2$ is invertible, then $\iota\ h$ is a monad isomorphism from $StateT\ S_1\ T$ to $StateT\ S_2\ T$. $\diamondsuit$

To show equivalence of $bx_1 :: StateTBX\ T\ S_1\ A\ B$ and $bx_2 :: StateTBX\ T\ S_2\ A\ B$, we just need to find an invertible function $h :: S_1 \to S_2$ such that the induced monad isomorphism $\iota\ h :: StateT\ S_1\ T \to StateT\ S_2\ T$ satisfies $\iota\ (bx_1.get_L) = bx_2.get_L$ and $\iota\ (bx_1.set_L\ a) = bx_2.set_L\ a$ and dually.

Theorem 26. Composition of transparent bx satisfies the identity and associativity laws, modulo $\equiv$.

(Identity) $\quad identity \fatsemi bx \equiv bx \equiv bx \fatsemi identity$
(Assoc) $\quad bx_1 \fatsemi (bx_2 \fatsemi bx_3) \equiv (bx_1 \fatsemi bx_2) \fatsemi bx_3$ $\diamondsuit$

6 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.

6.1 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]. As we shall see in Section 6.2, initialisation is also crucial for certain combinators.

Definition. 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:

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

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

We can extend composition to handle initialisation as follows:

$$\begin{array}{l}
(\mathit{bx}_1 \fatsemi \mathit{bx}_2).\mathit{init}_L\ a = \mathbf{do}\ \{s_1 \leftarrow \mathit{bx}_1.\mathit{init}_L\ a; b \leftarrow \mathit{bx}_1.\mathit{get}_R\ s_1; \\
\qquad\qquad\qquad\qquad\qquad s_2 \leftarrow \mathit{bx}_2.\mathit{init}_L\ b; \mathit{return}\ (s_1, s_2)\}
\end{array}$$

We refine the notions of bx isomorphism and equivalence to *InitStateTBX* as follows. As noted earlier, a bijection $h :: S_1 \to S_2$ induces a monad morphism $\iota\ h :: \mathit{StateT}\ S_1\ T \to \mathit{StateT}\ S_2\ T$. An isomorphism of *InitStateTBX*s consists of an h such that the following equations (and their duals) hold:

$$\begin{array}{ll}
(\iota\ h)\ (\mathit{bx}_1.\mathit{get}_L) & = \mathit{bx}_2.\mathit{get}_L \\
(\iota\ h)\ (\mathit{bx}_1.\mathit{set}_L\ a) & = \mathit{bx}_2.\mathit{set}_L\ a \\
\mathbf{do}\ \{s \leftarrow \mathit{bx}_1.\mathit{init}_L\ a; \mathit{return}\ (h\ s)\} & = \mathit{bx}_2.\mathit{init}_L\ a
\end{array}$$

Note that the first two equations (and their duals) imply that $\iota\ h$ 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. 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:

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

However, *initBoth* and $\mathit{init}_L, \mathit{init}_R$ 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*. ◇

6.2 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.

Definition 27 (duality). Trivially, we can dualise any bx:

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

which simply exchanges the left and right operations; this preserves well-behaved-ness, transparency, and overwritability of the underlying bx. $\Diamond$

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

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

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

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

Pairing is defined as follows:

$$\begin{array}{l}
\mathit{pairBX} :: \mathit{Monad}\ \tau \Rightarrow \mathit{StateTBX}\ \tau\ \sigma_1\ \alpha_1\ \beta_1 \to \mathit{StateTBX}\ \tau\ \sigma_2\ \alpha_2\ \beta_2 \to \\
\qquad\qquad \mathit{StateTBX}\ \tau\ (\sigma_1,\sigma_2)\ (\alpha_1,\alpha_2)\ (\beta_1,\beta_2) \\
\mathit{pairBX}\ \mathit{bx}_1\ \mathit{bx}_2 = \mathit{BX}\ \mathit{gl}\ \mathit{sl}\ \mathit{gr}\ \mathit{sr}\ \mathbf{where} \\
\quad \mathit{gl} = \mathbf{do}\ \{ a_1 \leftarrow \mathit{left}\ (\mathit{bx}_1.\mathit{get}_L); a_2 \leftarrow \mathit{right}\ (\mathit{bx}_2.\mathit{get}_L); \mathit{return}\ (a_1, a_2) \} \\
\quad \mathit{sl}\ (a_1, a_2) = \mathbf{do}\ \{ \mathit{left}\ (\mathit{bx}_1.\mathit{set}_L\ a_1); \mathit{right}\ (\mathit{bx}_2.\mathit{set}_L\ a_2) \} \\
\quad \mathit{gr} \qquad\quad = ... \ \ \text{-- dual} \\
\quad \mathit{sr} \qquad\quad = ... \ \ \text{-- dual}
\end{array}$$

$\Diamond$

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 well-behavedness; this extends further to transparency:

Proposition 28. If bx_1 and bx_2 are transparent, then $\mathit{pairBX}\ bx_1\ bx_2$ is transparent. ♢

Remark. 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 (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.

$$\begin{array}{l}
\mathit{inlBX} :: \mathit{Monad}\ \tau \Rightarrow \alpha \rightarrow \mathit{StateTBX}\ \tau\ (\alpha, \mathit{Maybe}\ \beta)\ \alpha\ (\mathit{Either}\ \alpha\ \beta)\\
\mathit{inrBX} :: \mathit{Monad}\ \tau \Rightarrow \beta \rightarrow \mathit{StateTBX}\ \tau\ (\beta, \mathit{Maybe}\ \alpha)\ \beta\ (\mathit{Either}\ \alpha\ \beta)
\end{array}$$

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.

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

♢

Proposition 29. If bx_1 and bx_2 are transparent then $\mathit{sumBX}\ bx_1\ bx_2$ is transparent. ♢

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 (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).

```
listIBX :: Monad τ ⇒
           InitStateTBX τ σ α β → InitStateTBX τ (Int, [σ]) [α] [β]
listIBX bx = InitStateTBX gl sl il gr sr ir where
  gl    = do { (n, cs) ← get; mapM (lift · eval bx.get_L) (take n cs) }
  sl as = do { (_, cs) ← get;
               cs′ ← lift (sets (exec · bx.set_L) bx.init_L as cs);
               set (length as, cs′) }
  il as = do { cs ← mapM (bx.init_L) as; return (length as, cs) }
  gr    = ...  -- dual
  sr bs = ...  -- dual
  ir bs = ...  -- dual
```

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:

```
sets :: Monad τ ⇒ (α → γ → τ γ) → (α → τ γ) → [α] → [γ] → τ [γ]
sets set init []       cs       = return cs
sets set init (x : xs) (c : cs) = do { c′ ← set x c; cs′ ← sets set init xs cs;
                                       return (c′ : cs′) }
sets set init xs       []       = mapM init xs
```

◇

Proposition 30. If *bx* is transparent then *listBX bx* is transparent. ◇

6.3 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. Interestingly, the first of our examples is well-behaved but not transparent. In the interests of brevity, we omit dual cases and initialisation functions, but these are defined in the Appendix.

Example 31 (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 [36] 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* γ. Then we can define

```
switch :: (γ → StateTBX (Reader γ) σ α β) → StateTBX (Reader γ) σ α β
switch f = BX gl sl gr sr where
  gl   = do { c ← lift ask; (f c).getL }
  sl a = do { c ← lift ask; (f c).setL a }
  gr   = ...  -- dual
  sr   = ...  -- dual
```

where the standard *ask* :: *Reader* γ operation reads the γ value. ♢

Proposition 32. If $f\ c$ :: *StateTBX* (*Reader* C) $S\ A\ B$ is transparent for any $c :: C$, then *switch* f is a well-behaved, but not necessarily transparent, *StateTBX* (*Reader* C) $S\ A\ B$. ♢

Remark. The reason why *switch* f is not (necessarily) transparent is that 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* γ)-pure. Moreover, it is not difficult to use this example 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. ♢

Example (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 *Float* 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*.

```
invBX :: StateTBX Maybe Float Float Float
invBX = BX get setL (gets (λa. 1/a)) setR where
  setL a = do { lift (guard (a ≠ 0)); set a }
  setR b = do { lift (guard (b ≠ 0)); set (1/b) }
```

where *guard* b = **do** {**if** b **then** *Just* () **else** *Nothing*} is a standard operation in the *Maybe* monad. As another example, suppose we know 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:

```
readSomeBX :: (Read α, Show α) ⇒ StateTBX Maybe (α, String) α String
readSomeBX = BX (gets fst) setL (gets snd) setR where
  setL a′ = set (a′, show a′)
  setR b′ = do { (_, b) ← get;
                 if b == b′ then return () else case reads b of
                   ((a′, ""): _) → set (a′, b)
                   _             → lift Nothing }
```

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 $err :: \forall\alpha.\, T\ \alpha$ and any pair of partial inverse functions $f :: A \to Maybe\ B$ and $g :: B \to Maybe\ A$ (*i.e.*, $f\ a = Just\ b$ if and only if $g\ b = Just\ a$, for any a, b):

```
partialBX :: Monad τ ⇒ (∀α.τ α) → (α → Maybe β) → (β → Maybe α) →
                 StateTBX τ (α, β) α β
partialBX err f g = BX (gets fst) set_L (gets snd) set_R where
  set_L a′ = case f a′ of Just b′   → set (a′, b′)
                          Nothing → lift err
  set_R b′ = case g b′ of Just a′   → set (a′, b′)
                          Nothing → lift err
```

Then we could define *invBX* and *readSomeBX* as instances of *partialBX*. ♢

Proposition 33. Let $f :: A \to Maybe\ B$ and $g :: B \to Maybe\ A$ be partial inverses and let err be a zero element for T. Then $partialBX\ err\ f\ g :: StateTBX\ T\ S\ A\ B$ is transparent, where $S = \{(a, b) \mid f\ a = Just\ b \wedge g\ b = Just\ a\}$. ♢

Example (nondeterminism—Scenario 1 revisited). For simplicity, 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 all the B values consistent with a given A, and symmetrically a fix-up function *as*.

```
nondetBX :: (α → β → Bool) → (α → [β]) → (β → [α]) →
                 StateTBX [] (α, β) α β
nondetBX ok bs as = BX (gets fst) set_L (gets snd) set_R where
  set_L a′ = do {(a, b) ← get;
                 if ok a′ b then set (a′, b) else
                   do {b′ ← lift (bs a′); set (a′, b′)}}
  set_R b′ = do {(a, b) ← get;
                 if ok a b′ then set (a, b′) else
                   do {a′ ← lift (as b′); set (a′, b′)}}
```

♢

Proposition 34. Given *ok*, $S = \{(a, b) \mid ok\ a\ b\}$, and *as* and *bs* satisfying

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

then the nondeterministic bx $nondetBX\ ok\ bs\ as :: StateTBX\ [\,]\ S\ A\ B$ is transparent. ♢

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

$$\begin{array}{l}
\mathit{signalBX} :: (\mathit{Eq}\ \alpha, \mathit{Eq}\ \beta, \mathit{Monad}\ \tau) \Rightarrow (\alpha \to \tau\ ()) \to (\beta \to \tau\ ()) \to \\
\qquad\qquad\qquad \mathit{StateTBX}\ \tau\ \sigma\ \alpha\ \beta \to \mathit{StateTBX}\ \tau\ \sigma\ \alpha\ \beta \\
\mathit{signalBX}\ \mathit{sigA}\ \mathit{sigB}\ \mathit{bx} = \mathit{BX}\ (\mathit{bx}.\mathit{get}_L)\ \mathit{sl}\ (\mathit{bx}.\mathit{get}_R)\ \mathit{sr}\ \mathbf{where} \\
\quad \mathit{sl}\ a' = \mathbf{do}\ \{\, a \leftarrow \mathit{bx}.\mathit{get}_L; \mathit{bx}.\mathit{set}_L\ a'; \\
\qquad\qquad\qquad\quad \mathit{lift}\ (\mathbf{if}\ a \neq a'\ \mathbf{then}\ \mathit{sigA}\ a'\ \mathbf{else}\ \mathit{return}\ ())\} \\
\quad \mathit{sr}\ b' = \mathbf{do}\ \{\, b \leftarrow \mathit{bx}.\mathit{get}_R; \mathit{bx}.\mathit{set}_R\ b'; \\
\qquad\qquad\qquad\quad \mathit{lift}\ (\mathbf{if}\ b \neq b'\ \mathbf{then}\ \mathit{sigB}\ b'\ \mathbf{else}\ \mathit{return}\ ())\}
\end{array}$$

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 performs $\mathit{sigA}\ a'$. If the bx is to be well-behaved, then no action can be performed in the case that $a == a'$.

For example, instantiating τ to IO we have:

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

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:

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

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. ♢

Proposition 35. If A and B are types equipped with a correct notion of equality (that is, $(a == b) = \mathit{True}$ 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 (interaction—Scenario 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.

```
dynamicBX :: (Eq α, Eq β, Monad τ) ⇒
             (α → β → τ β) → (α → β → τ α) →
             StateTBX τ ((α, β), [((α, β), β)], [((α, β), α)]) α β
dynamicBX f g = BX (gets (fst · fst3)) setL (gets (snd · fst3)) setR where
  setL a′ = do {((a, b), fs, bs) ← get;
                if a == a′ then return () else
                  case lookup (a′, b) fs of
                    Just b′  → set ((a′, b′), ((a′, b), b′) : fs, bs)
                    Nothing → do { b′ ← lift (f a′ b);
                                   set ((a′, b′), ((a′, b), b′) : fs, bs)}}
  setR b′ = ...   -- dual
```

where *fst3* $(a, b, c) = a$. For example, the bx below finds matching states by asking the user, writing to (*putStr*, *putStrLn*) and reading from (*getLine*) the terminal.

```
dynamicIOBX :: (Eq α, Eq β, Show α, Show β, Read α, Read β) ⇒
               StateTBX IO ((α, β), [((α, β), β)], [((α, β), α)]) α β
dynamicIOBX = dynamicBX matchIO (flip matchIO)

matchIO :: (Show α, Show β, Read β) ⇒ α → β → IO β
matchIO a b = do {putStrLn ("Setting " ++ show a);
                  putStr ("Replacement for " ++ show b ++ "?");
                  s ← getLine; return (read s)}
```

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

```
dynamicSearchBX ::
  (Eq α, Eq β, Enum α, Bounded α, Enum β, Bounded β) ⇒
  (α → β → Bool) →
  StateTBX Maybe ((α, β), [((α, β), β)], [((α, β), α)]) α β
dynamicSearchBX p = dynamicBX (search p) (flip (search (flip p)))

search :: (Enum β, Bounded β) ⇒ (α → β → Bool) → α → β → Maybe β
search p a _ = find (p a) [minBound..maxBound]
```

◇

Proposition 36. For any f, g, the dynamic bx *dynamicBX f g* is transparent. ◇

Proof (Sketch). Let *bx* = *dynamicBX f g* for some f, g. For $(\mathrm{S_L G_L})$, by construction, a call to *bx.set*$_L$ a' ends by setting the state to $((a', b'), fs, bs)$ for some b', fs, bs, and a subsequent *bx.get*$_L$ will return a'. For $(\mathrm{G_L S_L})$, a call to *bx.get*$_L$ in a state $((a, b), fs, bs)$ returns a, and by construction a subsequent *bx.set*$_L$ a has no effect. □

7 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*.

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 [1]. 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 appears to generalise 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 [2,19,30]. 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 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.

8 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 a wide-ranging 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 and to carefully consider 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 than the equivalence for symmetric lenses presented by Hofmann *et al.* [13]. Early investigations, guided by an alternative coalgebraic presentation [1] 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 [3] 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 [1] was made at CMCS 2014; but none of these presentations were accompanied by a full paper. We thank the organisers of and participants at those meetings and earlier anonymous reviewers for their helpful comments. The work was supported by the UK EPSRC-funded project *A Theory of Least Change for Bidirectional Transformations* [34] (EP/K020218/1, EP/K020919/1).

References

1. Faris Abou-Saleh and James McKinna. A coalgebraic approach to bidirectional transformations. Short presentation at CMCS, 2014.
2. Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *ICFP*, pp133–144. ACM, 2013.
3. James Cheney, James McKinna, Perdita Stevens, Jeremy Gibbons, and Faris Abou-Saleh. Entangled state monads (abstract). In [33].
4. Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. JTL: A bidirectional and change propagating transformation language. In *SLE 2010*, *LNCS 6563*, pp183–202. Springer, 2010.
5. 'Composers' example. `http://bx-community.wikidot.com/examples:composers`.
6. Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT*, *LNCS 5563*, pp260–283. Springer, 2009.
7. Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. From state- to delta-based bidirectional model transformations: the asymmetric case. *JOT*, 10:6: 1–25, 2011.
8. Péter Diviánszky. LGtk API correction. `http://people.inf.elte.hu/divip/LGtk/CorrectedAPI.html`, April 2013.
9. J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view update problem. In *popl*, pp233–246. ACM, 2005.
10. J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM TOPLAS*, 29(3):17, 2007. Extended version of [9].
11. Jeremy Gibbons and Ralf Hinze. Just **do** it: Simple monadic equational reasoning. In *ICFP*, pp2–14. ACM, 2011.
12. Stephen J. Hegner. An order-based theory of updates for closed database views. *Ann. Math. Art. Int.*, 40:63–125, 2004.
13. Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Symmetric lenses. In *POPL*, pp371–384. ACM, 2011.
14. Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Edit lenses. In *POPL*, pp495–508. ACM, 2012.
15. Martin Hyland, Gordon D. Plotkin, and John Power. Combining effects: Sum and tensor. *TCS*, 357(1-3):70–99, 2006.
16. Michael Johnson and Robert Rosebrugh. Spans of lenses. In [33].
17. Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report RR-1004, DCS, Yale, 1993.
18. Koji Kagawa. Compositional references for stateful functional programming. In *ICFP*, pp217–226, 1997.
19. Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *ICFP*, pp145–158. ACM, 2013.
20. Edward Kmett. `Data.Lens` library. `http://hackage.haskell.org/package/lenses-0.1.2/docs/Data-Lenses.html`.
21. Edward Kmett. `lens-4.0.4` library. `http://hackage.haskell.org/package/lens`.
22. Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In *POPL*, pp333–343, 1995.
23. Christoph Lüth and Neil Ghani. Composing monads using coproducts. In *ICFP*, pp133–144. ACM, 2002.
24. Nuno Macedo, Alcinho Cunha, and Hugo Pacheco. Toward a framework for multidirectional model transformations. In [33].
25. Eugenio Moggi. Notions of computation and monads. *Inf.&Comp.*, 93(1):55–92, 1991.
26. Till Mossakowski, Lutz Schröder, and Sergey Goncharov. 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. Hugo Pacheco, Zhenjiang Hu, and Sebastian Fischer. Monadic combinators for "putback" style bidirectional programming. In *PEPM*, pp39–50. ACM, 2014.
29. Gordon D. Plotkin and John Power. Notions of computation determine monads. In *FOSSACS*, *LNCS 2303*, pp342–356. Springer, 2002.
30. Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *LMCS*, 9(4), 2013.
31. Olha Shkaravska. Side-effect monad, its equational theory and applications. Seminar slides available at: `http://www.ioc.ee/~tarmo/tsem05/shkaravska1512-slides.pdf`, 2005.
32. Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. *SoSyM*, 9(1):7–20, 2010.

33. James Terwilliger and Soichiro Hidaka, editors. *BX Workshop.* `http://ceur-ws.org/Vol-1133/#bx`, 2014.
34. 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.
35. Dániel Varró. Model transformation by example. In *MoDELS 2006*, *LNCS 4199*, pp410–424. Springer, 2006.
36. Janis Voigtländer, Zhenjiang Hu, Kazutaka Matsuda, and Meng Wang. Enhancing semantic bidirectionalization via shape bidirectionalizer plug-ins. *JFP*, 23(5):515–551, 2013.

Appendices

We present a detailed comparison with symmetric lenses [13], proofs of the lemmas and theorems omitted from the body of the paper, and expand the code that was abbreviated in the paper.

A Comparison with symmetric lenses

Hofmann *et al.* [13] (henceforth: HPW) proposed *symmetric lenses* that use a *complement* to store (at least) the information that is not present in both views. There are several natural questions concerning the relationship between symmetric lenses and our effectful bx. In particular, could we not just add monads to symmetric lenses, to allow effects? Can we convert such 'monadic' symmetric lenses to effectful bx, and back? How does the 'missing' element used for initialisation in symmetric lenses relate to the $init_L$ and $init_R$ operations we employ? How does our approach to equivalence (based on monad isomorphisms) compare with HPW's approach for symmetric lenses? We consider these questions in turn. But before we do, we note that Definition 12 omitted the 'missing' value required in order to initialise a symmetric lens; a more faithful rendering would be:

$$\begin{aligned}\mathbf{data}\ \mathit{SLens}\ \gamma\ \alpha\ \beta = \mathit{SLens}\ \{&\mathit{put}_R \ \ :: (\alpha,\gamma) \to (\beta,\gamma),\\ &\mathit{put}_L \ \ :: (\beta,\gamma) \to (\alpha,\gamma),\\ &\mathit{missing} :: \gamma\}\end{aligned}$$

Since symmetric lenses involve no effects beyond bidirectionality, they are related to $\mathit{StateTBX}$s over the identity monad:

$$\mathbf{type}\ \mathit{StateBX}\ \sigma\ \alpha\ \beta = \mathit{StateTBX}\ \mathit{Id}\ \sigma\ \alpha\ \beta$$

A.1 Symmetric lenses and *StateBX*s

The differences between the complement in a symmetric lens and the state in a *StateBX* are illustrated by the following example.

Example 37. Consider the following variant of the Composers example [5] in which a set of triples (*Name*, *Nationality*, *Dates*) is kept consistent with a list of (*Name*, *Nationality*) pairs, the consistency condition being that the same (*Name*, *Nationality*) pairs occur in each view; we assume that the owner of the left view cares about *Dates*, the owner of the right view cares about order, and both are committed to *Name*s as keys. This situation is illustrated in Figure 1.

If we implement this as a symmetric lens, a natural complement – not the only choice – is a list of (*Name*, *Dates*) pairs. One way to restore consistency is to let put_R, being given a new view m which is a set of (*Name*, *Nationality*, *Dates*) triples and an old complement c which is a list of (*Name*, *Dates*) pairs, construct a list of triples consisting of those triples

A

$\{$ (Schumann, Germany, 1810–1856),
(Schubert, Austria, 1797–1828),
$\ldots\}$

B

Name	*Nationality*
Schubert	Austria
Schumann	Germany
...	...

C

Name	*Dates*
Schubert	1797–1828
Schumann	1810–1856
...	...

S

Name	*Nationality*	*Dates*
Schubert	Austria	1797–1828
Schumann	Germany	1810–1856
...	...	...

Fig. 1. Illustration of Example 37. A and B are the states of the left and right sides, respectively. C is the complement of the symmetric lens, and S is the state of the bx.

from m, ordered as they were in c, with triples whose *Name*s did not appear in c placed at the end of the list in *Name* order, and no triples corresponding to *Name*s that occurred in c but not m. From this list of triples the new complement c' is extracted by projecting away *Nationality*, and the new view n is obtained by projecting away *Dates*.

Rephrasing this as a *StateBX* between the same view types, the natural state is a list of $(\mathit{Name}, \mathit{Nationality}, \mathit{Dates})$ triples. The get_L function forgets the order and the get_R function discards the *Dates*. The set_L function updates the state by changing *Nationality* and *Dates* entries as required, adding new triples at the end in *Name* order, and deleting any triples whose *Name*s no longer occur. ♢

A.2 Naive symmetric monadic lenses

We now consider an obvious monadic generalisation of symmetric lenses, in which the put_L and put_R functions are allowed to have effects in some monad T:

Definition 38. A *monadic symmetric lens* from A to B with complement type C and effects T consists of two functions converting A to B and vice versa, each also operating on C and possibly having effects in T, and a complement value *missing* used for initialisation:

$$\begin{aligned}
\mathbf{data}\ \mathit{SMLens}\ \tau\ \gamma\ \alpha\ \beta = \mathit{SMLens}\ \{ & \mathit{mput}_R :: (\alpha,\gamma) \to \tau\ (\beta,\gamma),\\
& \mathit{mput}_L :: (\beta,\gamma) \to \tau\ (\alpha,\gamma),\\
& \mathit{missing} :: \gamma \}
\end{aligned}$$

Such a lens *sl* is called *well-behaved* if:

$$\begin{aligned}
&(\text{PutRLM}) && \mathbf{do}\ \{(b,c') \leftarrow sl.\mathit{mput}_R\ (a,c); sl.\mathit{mput}_L\ (b,c')\}\\
& && \quad = \mathbf{do}\ \{(b,c') \leftarrow sl.\mathit{mput}_R\ (a,c); \mathit{return}\ (a,c')\}\\
&(\text{PutLRM}) && \mathbf{do}\ \{(a,c') \leftarrow sl.\mathit{mput}_L\ (b,c); sl.\mathit{mput}_R\ (a,c')\}\\
& && \quad = \mathbf{do}\ \{(a,c') \leftarrow sl.\mathit{mput}_L\ (b,c); \mathit{return}\ (b,c')\}
\end{aligned}$$

♢

We can recover HPW's symmetric lenses by taking $T = \mathit{Id}$. In that case, the above laws have the following form:

$$\begin{array}{ll}
\text{(PutRLM)} & \mathbf{let}\ (b, c') = \mathit{sl.mput}_R\ (a, c)\ \mathbf{in}\ \mathit{sl.mput}_L\ (b, c') \\
& \quad = \mathbf{let}\ (b, c') = \mathit{sl.mput}_R\ (a, c)\ \mathbf{in}\ (a, c') \\
\text{(PutLRM)} & \mathbf{let}\ (a, c') = \mathit{sl.mput}_L\ (b, c)\ \mathbf{in}\ \mathit{sl.mput}_R\ (a, c') \\
& \quad = \mathbf{let}\ (a, c') = \mathit{sl.mput}_L\ (b, c)\ \mathbf{in}\ (b, c')
\end{array}$$

It is an easy exercise to show that these two equational laws are (essentially) equivalent to the conditional equations (PutRL) and (PutLR), so *SLens*es correspond to *SMLens*es where $T = \mathit{Id}$.

The above monadic generalisation of symmetric lenses appears natural, but it turns out to have nontrivial limitations compared to our definition of effectful bx. Specifically:

- Although there is a natural notion of composition for naive monadic symmetric lenses, it does not in general preserve well-behavedness for a non-commutative monad of effects.
- There is also a natural mapping from naive monadic symmetric lenses to effectful bx, which preserves well-behavedness in the pure case, but fails even in the presence of simple monadic effects such as *Maybe*.
- Finally, there is a natural reverse mapping from effectful bx back to naive monadic symmetric lenses, which preserves well-behavedness for any kind of effects.

The first observation clearly indicates that our approach is not merely a disguised form of the obvious approach to extending symmetric lenses with effects: the naive approach is closed under composition only in the presence of commutative effects, while our approach is closed under composition in the presence of arbitrary effects. The second observation shows that in the absence of effects, our notion of bx is equivalent to pure symmetric lenses. Finally, the second and third observations suggest that we can view our effectful bx as a subcategory of monadic symmetric lenses with particularly good behaviour. It is an open question whether we can characterise this subcategory using laws based solely on the symmetric lens operations.

In the rest of this section we explain the above three observations in greater detail.

Composition and well-behavedness Consider the following candidate definition of composition for *SMLens*:

$$\begin{array}{l}
(;) :: \mathit{Monad}\ \tau \Rightarrow \mathit{SMLens}\ \tau\ \sigma_1\ \alpha\ \beta \to \mathit{SMLens}\ \tau\ \sigma_2\ \beta\ \gamma \to \mathit{SMLens}\ \tau\ (\sigma_1, \sigma_2)\ \alpha\ \gamma \\
\mathit{sl}_1\ ;\ \mathit{sl}_2 = \mathit{SMLens}\ \mathit{put}_R\ \mathit{put}_L\ \mathit{missing}\ \mathbf{where} \\
\quad \mathit{put}_R\ (a, (s_1, s_2)) = \mathbf{do}\ \{(b, s_1') \leftarrow \mathit{sl}_1.\mathit{mput}_R\ (a, s_1); \\
\qquad\qquad\qquad\qquad\qquad\quad (c, s_2') \leftarrow \mathit{sl}_2.\mathit{mput}_R\ (b, s_2); \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (c, (s_1', s_2'))\} \\
\quad \mathit{put}_L = \ldots \quad \text{-- dual} \\
\quad \mathit{missing} = (\mathit{sl}_1.\mathit{missing}, \mathit{sl}_2.\mathit{missing})
\end{array}$$

which seems to be the obvious generalisation of pure symmetric lens composition to the monadic case. However, it does not always preserve well-behavedness:

Example 39. Consider the following construction:

$$\begin{array}{l}
\mathit{setBool} :: \mathit{Bool} \to \mathit{SMLens}\ (\mathit{State}\ \mathit{Bool})\ ()\ ()\ () \\
\mathit{setBool}\ b = \mathit{SMLens}\ m\ m\ ()\ \mathbf{where}\ m\ \_ = \mathbf{do}\ \{\mathit{set}\ b; \mathit{return}\ ((),())\}
\end{array}$$

The lens *setBool True* has no effect on the complement or values (indeed, there is no other choice), but sets the state value to *True*. Both *setBool True* and *setBool False* are well-behaved, but their composition (in either direction) is not well-behaved. Essentially, the reason is that *setBool True* and *setBool False* share a single *Bool* state value. ♢

Proposition 40. *setBool* b is well-behaved for $b \in \{\mathit{True}, \mathit{False}\}$. ♢

Proof. Let $sl = \mathit{setBool}\ x$. We consider (PutRLM), and (PutLRM) is symmetric.

$$\begin{array}{ll}
& \mathbf{do}\ \{(b,c') \leftarrow (\mathit{setBool}\ x).\mathit{mput}_R\ ((),()); (\mathit{setBool}\ x).\mathit{mput}_L\ (b,c')\} \\
= & [\![\text{ Definition }]\!] \\
& \mathbf{do}\ \{(b,c') \leftarrow \mathbf{do}\ \{\mathit{set}\ x; \mathit{return}\ ((),())\}; \mathit{set}\ x; \mathit{return}\ ((),c')\} \\
= & [\![\text{ monad associativity }]\!] \\
& \mathbf{do}\ \{\mathit{set}\ x; (b,c') \leftarrow \mathit{return}\ ((),()); \mathit{set}\ x; \mathit{return}\ ((),c')\} \\
= & [\![\text{ commutativity of } \mathit{return}]\!] \\
& \mathbf{do}\ \{\mathit{set}\ x; \mathit{set}\ x; (b,c') \leftarrow \mathit{return}\ ((),()); \mathit{return}\ ((),c')\} \\
= & [\![\text{ (SS) }]\!] \\
& \mathbf{do}\ \{\mathit{set}\ x; (b,c') \leftarrow \mathit{return}\ ((),()); \mathit{return}\ ((),c')\} \\
= & [\![\text{ monad associativity }]\!] \\
& \mathbf{do}\ \{(b,c') \leftarrow \mathbf{do}\ \{\mathit{set}\ x; \mathit{return}\ ((),())\}; \mathit{return}\ ((),c')\} \\
= & [\![\text{ Definition }]\!] \\
& \mathbf{do}\ \{(b,c') \leftarrow (\mathit{setBool}\ x).\mathit{mput}_R\ ((),()); \mathit{return}\ ((),c')\}
\end{array}$$

□

Proposition 41. *setBool True* ; *setBool False* is not well-behaved. ♢

Proof. Taking $sl = \mathit{setBool}\ \mathit{True}\ ;\ \mathit{setBool}\ \mathit{False}$, we proceed as follows:

$$\begin{array}{ll}
& \mathbf{do}\ \{(c,s') \leftarrow sl.\mathit{mput}_R\ (a,s); sl.\mathit{mput}_L\ (c,s')\} \\
= & [\![\text{ let } s = (s_1,s_2) \text{ and } s' = (s_1''',s_2'''); \text{ definition }]\!] \\
& \mathbf{do}\ \{(b,s_1') \leftarrow (\mathit{setBool}\ \mathit{True}).\mathit{mput}_R\ (a,s_1); \\
& \qquad (c,s_2') \leftarrow (\mathit{setBool}\ \mathit{False}).\mathit{mput}_R\ (b,s_2); \\
& \qquad (c',(s_1'',s_2'')) \leftarrow \mathit{return}\ (c,(s_1',s_2')); \\
& \qquad (b',s_2''') \leftarrow (\mathit{setBool}\ \mathit{False}).\mathit{mput}_L\ (c',s_2''); \\
& \qquad (a',s_2''') \leftarrow (\mathit{setBool}\ \mathit{True}).\mathit{mput}_L\ (b',s_1''); \\
& \qquad \mathit{return}\ (c,(s_1''',s_2'''))\} \\
= & [\![\text{ monad unit }]\!] \\
& \mathbf{do}\ \{(b,s_1') \leftarrow (\mathit{setBool}\ \mathit{True}).\mathit{mput}_R\ (a,s_1); \\
& \qquad (c,s_2') \leftarrow (\mathit{setBool}\ \mathit{False}).\mathit{mput}_R\ (b,s_2); \\
& \qquad (b',s_2''') \leftarrow (\mathit{setBool}\ \mathit{False}).\mathit{mput}_L\ (c',s_2'); \\
& \qquad (a',s_2''') \leftarrow (\mathit{setBool}\ \mathit{True}).\mathit{mput}_L\ (b',s_1');
\end{array}$$

$$\begin{aligned}
&\qquad \mathit{return}\ (c, (s_1''', s_2'''))\,\}\\
=\ &\llbracket\ \text{(PutRLM) for } \mathit{setBool\ False}\ \rrbracket\\
&\mathbf{do}\ \{\,(b, s_1') \leftarrow (\mathit{setBool\ True}).\mathit{mput}_R\ (a, s_1);\\
&\qquad (c, s_2') \leftarrow (\mathit{setBool\ False}).\mathit{mput}_R\ (b, s_2);\\
&\qquad (b', s_2''') \leftarrow \mathit{return}\ (b, s_2');\\
&\qquad (a', s_2''') \leftarrow (\mathit{setBool\ False}).\mathit{mput}_L\ (b', s_1');\\
&\qquad \mathit{return}\ (c, (s_1''', s_2'''))\,\}\\
=\ &\llbracket\ \text{monad unit}\ \rrbracket\\
&\mathbf{do}\ \{\,(b, s_1') \leftarrow (\mathit{setBool\ True}).\mathit{mput}_R\ (a, s_1);\\
&\qquad (c, s_2') \leftarrow (\mathit{setBool\ False}).\mathit{mput}_R\ (b, s_2);\\
&\qquad (a', s_2''') \leftarrow (\mathit{setBool\ True}).\mathit{mput}_L\ (b, s_1');\\
&\qquad \mathit{return}\ (c, (s_1''', s_2'))\,\}
\end{aligned}$$

However, we cannot simplify this any further. Moreover, it should be clear that the shared state will be *True* after this operation is performed. Considering the other side of the desired equation:

$$\begin{aligned}
&\mathbf{do}\ \{\,(c, s') \leftarrow \mathit{sl}.\mathit{mput}_R\ (a, s); \mathit{sl}.\mathit{mput}_L\ (c, s'')\,\}\\
=\ &\llbracket\ \text{let } s = (s_1, s_2) \text{ and } s' = (s_1''', s_2'''); \text{ Definition}\ \rrbracket\\
&\mathbf{do}\ \{\,(b, s_1') \leftarrow (\mathit{setBool\ True}).\mathit{mput}_R\ (a, s_1);\\
&\qquad (c, s_2') \leftarrow (\mathit{setBool\ False}).\mathit{mput}_R\ (b, s_2);\\
&\qquad (c', (s_1'', s_2'')) \leftarrow \mathit{return}\ (c, (s_1', s_2'));\\
&\qquad \mathit{return}\ (c', (s_1'', s_2''))\,\}\\
=\ &\llbracket\ \text{Monad unit}\ \rrbracket\\
&\mathbf{do}\ \{\,(b, s_1') \leftarrow (\mathit{setBool\ True}).\mathit{mput}_R\ (a, s_1);\\
&\qquad (c, s_2') \leftarrow (\mathit{setBool\ False}).\mathit{mput}_R\ (b, s_2);\\
&\qquad \mathit{return}\ (c, (s_1', s_2'))\,\}
\end{aligned}$$

it should be clear that the shared state will be *False* after this operation is performed. Therefore, (PutRLM) is not satisfied by *sl*. □

We can show that for *commutative* monads T, composition preserves well-behavedness:

Theorem 42. If sl_1 and sl_2 are well-behaved symmetric lenses over commutative monad T, then $\mathit{sl}_1 \mathbin{;} \mathit{sl}_2$ is well-behaved. ♢

Proof. We need to show that $\mathit{sl} = \mathit{sl}_1 \mathbin{;} \mathit{sl}_2$ satisfies (PutLRM) and (PutRLM). We show (PutLRM), and appeal to symmetry for the proof of the other law.

$$\begin{aligned}
&\mathbf{do}\ \{\,(z, c') \leftarrow \mathit{sl}.\mathit{mput}_R\ (x, c); \mathit{sl}.\mathit{mput}_L\ (z, c')\,\}\\
=\ &\llbracket\ \text{eta expansion}\ \rrbracket\\
&\mathbf{do}\ \{\,(z, (c_1', c_2')) \leftarrow \mathit{sl}.\mathit{mput}_R\ (x, (c_1, c_2)); \mathit{sl}.\mathit{mput}_L\ (z, (c_1', c_2'))\,\}\\
=\ &\llbracket\ \text{definition}\ \rrbracket\\
&\mathbf{do}\ \{\,(y, c_1'') \leftarrow \mathit{sl}_1.\mathit{mput}_R\ (x, c_1);\\
&\qquad (z, c_2'') \leftarrow \mathit{sl}_2.\mathit{mput}_R\ (y, c_2);
\end{aligned}$$

$$
\begin{array}{ll}
 & \qquad (z,(c'_1,c'_2)) \leftarrow \mathit{return}\ (z,(c''_1,c''_2));\\
 & \qquad (y',c'''_2) \leftarrow \mathit{sl}_2.\mathit{mput}_L\ (z,c'_2);\\
 & \qquad (x',c'''_1) \leftarrow \mathit{sl}_1.\mathit{mput}_L\ (y',c'_1);\\
 & \qquad \mathit{return}\ (x',(c'''_1,c'''_2))\,\}\\
= & \llbracket\ \text{monad unit}\ \rrbracket\\
 & \mathbf{do}\,\{\,(y,c''_1) \leftarrow \mathit{sl}_1.\mathit{mput}_R\ (x,c_1);\\
 & \qquad (z,c''_2) \leftarrow \mathit{sl}_2.\mathit{mput}_R\ (y,c_2);\\
 & \qquad (y',c'''_2) \leftarrow \mathit{sl}_2.\mathit{mput}_L\ (z,c''_2);\\
 & \qquad (x',c'''_1) \leftarrow \mathit{sl}_1.\mathit{mput}_L\ (y',c''_1);\\
 & \qquad \mathit{return}\ (x',(c'''_1,c'''_2))\,\}\\
= & \llbracket\ \text{(PutRLM)}\ \rrbracket\\
 & \mathbf{do}\,\{\,(y,c''_1) \leftarrow \mathit{sl}_1.\mathit{mput}_R\ (x,c_1);\\
 & \qquad (z,c''_2) \leftarrow \mathit{sl}_2.\mathit{mput}_R\ (y,c_2);\\
 & \qquad (y',c'''_2) \leftarrow \mathit{return}\ (z,c''_2);\\
 & \qquad (x',c'''_1) \leftarrow \mathit{sl}_1.\mathit{mput}_L\ (y',c''_1);\\
 & \qquad \mathit{return}\ (x',(c'''_1,c'''_2))\,\}\\
= & \llbracket\ \text{Monad unit}\ \rrbracket\\
 & \mathbf{do}\,\{\,(y,c''_1) \leftarrow \mathit{sl}_1.\mathit{mput}_R\ (x,c_1);\\
 & \qquad (z,c''_2) \leftarrow \mathit{sl}_2.\mathit{mput}_R\ (y,c_2);\\
 & \qquad (x',c'''_1) \leftarrow \mathit{sl}_1.\mathit{mput}_L\ (y,c''_1);\\
 & \qquad \mathit{return}\ (x',(c'''_1,c''_2))\,\}\\
= & \llbracket\ \text{Commutativity of } T \text{ (or just of } \mathit{mput}_R \text{ or } \mathit{mput}_L\text{)}\ \rrbracket\\
 & \mathbf{do}\,\{\,(y,c''_1) \leftarrow \mathit{sl}_1.\mathit{mput}_R\ (x,c_1);\\
 & \qquad (x',c'''_1) \leftarrow \mathit{sl}_1.\mathit{mput}_L\ (y,c''_1);\\
 & \qquad (z,c''_2) \leftarrow \mathit{sl}_2.\mathit{mput}_R\ (y,c_2);\\
 & \qquad \mathit{return}\ (x',(c'''_1,c''_2))\,\}\\
= & \llbracket\ \text{(PutRLM)}\ \rrbracket\\
 & \mathbf{do}\,\{\,(y,c''_1) \leftarrow \mathit{sl}_1.\mathit{mput}_R\ (x,c_1);\\
 & \qquad (x',c'''_1) \leftarrow \mathit{return}\ (x,c_1)\\
 & \qquad (z,c''_2) \leftarrow \mathit{sl}_2.\mathit{mput}_R\ (y,c_2);\\
 & \qquad \mathit{return}\ (x',(c'''_1,c''_2))\,\}\\
= & \llbracket\ \text{monad unit}\ \rrbracket\\
 & \mathbf{do}\,\{\,(y,c''_1) \leftarrow \mathit{sl}_1.\mathit{mput}_R\ (x,c_1);\\
 & \qquad (z,c''_2) \leftarrow \mathit{sl}_2.\mathit{mput}_R\ (y,c_2);\\
 & \qquad \mathit{return}\ (x,(c''_1,c''_2))\,\}\\
= & \llbracket\ \text{definition}\ \rrbracket\\
 & \mathbf{do}\,\{\,(z,(c''_1,c''_2)) \leftarrow \mathit{sl}.\mathit{mput}_R\ (x,(c_1,c_2)); \mathit{return}\ (x,(c''_1,c''_2))\,\}\\
= & \llbracket\ \text{eta contraction}\ \rrbracket\\
 & \mathbf{do}\,\{\,(z,c') \leftarrow \mathit{sl}.\mathit{mput}_R\ (x,c); \mathit{return}\ (x,c')\,\}
\end{array}
$$

□

Some commutativity assumption is necessary: in order to apply (PutLRM) for sl_1, we need to be able to reorder the operations $sl_2.mput_R$ and $sl_1.mput_L$, and this is not possible in an arbitrary (non-commutative) monad such as $T = \mathit{State}\ \mathit{Bool}$.

Mapping monadic symmetric lenses to effectful bx Recall the abbreviations (*fst3* was introduced originally in Section 6):

$$\begin{aligned}
&\mathit{fst3}\;(a,\_,\_)\;\; = a\\
&\mathit{snd3}\;(\_,b,\_) = b\\
&\mathit{thd3}\;(\_,\_,c)\; = c
\end{aligned}$$

Given $\mathit{sl} :: \mathit{SMLens}\;\;T\;\;C\;\;A\;\;B$, let $S \subseteq A \times B \times C$ be the set of *consistent triples* (a, b, c), that is, those for which $\mathit{sl}.\mathit{mput}_R\;\;a\;\;c = \mathit{return}\;(b, c)$ and $\mathit{sl}.\mathit{mput}_L\;\;b\;\;c = \mathit{return}\;(a, c)$. We construct $\mathit{bx} :: \mathit{InitStateTBX}\;\;T\;\;S\;\;A\;\;B$ by

$$\begin{aligned}
&\mathit{smlens2bx}\;\mathit{sl} = \mathit{InitStateTBX}\;(\mathit{gets}\;\mathit{fst3})\;\mathit{set}_L\;\mathit{init}_L\;(\mathit{gets}\;\mathit{snd3})\;\mathit{set}_R\;\mathit{init}_R\;\mathbf{where}\\
&\quad \mathit{set}_L\;a' \;= \mathbf{do}\;\{\,c \leftarrow \mathit{gets}\;\mathit{thd3};(b',c') \leftarrow \mathit{lift}\;(\mathit{sl}.\mathit{mput}_R\;(a',c));\mathit{set}\;(a',b',c')\,\}\\
&\quad \mathit{set}_R\;b' \;= \mathbf{do}\;\{\,c \leftarrow \mathit{gets}\;\mathit{thd3};(a',c') \leftarrow \mathit{lift}\;(\mathit{sl}.\mathit{mput}_L\;(b',c));\mathit{set}\;(a',b',c')\,\}\\
&\quad \mathit{init}_L\;a = \mathbf{do}\;\{\,(b,c) \leftarrow \mathit{sl}.\mathit{mput}_R\;(a,\mathit{sl}.\mathit{missing});\mathit{return}\;(a,b,c)\,\}\\
&\quad \mathit{init}_R\;b = \mathbf{do}\;\{\,(a,c) \leftarrow \mathit{sl}.\mathit{mput}_L\;(b,\mathit{sl}.\mathit{missing});\mathit{return}\;(a,b,c)\,\}
\end{aligned}$$

However, *smlens2bx* may not preserve well-behavedness even for commutative monads such as *Maybe*, as the following counterexample illustrates:

Example 43. Consider the following monadic symmetric lens construction:

$$\begin{aligned}
&\mathit{fail} :: \mathit{SMLens}\;\mathit{Maybe}\;()\;()\;()\\
&\mathit{fail} = \mathit{SMLens}\;m\;m\;()\;\mathbf{where}\;m\;\_ = \mathit{Nothing}
\end{aligned}$$

This is well-behaved but *smlens2bx fail* is not (e.g. it does not obey ($\mathrm{G_LS_L}$)). ♢

Proposition 44. *fail* is well-behaved. ♢

Proof. We consider (PutRLM); (PutLRM) is symmetric.

$$\begin{aligned}
&\quad \mathbf{do}\;\{\,(b,c') \leftarrow \mathit{fail}.\mathit{mput}_R\;((),());\mathit{fail}.\mathit{mput}_L\;(b,c')\,\}\\
&= \quad [\![\text{ Definition }]\!]\\
&\quad \mathbf{do}\;\{\,(b,c') \leftarrow \mathit{Nothing};\mathit{Nothing}\,\}\\
&= \quad [\![\;\mathit{Nothing}\text{ is a zero element }]\!]\\
&\quad \mathit{Nothing}\\
&= \quad [\![\;\mathit{Nothing}\text{ is a zero element }]\!]\\
&\quad \mathbf{do}\;\{\,(b,c') \leftarrow \mathit{Nothing};\mathit{return}\;((),c')\,\}\\
&= \quad [\![\text{ Definition }]\!]\\
&\quad \mathbf{do}\;\{\,(b,c') \leftarrow \mathit{fail}.\mathit{mput}_R\;((),());\mathit{return}\;((),c')\,\}
\end{aligned}$$

□

Proposition 45. *smlens2bx fail* is not well-behaved. ♢

Proof. We reason as follows

$$\begin{aligned}
&\quad \mathbf{do}\;\{\,a \leftarrow (\mathit{sm2lens}\;\mathit{fail}).\mathit{get}_L;(\mathit{smlens2bx}\;\mathit{fail}).\mathit{set}_L\;a\,\}\\
&= \quad [\![\text{ Definition }]\!]
\end{aligned}$$

$$\begin{aligned}
&\quad \mathbf{do}\,\{\,a \leftarrow \mathit{gets}\ \mathit{fst3}; c \leftarrow \mathit{gets}\ \mathit{thd3}; (b', c') \leftarrow \mathit{lift}\ \mathit{fail}.\mathit{mput}_R\ (a, c); \mathit{set}\ (a, b', c')\,\}\\
&= \quad \llbracket \text{ Definition of } \mathit{gets}, \text{(GG), monad laws } \rrbracket\\
&\quad \mathbf{do}\,\{\,(a, b, c) \leftarrow \mathit{get}; (b', c') \leftarrow \mathit{lift}\ \mathit{fail}.\mathit{mput}_R\ (a, c); \mathit{set}\ (a, b', c')\,\}\\
&= \quad \llbracket \text{ Definition of } \mathit{setBool} \ \rrbracket\\
&\quad \mathbf{do}\,\{\,(a, b, c) \leftarrow \mathit{get}; (b', c') \leftarrow \mathit{lift}\ (\mathit{Nothing}); \mathit{set}\ (a, b', c')\,\}\\
&= \quad \llbracket\ \mathit{lift} \text{ a monad morphism } \rrbracket\\
&\quad \mathbf{do}\,\{\,(a, b, c) \leftarrow \mathit{get}; (\_, c'') \leftarrow \mathit{lift}\ (\mathit{Nothing}); (b', c') \leftarrow \mathit{return}\ ((), ()); \mathit{set}\ (a, b', c')\,\}\\
&= \quad \llbracket \text{ monad unit } \rrbracket\\
&\quad \mathbf{do}\,\{\,(a, b, c) \leftarrow \mathit{get}; (\_, c'') \leftarrow \mathit{lift}\ (\mathit{Nothing}); \mathit{set}\ (a, (), ())\,\}\\
&= \quad \llbracket \text{ Nothing is a zero element } \rrbracket\\
&\quad \mathit{Nothing}
\end{aligned}$$

This is not equal to *return* (), because it always fails. □

For pure symmetric lenses, *smlens2bx* does preserve well-behavedness.

Theorem 46. If $sl :: \mathit{SMLens}\ \mathit{Id}\ C\ A\ B$ is well-behaved, then $\mathit{smlens2bx}\ sl$ is also well-behaved, with state space S consisting of the consistent triples of sl. ♢

Proof. First we show that, given a symmetric lens sl, the operations of $bx = \mathit{smlens2bx}\ sl$ preserve consistency of the state. Assume (a, b, c) is consistent. To show that $bx.\mathit{set}_L\ a'$ preserves consistency for any a', we have to show that (a', b', c') is consistent, where a' is arbitrary and $(b', c') = sl.\mathit{mput}_R\ (a', c)$. For one half of consistency, we have:

$$\begin{aligned}
&\quad sl.\mathit{mput}_L\ (b', c')\\
&= \quad \llbracket\ sl.\mathit{mput}_R\ (a', c) = (b', c'), \text{ and (PutRLM) } \rrbracket\\
&\quad (a', c')
\end{aligned}$$

and then for the other half:

$$\begin{aligned}
&\quad sl.\mathit{mput}_R\ (a', c')\\
&= \quad \llbracket \text{ above, and (PutLRM) } \rrbracket\\
&\quad (b', c')
\end{aligned}$$

as required. The proof that $bx.\mathit{set}_R\ b'$ also preserves consistency is dual.

We will now show that *smlens2bx sl* satisfies the bx laws for any symmetric lens *sl*. For ($\mathrm{G_L S_L}$), we proceed as follows:

$$\begin{aligned}
&\quad \mathbf{do}\,\{\,a \leftarrow \mathit{get}_L; \mathit{set}_L\ a\,\}\\
&= \quad \llbracket \text{ definition of } \mathit{get}_L, \mathit{gets}\ \mathit{fst3} \ \rrbracket\\
&\quad \mathbf{do}\,\{\,(a_1, b_1, c_1) \leftarrow \mathit{get}; a \leftarrow \mathit{return}\ a_1; \mathit{set}_L\ a\,\}\\
&= \quad \llbracket \text{ monad unit, definition of } \mathit{set}_L, \mathit{thd3} \ \rrbracket\\
&\quad \mathbf{do}\,\{\,(a_1, b_1, c_1) \leftarrow \mathit{get}; (a_2, b_2, c_2) \leftarrow \mathit{get}; (b, c) \leftarrow sl.\mathit{mput}_R\ (a_1, c_2); \mathit{set}\ (a_1, b, c)\,\}\\
&= \quad \llbracket\ \mathit{get} \text{ is copyable } \rrbracket\\
&\quad \mathbf{do}\,\{\,(a_1, b_1, c_1) \leftarrow \mathit{get}; (b, c) \leftarrow sl.\mathit{mput}_R\ (a_1, c_1); \mathit{set}\ (a_1, b, c)\,\}
\end{aligned}$$

$$
\begin{array}{ll}
= & [\![\ \text{consistency of } (a_1, b_1, c_1)\ ]\!] \\
 & \mathbf{do}\ \{(a_1, b_1, c_1) \leftarrow \mathit{get}; (b, c) \leftarrow \mathit{return}\ (b_1, c_1); \mathit{set}\ (a_1, b, c)\} \\
= & [\![\ \text{let-binding}\ ]\!] \\
 & \mathbf{do}\ \{(a_1, b_1, c_1) \leftarrow \mathit{get}; \mathit{set}\ (a_1, b_1, c_1)\} \\
= & [\![\ \text{(GS) for state monad}\ ]\!] \\
 & \mathit{return}\ ()
\end{array}
$$

For ($\mathrm{S_L G_L}$), we have:

$$
\begin{array}{ll}
 & \mathbf{do}\ \{\mathit{set}_L\ a; \mathit{get}_L\} \\
= & [\![\ \text{definition of } \mathit{set}_L,\ \mathit{get}_L\ ]\!] \\
 & \mathbf{do}\ \{(a_1, b_1, c_1) \leftarrow \mathit{get}; (b, c) \leftarrow \mathit{sl.mput}_R\ (a, c_1); \mathit{set}\ (a, b, c); (a_2, b_2, c_2) \leftarrow \mathit{get}; \mathit{return}\ a_2\} \\
= & [\![\ \text{(SG) for state monad; monad unit}\ ]\!] \\
 & \mathbf{do}\ \{(a_1, b_1, c_1) \leftarrow \mathit{get}; (b, c) \leftarrow \mathit{sl.mput}_R\ (a, c_1); \mathit{set}\ (a, b, c); \mathit{return}\ a\} \\
= & [\![\ \text{definition of } \mathit{set}_L\ ]\!] \\
 & \mathbf{do}\ \{\mathit{set}_L\ a; \mathit{return}\ a\}
\end{array}
$$

$\square$

Mapping well-behaved effectful bx to monadic symmetric lenses Conversely, given $\mathit{bx} :: \mathit{InitStateTBX}\ T\ S\ A\ B$, we construct a symmetric lens $\mathit{sl} :: \mathit{SMLens}\ T\ (\mathit{Maybe}\ S)\ A\ B$ by

$$
\begin{array}{l}
\mathit{bx2smlens}\ \mathit{bx} = \mathit{SMLens}\ \mathit{mput}_R\ \mathit{mput}_L\ \mathit{missing}\ \mathbf{where} \\
\quad \mathit{mput}_R\ (a, \mathit{Just}\ s) \quad\ \ = \mathbf{do}\ \{(b, s') \leftarrow \mathbf{do}\ \{\mathit{bx.set}_L\ a; \mathit{bx.get}_R\}\ s; \mathit{return}\ (b, \mathit{Just}\ s')\} \\
\quad \mathit{mput}_R\ (a, \mathit{Nothing}) = \mathbf{do}\ \{(b, s') \leftarrow \mathit{bx.run}_L\ (\mathit{bx.get}_R)\ a; \mathit{return}\ (b, \mathit{Just}\ s')\} \\
\quad \mathit{mput}_L \qquad\qquad\quad\ = \ldots \quad \text{-- dual} \\
\quad \mathit{missing} = \mathit{Nothing}
\end{array}
$$

where we define run_L as follows:

$$
\begin{array}{l}
\mathit{run}_L :: \mathit{Monad}\ \tau \Rightarrow \mathit{InitStateTBX}\ \tau\ \sigma\ \alpha\ \beta \to \mathit{StateT}\ \sigma\ \tau\ \gamma \to \alpha \to \tau\ (\gamma, \sigma) \\
\mathit{run}_L\ \mathit{bx}\ m\ a = \mathbf{do}\ \{s \leftarrow \mathit{bx.init}_L\ a; m\ s\}
\end{array}
$$

and symmetrically for run_R. Essentially, these operations adapt a computation m in $\mathit{StateT}\ S\ T\ C$ to run starting with an A or B value, using init_L to build the initial S state. Note that $\mathit{run}_L\ \mathit{bx}\ m\ a$ produces a computation, not a pure value. Thus, as with monadic programming in Haskell generally, to run such a computation we may need to construct a suitable monad T, using for example Haskell's library of monad transformers.

Well-behavedness is preserved by the conversion from $\mathit{StateTBX}$ to SMLens, for arbitrary monads T:

Theorem 47. If $\mathit{bx} :: \mathit{StateTBX}\ T\ S\ A\ B$ is well-behaved, then $\mathit{bx2smlens}\ \mathit{bx}$ is also well-behaved. $\diamondsuit$

Proof. Let $\mathit{sl} = \mathit{bx2smlens}\ \mathit{bx}$. We need to show that the laws (PutRLM) and (PutLRM) hold. We show (PutRLM), and (PutLRM) is symmetric.

We need to show that

$$
\begin{array}{l}
\quad \mathbf{do}\,\{(b', \mathit{mc}') \leftarrow \mathit{sl.mput}_R\ (a, \mathit{mc}); \mathit{sl.mput}_L\ (b', \mathit{mc}')\}\\
=\\
\quad \mathbf{do}\,\{(b', \mathit{mc}') \leftarrow \mathit{sl.mput}_R\ (a, \mathit{mc}); \mathit{return}\ (a, \mathit{mc}')\}
\end{array}
$$

There are two cases, depending on whether the initial state mc is $\mathit{Nothing}$ or $\mathit{Just}\ c$ for some c.

If $\mathit{mc} = \mathit{Nothing}$ then we reason as follows:

$$
\begin{array}{ll}
& \mathbf{do}\,\{(b', \mathit{mc}') \leftarrow \mathit{sl.mput}_R\ (a, \mathit{Nothing}); \mathit{sl.mput}_L\ (b', \mathit{mc}')\}\\
= & [\![\ \text{Definition}\ ]\!]\\
& \mathbf{do}\,\{(b, s') \leftarrow \mathit{bx.run}_L\ (\mathit{bx.get}_R)\ a; (b', \mathit{mc}') \leftarrow \mathit{return}\ (b, \mathit{Just}\ s')\}; \mathit{sl.mput}_L\ (b', \mathit{mc}')\}\\
= & [\![\ \text{monad unit}\ ]\!]\\
& \mathbf{do}\,\{(b, s') \leftarrow \mathit{bx.run}_L\ (\mathit{bx.get}_R)\ a; \mathit{sl.mput}_L\ (b, \mathit{Just}\ s')\}\\
= & [\![\ \text{definition}\ ]\!]\\
& \mathbf{do}\,\{(b, s') \leftarrow \mathit{bx.run}_L\ (\mathit{bx.get}_R)\ a;\\
& \qquad (a', s'') \leftarrow \mathbf{do}\,\{\mathit{bx.set}_R\ b; \mathit{bx.get}_L\}\ s'; \mathit{return}\ (a', \mathit{Just}\ s'')\}\\
= & [\![\ \text{Definition of } \mathit{run}_L\ ]\!]\\
& \mathbf{do}\,\{s \leftarrow \mathit{bx.init}_L\ a; (b, s') \leftarrow (\mathit{bx.get}_R)\ s;\\
& \qquad (a', s'') \leftarrow \mathbf{do}\,\{\mathit{bx.set}_R\ b; \mathit{bx.get}_L\}\ s'; \mathit{return}\ (a', \mathit{Just}\ s'')\}\\
= & [\![\ \text{definition of bind}\ ]\!]\\
& \mathbf{do}\,\{s \leftarrow \mathit{bx.init}_L\ a; (a', s'') \leftarrow \mathbf{do}\,\{b \leftarrow \mathit{bx.get}_R; \mathit{bx.set}_R\ b; \mathit{bx.get}_L\}\ s;\\
& \qquad \mathit{return}\ (a', \mathit{Just}\ s'')\}\\
= & [\![\ (\mathrm{G_R S_R})\ ]\!]\\
& \mathbf{do}\,\{s \leftarrow \mathit{bx.init}_L\ a; (a', s'') \leftarrow \mathbf{do}\,\{\_ \leftarrow \mathit{return}\ (); \mathit{bx.get}_L\}\ s;\\
& \qquad \mathit{return}\ (a', \mathit{Just}\ s'')\}\\
= & [\![\ \text{definition of } \mathit{return}\ ]\!]\\
& \mathbf{do}\,\{s \leftarrow \mathit{bx.init}_L\ a; (a', s'') \leftarrow \mathit{bx.get}_L\ s; \mathit{return}\ (a', \mathit{Just}\ s'')\}\\
= & [\![\ (\mathrm{I_L G_L})\ \text{law}\ ]\!]\\
& \mathbf{do}\,\{s \leftarrow \mathit{bx.init}_L\ a; (a', s'') \leftarrow \mathit{return}\ (a, s); \mathit{return}\ (a', \mathit{Just}\ s'')\}\\
= & [\![\ \text{Monad unit}\ ]\!]\\
& \mathbf{do}\,\{s \leftarrow \mathit{bx.init}_L\ a; \mathit{return}\ (a, \mathit{Just}\ s)\}\\
= & [\![\ \mathit{get}\ \text{discardable}\ ]\!]\\
& \mathbf{do}\,\{s \leftarrow \mathit{bx.init}_L\ a; (b, s') \leftarrow (\mathit{bx.get}_R)\ s; \mathit{return}\ (a, \mathit{Just}\ s')\}\\
= & [\![\ \text{Monad unit}\ ]\!]\\
& \mathbf{do}\,\{s \leftarrow \mathit{bx.init}_L\ a; (b, s') \leftarrow (\mathit{bx.get}_R)\ s; (b', \mathit{mc}') \leftarrow \mathit{return}\ (b, \mathit{Just}\ s');\\
& \qquad \mathit{return}\ (a, \mathit{mc}')\}\\
= & [\![\ \text{Definition of } \mathit{run}_L\ ]\!]\\
& \mathbf{do}\,\{(b, s') \leftarrow \mathit{bx.run}_L\ (\mathit{bx.get}_R)\ a; (b', \mathit{mc}') \leftarrow \mathit{return}\ (b, \mathit{Just}\ s'); \mathit{return}\ (a, \mathit{mc}')\}\\
= & [\![\ \text{Definition}\ ]\!]\\
& \mathbf{do}\,\{(b', \mathit{mc}') \leftarrow \mathit{sl.mput}_R\ (a, \mathit{Nothing}); \mathit{return}\ (a, \mathit{mc}')\}
\end{array}
$$

If $mc = \mathit{Just}\ c$ then we reason as follows:

$$
\begin{array}{ll}
 & \mathbf{do}\ \{(b', mc') \leftarrow \mathit{sl}.\mathit{mput}_R\ (a, \mathit{Just}\ c); \mathit{sl}.\mathit{mput}_L\ (b', mc')\} \\
= & [\![\ \text{Definition}\ ]\!] \\
 & \mathbf{do}\ \{(b, c') \leftarrow \mathbf{do}\ \{\mathit{bx}.\mathit{set}_L\ a; \mathit{bx}.\mathit{get}_R\}\ c; (b, mc') \leftarrow \mathit{return}\ (b', \mathit{Just}\ c'); \\
 & \qquad \mathit{sl}.\mathit{mput}_L\ (b', mc')\} \\
= & [\![\ \text{monad unit}\ ]\!] \\
 & \mathbf{do}\ \{(b, c') \leftarrow \mathbf{do}\ \{\mathit{bx}.\mathit{set}_L\ a; \mathit{bx}.\mathit{get}_R\}\ c; \mathit{sl}.\mathit{mput}_L\ (b', \mathit{Just}\ c')\} \\
= & [\![\ \text{definition}\ ]\!] \\
 & \mathbf{do}\ \{(b, c') \leftarrow \mathbf{do}\ \{\mathit{bx}.\mathit{set}_L\ a; \mathit{bx}.\mathit{get}_R\}\ c; \\
 & \qquad (a', c'') \leftarrow \mathbf{do}\ \{\mathit{bx}.\mathit{set}_R\ b; \mathit{bx}.\mathit{get}_L\}\ c'; \mathit{return}\ (a', \mathit{Just}\ c'')\} \\
= & [\![\ \text{definition}\ ]\!] \\
 & \mathbf{do}\ \{(a', c'') \leftarrow \mathbf{do}\ \{\mathit{bx}.\mathit{set}_L\ a; b' \leftarrow \mathit{bx}.\mathit{get}_R; \mathit{bx}.\mathit{set}_R\ b'; \mathit{bx}.\mathit{get}_L\}\ c; \\
 & \qquad \mathit{return}\ (a', \mathit{Just}\ c'')\} \\
= & [\![\ (\mathrm{G_R S_R})\ ]\!] \\
 & \mathbf{do}\ \{(a', c'') \leftarrow \mathbf{do}\ \{\mathit{bx}.\mathit{set}_L\ a; \mathit{bx}.\mathit{get}_L\}\ c; \mathit{return}\ (a', \mathit{Just}\ c'')\} \\
= & [\![\ (\mathrm{S_L G_L})\ ]\!] \\
 & \mathbf{do}\ \{(a', c'') \leftarrow \mathbf{do}\ \{\mathit{bx}.\mathit{set}_L\ a; \mathit{return}\ a\}\ c; \mathit{return}\ (a', \mathit{Just}\ c'')\} \\
= & [\![\ \text{definition}\ ]\!] \\
 & \mathbf{do}\ \{(\_, c') \leftarrow \mathit{bx}.\mathit{set}_L\ a\ c; (a', c'') \leftarrow \mathbf{do}\ \{\mathit{return}\ a\}\ c'; \\
 & \qquad \mathit{return}\ (a', \mathit{Just}\ c'')\} \\
= & [\![\ \text{definition of } \mathit{return}\ ]\!] \\
 & \mathbf{do}\ \{(\_, c') \leftarrow \mathit{bx}.\mathit{set}_L\ a\ c; \mathit{return}\ (a, \mathit{Just}\ c')\} \\
= & [\![\ \mathit{get}_R \text{ discardable}\ ]\!] \\
 & \mathbf{do}\ \{(\_, c'') \leftarrow \mathit{bx}.\mathit{set}_L\ a\ c; (b', c') \leftarrow \mathit{bx}.\mathit{get}_R\ c''; \mathit{return}\ (a, \mathit{Just}\ c'')\} \\
= & [\![\ \mathit{get}_R \text{ a query so } c' = c''\ ]\!] \\
 & \mathbf{do}\ \{(\_, c'') \leftarrow \mathit{bx}.\mathit{set}_L\ a\ c; (b', c') \leftarrow \mathit{bx}.\mathit{get}_R\ c''; \mathit{return}\ (a, \mathit{Just}\ c')\} \\
= & [\![\ \text{Definition, monad unit}\ ]\!] \\
 & \mathbf{do}\ \{(b', mc') \leftarrow \mathbf{do}\ \{(b', c') \leftarrow \mathbf{do}\ \{\mathit{bx}.\mathit{set}_L\ a; \mathit{bx}.\mathit{get}_R\}\ c; \mathit{return}\ (b, \mathit{Just}\ c')\}; \\
 & \qquad \mathit{return}\ (a, mc')\} \\
= & [\![\ \text{Definition}\ ]\!] \\
 & \mathbf{do}\ \{(b', mc') \leftarrow \mathit{sl}.\mathit{mput}_R\ (a, \mathit{Just}\ c); \mathit{return}\ (a, mc')\}
\end{array}
$$

□

A.3 Pure symmetric lenses, initialisation and equivalence

The issues of initialisation and equivalence for symmetric lenses are subtly interrelated, as recently explored by Johnson and Rosebrugh [16]. HPW included a *missing* complement value in their definition of symmetric lenses to account for initialisation, and they defined a notion of equivalence of symmetric lenses that requires their *missing* values to have similar behaviour. In contrast, we handle initialisation using *init* functions and take monad isomorphisms (induced by state space isomorphisms) as our notion of equivalence.

As noted above, extending symmetric lenses to incorporate effects is problematic. Even ignoring these problems, it is not obvious how to extend HPW's notion of equivalence to monadic symmetric lenses. Therefore, we focus in the rest of this section on the effect-free case, and use the concrete types *SLens* (defined earlier) and *InitStateBX*, defined as follows:

```
data InitStateBX σ α β =
  InitStateBX { get_L :: State σ α, set_L :: α → State σ (), init_L :: α → σ,
                get_R :: State σ β, set_R :: β → State σ (), init_R :: β → σ }
```

Moreover, we use the following functions mapping between the two:

```
bx2slens :: InitStateBX σ α β → SLens (Maybe σ) α β
bx2slens bx = SLens put_R put_L missing where
  put_R (a, Just s)  = let (b, s') = do {bx.set_L a; bx.get_R} s in (b, Just s')
  put_R (a, Nothing) = let (b, s') = (bx.get_R) (bx.init_L a) in (b, Just s')
  put_L              = ...  -- dual
  missing            = Nothing
slens2bx :: SLens γ α β → InitStateBX (α, β, γ) α β
slens2bx sl = InitStateBX get_L set_L init_L get_R set_R init_R where
  get_L   = gets fst3
  get_R   = gets snd3
  set_L a' = do { c ← gets thd3; let (b', c') = sl.put_R (a', c) in set (a', b', c')}
  set_R = ...   -- dual
  init_L a = let (b, c) = sl.put_R (a, sl.missing) in (a, b, c)
  init_R = ...   -- dual
```

These are essentially just specialisations to $\tau = Id$ of the definitions given above for monadic symmetric lenses. Observe that *bx2slens* and *slens2bx* are not inverses: in particular, $slens2bx\ (bx2slens\ bx)$ and bx do not even have the same type and likewise for $slens2bx\ (bx2slens\ bx)$ and bx. Nevertheless, we may reasonably ask whether they are equivalent in some sense. For the first question, using equivalences of bx based on monad isomorphisms, the answer is affirmative:

Theorem 48. If $bx :: InitStateBX\ S\ A\ B$ is well-behaved, then $bx \equiv slens2bx\ (bx2slens\ bx)$. ♢

Proof. Take $sl = bx2slens\ bx$ and $bx' = slens2bx\ sl$. Observe that their types are:

```
sl :: SLens (Maybe S) A B
bx' :: InitStateTBX S' A B
```

where S' is the set of all consistent triples (a, b, mc) such that $sl.mput_R\ (a, mc) = (b, mc)$ and $sl.mput_L\ (b, mc) = (a, mc)$. It is easy to see by the definition of *bx2slens* that mc is of the form $Just\ s$ for some s in every consistent triple (a, b, mc), and moreover that consistency entails that $a = bx.read_L\ s$ and $b = bx.read_R\ s$.

Therefore, it suffices to exhibit an isomorphism between S and S' that maps the operations of bx onto those of bx'. We define an isomorphism $h :: S \to S'$ on the state spaces as follows:

$$\begin{aligned}&h\ s = (bx.read_L\ s, bx.read_R\ s, Just\ s)\\&h^{-1}\ (a, b, Just\ s) = s\end{aligned}$$

It is straightforward (but tedious) to verify that h satisfies the following equations:

$$\begin{aligned}&(\iota\ h)\ (bx.get_L) &&= bx'.get_L\\&(\iota\ h)\ (bx.set_L\ a) &&= bx'.set_L\ a\\&(bx.init_L\ a) \ggg\!= gets\ h &&= bx'.init_L\ a\end{aligned}$$

and their duals (which are symmetric). □

If we consider the question whether *sl* and *bx2slens* (*slens2bx sl*) are equivalent, then we must first identify a suitable notion of equivalence. HPW defined equivalence of symmetric lenses as follows:

Definition 49. Suppose $r \subseteq C_1 \times C_2$. Then $f \sim_r g$ means that for all c_1, c_2, x, if $(c_1, c_2) \in r$ and $(y, c'_1) = f\ (x, c_1)$ and $(y', c'_2) = g\ (y, c_2)$, then $y = y'$ and $(c'_1, c'_2) \in r$. ♢

Definition 50 (HPW equivalence). Two symmetric lenses $sl_1 :: SLens\ C_1\ X\ Y$ and $sl_2 :: SLens\ C_2\ X\ Y$ are considered *equivalent* ($sl_1 \equiv_{sl} sl_2$) if there is a relation $r \subseteq C_1 \times C_2$ such that

1. $(sl_1.missing, sl_2.missing) \in r$,
2. $sl_1.put_R \sim_r sl_2.put_R$, and
3. $sl_1.put_L \sim_r sl_2.put_L$. ♢

We can now show that *sl* and *bx2slens* (*slens2bx sl*) are equivalent in this sense:

Theorem 51. If $sl :: SLens\ C\ A\ B$ is well-behaved then $sl \equiv_{sl} bx2slens\ (slens2bx\ sl)$. ♢

Proof. Take $bx = slens2bx\ sl$ and $sl' = bx2slens\ bx$. Observe that their types are:

$$\begin{aligned}&bx :: InitStateBX\ S'\ A\ B\\&sl :: SLens\ (Maybe\ S')\ A\ B\end{aligned}$$

where S' is the set of consistent triples (a, b, c) where $sl.put_R\ (a, c) = (b, c)$ and $sl.put_L\ (b, c) = (a, c)$.

Towards showing that *sl* and *sl′* are equivalent (according to Definition 50), we need a relation on the state spaces S and $Maybe\ S'$. Define relation r as $\{(sl.missing, Nothing)\} \cup \{(s, Just\ (a, b, s)) \mid (a, b, s) \in S'\}$.

We now proceed to check the conditions on r needed to conclude $sl \equiv sl'$. First, for part (a), note that $sl'.missing = Nothing$, so $(sl.missing, sl'.missing) \in r$.

Next, for part (b), we wish to show that $sl.put_R \sim_r sl'.put_R$. Suppose that $(s_1, s_2) \in r$, and let x be given. Let

$$\begin{aligned}&(y, s'_1) &&= sl.put_R\ (x, s_1)\\&(y', s'_2) &&= sl'.put_R\ (x, s_2)\end{aligned}$$

We need to show that $y = y'$ and $(s'_1, s'_2) \in r$. There are two cases: either $s_2 = \mathit{Nothing}$ or $s_2 = \mathit{Just}\ s$.

Case $s_2 = \mathit{Nothing}$: We first simplify as follows:

$$
\begin{array}{ll}
 & sl'.put_R\ (x, \mathit{Nothing}) \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
 & \mathbf{let}\ (b', s') = bx.get_R\ (bx.init_L\ x)\ \mathbf{in}\ (b', \mathit{Just}\ s') \\
= & \llbracket\ \text{definition of } bx.get_R\ \rrbracket \\
 & \mathbf{let}\ (b', s') = bx.get_R\ (bx.init_L\ x)\ \mathbf{in}\ (b', \mathit{Just}\ s') \\
= & \llbracket\ \text{definition of } get_R\ \rrbracket \\
 & \mathbf{let}\ (b', s') = gets\ snd3\ (bx.init_L\ x)\ \mathbf{in}\ (b', \mathit{Just}\ s') \\
= & \llbracket\ \text{definition of } init_L\ \rrbracket \\
 & \mathbf{let}\ (b', s') = gets\ snd3\ (\mathbf{let}\ (b, c) = sl.put_R\ (x, sl.missing)\ \mathbf{in}\ (x, b, c)) \\
 & \mathbf{in}\ (b', \mathit{Just}\ s') \\
= & \llbracket\ \text{rearrange } \mathbf{let}\ \rrbracket \\
 & \mathbf{let}\ (b, c) = sl.put_R\ (x, sl.missing) \\
 & \qquad (b', s') = gets\ snd3\ (x, b, c) \\
 & \mathbf{in}\ (b', \mathit{Just}\ s') \\
= & \llbracket\ \text{simplify } gets\ snd3\ (x, b, c) = (b, (x, b, c))\ \rrbracket \\
 & \mathbf{let}\ (b, c) = sl.put_R\ (x, sl.missing) \\
 & \qquad (b', s') = (b, (x, b, c)) \\
 & \mathbf{in}\ (b', \mathit{Just}\ s') \\
= & \llbracket\ \text{simplify}\ \rrbracket \\
 & \mathbf{let}\ (b, c) = sl.put_R\ (x, sl.missing)\ \mathbf{in}\ (b, \mathit{Just}\ (x, b, c)) \\
= & \llbracket\ \text{assumption}\ \rrbracket \\
 & \mathbf{let}\ (b, c) = (y, s'_1)\ \mathbf{in}\ (b, \mathit{Just}\ (x, b, c)) \\
= & \llbracket\ \text{simplify}\ \rrbracket \\
 & (y, \mathit{Just}\ (x, y, s'_1))
\end{array}
$$

Thus, $y' = y$ and $s'_2 = \mathit{Just}\ (x, y, s'_1)$. Moreover, $(s'_1, \mathit{Just}\ (x, b, s'_1)) \in r$.

Case $s_2 = \mathit{Just}\ s$. We first simplify as follows:

$$
\begin{array}{ll}
 & sl'.put_R\ (x, \mathit{Just}\ s) \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
 & \mathbf{let}\ (b, s') = \mathbf{do}\ \{bx.set_L\ x; bx.get_R\}\ s\ \mathbf{in}\ (b, \mathit{Just}\ s') \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
 & \mathbf{let}\ (b, s') = \mathbf{do}\ \{(a, b, c) \leftarrow get; \\
 & \qquad\qquad\qquad\quad \mathbf{let}\ (b', c') = sl.put_R\ (x, c); \\
 & \qquad\qquad\qquad\quad set\ (x, b', c'); \\
 & \qquad\qquad\qquad\quad (a'', b'', c'') \leftarrow get; return\ b''\}\ s \\
 & \mathbf{in}\ (b, \mathit{Just}\ s') \\
= & \llbracket\ \text{(SG)}\ \rrbracket \\
 & \mathbf{let}\ (b, s') = \mathbf{do}\ \{(a, b, c) \leftarrow get; \\
 & \qquad\qquad\qquad\quad \mathbf{let}\ (b', c') = sl.put_R\ (x, c);
\end{array}
$$

$$
\begin{array}{ll}
 & \qquad\qquad\qquad\quad \mathit{set}\ (x, b', c'); \mathit{return}\ b' \}\ s \\
 & \mathbf{in}\ (b, \mathit{Just}\ s') \\
= & \llbracket\ s = (a_0, b_0, c_0)\ \rrbracket \\
 & \mathbf{let}\ (b, s') = \mathbf{do}\ \{ (a, b, c) \leftarrow \mathit{get}; \\
 & \qquad\qquad\qquad\quad \mathbf{let}\ (b', c') = \mathit{sl}.\mathit{put}_R\ (x, c); \\
 & \qquad\qquad\qquad\quad \mathit{set}\ (x, b', c'); \mathit{return}\ b' \}\ (a_0, b_0, c_0) \\
 & \mathbf{in}\ (b, \mathit{Just}\ s') \\
= & \llbracket\ \text{definition of } \mathit{get}\ \rrbracket \\
 & \mathbf{let}\ (b, s') \leftarrow \mathbf{do}\ \{ \mathbf{let}\ (b', c') = \mathit{sl}.\mathit{put}_R\ (x, c_0); \\
 & \qquad\qquad\qquad\quad \mathit{set}\ (x, b', c'); \mathit{return}\ b' \}\ (a_0, b_0, c_0) \\
 & \mathbf{in}\ (b, \mathit{Just}\ s') \\
= & \llbracket\ \text{lift } \mathbf{let} \text{ out of } \mathbf{do}\ \rrbracket \\
 & \mathbf{let}\ (b', c') = \mathit{sl}.\mathit{put}_R\ (x, c_0) \\
 & \qquad (b, s') = \mathbf{do}\ \{\ \mathit{set}\ (x, b', c'); \mathit{return}\ b' \}\ (a_0, b_0, c_0) \\
 & \mathbf{in}\ (b, \mathit{Just}\ s') \\
= & \llbracket\ \text{definition of } \mathit{set}\ \rrbracket \\
 & \mathbf{let}\ (b', c') = \mathit{sl}.\mathit{put}_R\ (x, c_0) \\
 & \qquad (b, s') = \mathbf{do}\ \{\ \mathit{return}\ b' \}\ (x, b', c')\ \mathbf{in}\ (b, \mathit{Just}\ s') \\
= & \llbracket\ \text{definition of } \mathit{return}\ \rrbracket \\
 & \mathbf{let}\ (b', c') = \mathit{sl}.\mathit{put}_R\ (x, c_0) \\
 & \qquad (b, s') = \mathit{return}\ (b', (x, b', c'))\ \mathbf{in}\ \mathit{return}\ (b, \mathit{Just}\ s') \\
= & \llbracket\ \text{inline } \mathbf{let}\ \rrbracket \\
 & \mathbf{let}\ (b', c') = \mathit{sl}.\mathit{put}_R\ (x, c_0)\ \mathbf{in}\ \mathit{return}\ (b', \mathit{Just}\ (x, b', c')) \\
= & \llbracket\ \text{assumption}\ \rrbracket \\
 & \mathbf{let}\ (b', c') = (y, s'_1)\ \mathbf{in}\ (b', \mathit{Just}\ (x, b', c')) \\
= & \llbracket\ \text{inline } \mathbf{let}\ \rrbracket \\
 & (y, \mathit{Just}\ (x, y, s'_1))
\end{array}
$$

Thus, $y' = y$ and $s_2 = \mathit{Just}\ (x, y, s'_1)$. Moreover, $(s'_1, \mathit{Just}\ (x, y, s'_1)) \in r$. Therefore, in either case $\mathit{sl}.\mathit{put}_R \sim_r \mathit{sl}'.\mathit{put}_R$ holds. The proof of part (c), that $\mathit{sl}.\mathit{put}_L \sim_r \mathit{sl}'.\mathit{put}_L$ holds, is similar, so we conclude that $\mathit{sl} \equiv \mathit{sl}'$. □

Together, Theorems 46, 47, 48 and 51 show that in the pure case, our approach to bx is essentially the same as that given by symmetric lenses. However, our formulation naturally generalises to a class of effectful bx that is closed under composition in the presence of arbitrary effects, whereas the naive monadic extension of symmetric lenses is only closed under composition in the presence of commutative effects.

B Proofs from Section 2

Lemma 6. If the laws (GG) and (GS) are satisfied, then unused *get*s are discardable:

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

Proof.

$$
\begin{array}{ll}
& \mathbf{do}\,\{\, s \leftarrow \mathit{get}; m \,\} \\
= & [\![\ \text{(GS)}\ ]\!] \\
& \mathbf{do}\,\{\, s \leftarrow \mathit{get}; s' \leftarrow \mathit{get}; \mathit{set}\ s'; m \,\} \\
= & [\![\ \text{(GG)}\ ]\!] \\
& \mathbf{do}\,\{\, s \leftarrow \mathit{get}; \mathbf{let}\ s' = s; \mathit{set}\ s'; m \,\} \\
= & [\![\ \mathbf{let}\ ]\!] \\
& \mathbf{do}\,\{\, s \leftarrow \mathit{get}; \mathit{set}\ s; m \,\} \\
= & [\![\ \text{(GS)}\ ]\!] \\
& \mathbf{do}\,\{\, m \,\}
\end{array}
$$

□

Lemma 7. Suppose a, b are distinct variables not appearing in expression m. Then:

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

♢

Proof. For the first part:

$$
\begin{array}{ll}
& \mathbf{do}\,\{\, a \leftarrow \mathit{get}; b \leftarrow \mathit{lift}\ m; \mathit{return}\ (a, b) \,\}\ s \\
= & [\![\ \text{Definitions of bind, } \mathit{get}, \mathit{return}\ ]\!] \\
& \mathbf{do}\,\{\, (a, s') \leftarrow \mathit{return}\ (s, s); (b, s'') \leftarrow \mathbf{do}\,\{\, b' \leftarrow m; \mathit{return}\ (b', s') \,\}; \mathit{return}\ ((a, b), s'') \,\} \\
= & [\![\ \text{monad unit}\ ]\!] \\
& \mathbf{do}\,\{\, (b, s'') \leftarrow \mathbf{do}\,\{\, b' \leftarrow m; \mathit{return}\ (b', s) \,\}; \mathit{return}\ ((s, b), s'') \,\} \\
= & [\![\ \text{monad associativity}\ ]\!] \\
& \mathbf{do}\,\{\, b' \leftarrow m; (b, s'') \leftarrow \mathit{return}\ (b', s); \mathit{return}\ ((s, b), s'') \,\} \\
= & [\![\ \text{monad unit}\ ]\!] \\
& \mathbf{do}\,\{\, b' \leftarrow m; \mathit{return}\ ((s, b'), s) \,\} \\
= & [\![\ \text{reversing above steps}\ ]\!] \\
& \mathbf{do}\,\{\, (b, s') \leftarrow \mathbf{do}\,\{\, b' \leftarrow m; \mathit{return}\ (b', s) \,\}; (a, s'') \leftarrow \mathit{return}\ (s', s'); \mathit{return}\ ((a, b), s'') \,\} \\
= & [\![\ \text{definition}\ ]\!] \\
& \mathbf{do}\,\{\, b \leftarrow \mathit{lift}\ m; a \leftarrow \mathit{get}; \mathit{return}\ (a, b) \,\}\ s
\end{array}
$$

so the desired result holds by eta equivalence. For the second part:

$$
\begin{array}{ll}
& \mathbf{do}\,\{\, \mathit{set}\ a; b \leftarrow \mathit{lift}\ m; \mathit{return}\ b \,\}\ s \\
= & [\![\ \text{definitions of bind, } \mathit{lift}, \mathit{set}, \mathit{return}\ ]\!] \\
& \mathbf{do}\,\{\, (\_, s') \leftarrow \mathit{return}\ ((), a); (b, s'') \leftarrow \mathbf{do}\,\{\, b' \leftarrow m; \mathit{return}\ (b', s') \,\}; \mathit{return}\ (b, s'') \,\} \\
= & [\![\ \text{monad unit}\ ]\!] \\
& \mathbf{do}\,\{\, (b, s'') \leftarrow \mathbf{do}\,\{\, b' \leftarrow m; \mathit{return}\ (b', a) \,\}; \mathit{return}\ (b, s'') \,\} \\
= & [\![\ \text{monad associativity}\ ]\!] \\
& \mathbf{do}\,\{\, b' \leftarrow m; (b, s'') \leftarrow \mathit{return}\ (b', a); \mathit{return}\ (b, s'') \,\} \\
= & [\![\ \text{monad unit}\ ]\!] \\
& \mathbf{do}\,\{\, b' \leftarrow m; \mathit{return}\ (b', a) \,\}
\end{array}
$$

$$\begin{array}{ll} = & [\![\ \text{reversing above steps}\ ]\!] \\ & \textbf{do}\,\{\,(b,s') \leftarrow \textbf{do}\,\{\,b' \leftarrow m; \mathit{return}\;(b',s)\,\}; (\_,s'') \leftarrow \mathit{return}\;((),a); \mathit{return}\;(b,s'')\,\} \\ = & [\![\ \text{definition}\ ]\!] \\ & \textbf{do}\,\{\,b \leftarrow \mathit{lift}\;m; \mathit{set}\;a; \mathit{return}\;b\,\}\;s \end{array}$$

so again the result holds by eta-equivalence. □

Lemma 9. Given an arbitrary monad T, not assumed to be an instance of StateT, with operations $\mathit{get}_T :: T\;S$ and $\mathit{set}_T :: S \to T\;()$ for a type S, such that get_T and set_T satisfy the laws (GG), (GS), and (SG) of Definition 5, then there is a data refinement from T to $\mathit{StateT}\;S\;T$. ♢

Proof. Define the abstraction function abs from $\mathit{StateT}\;S\;T$ to T and the reification function conc in the opposite direction by

$$\begin{array}{ll} \mathit{abs}\;m & = \textbf{do}\,\{\,s \leftarrow \mathit{get}_T; (a,s') \leftarrow m\;s; \mathit{set}_T\;s'; \mathit{return}\;a\,\} \\ \mathit{conc}\;m & = \lambda s.\,\textbf{do}\,\{\,a \leftarrow m; s' \leftarrow \mathit{get}_T; \mathit{return}\;(a,s')\,\} \\ & = \textbf{do}\,\{\,a \leftarrow \mathit{lift}\;m; s' \leftarrow \mathit{lift}\;\mathit{get}_T; \mathit{set}\;s'; \mathit{return}\;a\,\} \end{array}$$

On account of the get_T and set_T operations that it provides, monad T is implicitly recording a state S; the idea of the data refinement is to track this state explicitly in monad $\mathit{StateT}\;S\;T$. We say that a computation m in $\mathit{StateT}\;S\;T$ is *synchronised* if on completion the inner implicit and the outer explicit S values agree:

$$\textbf{do}\,\{\,a \leftarrow m; s' \leftarrow \mathit{get}; \mathit{return}\;(a,s')\,\} = \textbf{do}\,\{\,a \leftarrow m; s'' \leftarrow \mathit{lift}\;\mathit{get}_T; \mathit{return}\;(a,s'')\,\}$$

or equivalently, as computations in T,

$$\textbf{do}\,\{\,(a,s') \leftarrow m\;s; \mathit{return}\;(a,s')\,\} = \textbf{do}\,\{\,(a,s') \leftarrow m\;s; s'' \leftarrow \mathit{get}_T; \mathit{return}\;(a,s'')\,\}$$

It is straightforward to check that $\mathit{return}\;a$ is synchronised, that bind preserves synchronisation, and that conc yields only synchronised computations.

We have to verify the three conditions of Definition 8. For the first, we have to show that conc distributes over $(\gg\!=)$; we have

$$\begin{array}{ll} & \mathit{conc}\;(m \gg\!= k) \\ = & [\![\ \mathit{conc}\ ]\!] \\ & \textbf{do}\,\{\,b \leftarrow \mathit{lift}\;(m \gg\!= k); s'' \leftarrow \mathit{lift}\;\mathit{get}_T; \mathit{set}\;s''; \mathit{return}\;b\,\} \\ = & [\![\ \mathit{lift}\text{ and bind}\ ]\!] \\ & \textbf{do}\,\{\,a \leftarrow \mathit{lift}\;m; b \leftarrow \mathit{lift}\;(k\;a); s'' \leftarrow \mathit{lift}\;\mathit{get}_T; \mathit{set}\;s''; \mathit{return}\;b\,\} \\ = & [\![\ \mathit{get}_T\text{ is discardable}\ ]\!] \\ & \textbf{do}\,\{\,a \leftarrow \mathit{lift}\;m; s' \leftarrow \mathit{lift}\;\mathit{get}_T; b \leftarrow \mathit{lift}\;(k\;a); s'' \leftarrow \mathit{lift}\;\mathit{get}_T; \mathit{set}\;s''; \mathit{return}\;b\,\} \\ = & [\![\ \text{(SS) for }\mathit{StateT}\ ]\!] \\ & \textbf{do}\,\{\,a \leftarrow \mathit{lift}\;m; s' \leftarrow \mathit{lift}\;\mathit{get}_T; b \leftarrow \mathit{lift}\;(k\;a); s'' \leftarrow \mathit{lift}\;\mathit{get}_T; \mathit{set}\;s'; \mathit{set}\;s''; \mathit{return}\;b\,\} \\ = & [\![\ \text{Lemma 7}\ ]\!] \end{array}$$

$$
\begin{array}{ll}
 & \mathbf{do}\,\{\,a \leftarrow \mathit{lift}\ m; s' \leftarrow \mathit{lift}\ \mathit{get}_T; \mathit{set}\ s'; b \leftarrow \mathit{lift}\ (k\ a); s'' \leftarrow \mathit{lift}\ \mathit{get}_T; \mathit{set}\ s''; \mathit{return}\ b\,\} \\
= & \llbracket\ \text{bind}\ \rrbracket \\
 & \mathbf{do}\,\{\,a \leftarrow \mathit{lift}\ m; s' \leftarrow \mathit{lift}\ \mathit{get}_T; \mathit{set}\ s'; \mathit{return}\ a\,\} \mathbin{>\!\!>\!\!=} \lambda a. \\
 & \mathbf{do}\,\{\,b \leftarrow \mathit{lift}\ (k\ a); s'' \leftarrow \mathit{lift}\ \mathit{get}_T; \mathit{set}\ s''; \mathit{return}\ b\,\} \\
= & \llbracket\ \mathit{conc}\ \rrbracket \\
 & \mathit{conc}\ m \mathbin{>\!\!>\!\!=} \lambda a.\ \mathit{conc}\ (k\ a) \\
= & \llbracket\ \text{eta contraction}\ \rrbracket \\
 & \mathit{conc}\ m \mathbin{>\!\!>\!\!=} (\mathit{conc} \cdot k)
\end{array}
$$

(Note that *conc* does not preserve *return*, and so *conc* is not a monad morphism: $\mathit{conc}\ (\mathit{return}\ a)$ not only returns a, it also synchronises the two copies of the state.) For the second, we have to show that $\mathit{abs} \cdot \mathit{conc}$ is the identity:

$$
\begin{array}{ll}
 & \mathit{abs}\ (\mathit{conc}\ m) \\
= & \llbracket\ \mathit{abs}\ \rrbracket \\
 & \mathbf{do}\,\{\,s \leftarrow \mathit{get}_T; (a, s') \leftarrow \mathit{conc}\ m\ s; \mathit{set}_T\ s'; \mathit{return}\ a\,\} \\
= & \llbracket\ \mathit{conc}\ \rrbracket \\
 & \mathbf{do}\,\{\,s \leftarrow \mathit{get}_T; a \leftarrow m; s' \leftarrow \mathit{get}_T; \mathit{set}_T\ s'; \mathit{return}\ a\,\} \\
= & \llbracket\ \text{(GS) for } T\ \rrbracket \\
 & \mathbf{do}\,\{\,s \leftarrow \mathit{get}_T; a \leftarrow m; \mathit{return}\ a\,\} \\
= & \llbracket\ \text{monad unit}\ \rrbracket \\
 & \mathbf{do}\,\{\,s \leftarrow \mathit{get}_T; m\,\} \\
= & \llbracket\ \text{unused } \mathit{get}_T \text{ is discardable}\ \rrbracket \\
 & \mathbf{do}\,\{\,m\,\}
\end{array}
$$

For the third, we have to show that post-composition with *abs* transforms the T operations into the corresponding $\mathit{StateT}\ S\ T$ operations. We do this by construction, defining:

$$
\begin{array}{l}
\mathit{sget} \quad\ = \mathit{conc}\ \mathit{get}_T \\
\mathit{sset}\ s' = \mathit{conc}\ (\mathit{set}_T\ s')
\end{array}
$$

Expanding and simplifying, we see that synchronised set writes to both copies of the state, and synchronised get reads from the inner copy, but overwrites the outer copy to ensure that it agrees:

$$
\begin{array}{l}
\mathit{sget} \quad\ = \mathbf{do}\,\{\,s' \leftarrow \mathit{lift}\ \mathit{get}_T; \mathit{set}\ s'; \mathit{return}\ s'\,\} \\
\mathit{sset}\ s' = \mathbf{do}\,\{\,\mathit{lift}\ (\mathit{set}_T\ s'); \mathit{set}\ s'\,\}
\end{array}
$$

□

Lemma 11. A very well-behaved lens $l :: \mathit{Lens}\ S\ V$ induces a monad morphism $\varphi :: \forall \alpha. \mathit{State}\ V\ \alpha \to \mathit{State}\ S\ \alpha$, defined by

$$
\varphi\ m = \mathbf{do}\,\{\,s \leftarrow \mathit{get}; \mathbf{let}\ (a, v') = m\ (l.\mathit{view}\ s); \mathit{set}\ (l.\mathit{update}\ s\ v'); \mathit{return}\ a\,\}
$$

◇

Proof. We have to show that φ is a monad morphism. We have:

$$\begin{array}{ll}
 & \varphi\ (\mathit{return}\ a) \\
= & [\![\ \varphi\ ]\!] \\
 & \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ (a', v') = (\mathit{return}\ a)\ (l.\mathit{view}\ s); \mathit{set}\ (l.\mathit{update}\ s\ v'); \mathit{return}\ a'\} \\
= & [\![\ \mathit{return}\ \text{for}\ \mathit{State}\ ]\!] \\
 & \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ (a', v') = (a, l.\mathit{view}\ s); \mathit{set}\ (l.\mathit{update}\ s\ v'); \mathit{return}\ a'\} \\
= & [\![\ \text{(VU)}\ ]\!] \\
 & \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ (a', v') = (a, l.\mathit{view}\ s); \mathit{set}\ s; \mathit{return}\ a'\} \\
= & [\![\ \mathbf{let}\ ]\!] \\
 & \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathit{set}\ s; \mathit{return}\ a\} \\
= & [\![\ \text{(GS)}\ ]\!] \\
 & \mathit{return}\ a
\end{array}$$

and:

$$\begin{array}{ll}
 & \varphi\ (m \mathbin{>\!\!>\!\!=} k) \\
= & [\![\ \varphi\ ]\!] \\
 & \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ (b, v'') = (m \mathbin{>\!\!>\!\!=} k)\ (l.\mathit{view}\ s); \mathit{set}\ (l.\mathit{update}\ s\ v''); \mathit{return}\ b\} \\
= & [\![\ \text{bind for}\ \mathit{State}\ ]\!] \\
 & \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ (a, v') = m\ (l.\mathit{view}\ s); \mathbf{let}\ (b, v'') = k\ a\ v'; \\
 & \qquad \mathit{set}\ (l.\mathit{update}\ s\ v''); \mathit{return}\ b\} \\
= & [\![\ \text{(SS)}\ ]\!] \\
 & \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ (a, v') = m\ (l.\mathit{view}\ s); \mathbf{let}\ (b, v'') = k\ a\ v'; \\
 & \qquad \mathit{set}\ (l.\mathit{update}\ s\ v'); \mathit{set}\ (l.\mathit{update}\ s\ v''); \mathit{return}\ b\} \\
= & [\![\ \text{rearranging}\ \mathbf{let}\text{s}\ ]\!] \\
 & \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ (a, v') = m\ (l.\mathit{view}\ s); \mathit{set}\ (l.\mathit{update}\ s\ v'); \\
 & \qquad \mathbf{let}\ s' = l.\mathit{update}\ s\ v'; \mathbf{let}\ (b, v'') = k\ a\ v'; \mathit{set}\ (l.\mathit{update}\ s\ v''); \mathit{return}\ b\} \\
= & [\![\ \text{(UV), (UU)}\ ]\!] \\
 & \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ (a, v') = m\ (l.\mathit{view}\ s); \mathit{set}\ (l.\mathit{update}\ s\ v'); \\
 & \qquad \mathbf{let}\ s' = l.\mathit{update}\ s\ v'; \mathbf{let}\ (b, v'') = k\ a\ (l.\mathit{view}\ s'); \mathit{set}\ (l.\mathit{update}\ s'\ v''); \mathit{return}\ b\} \\
= & [\![\ \text{(SG)}\ ]\!] \\
 & \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ (a, v') = m\ (l.\mathit{view}\ s); \mathit{set}\ (l.\mathit{update}\ s\ v'); \\
 & \qquad s' \leftarrow \mathit{get}; \mathbf{let}\ (b, v'') = k\ a\ (l.\mathit{view}\ s'); \mathit{set}\ (l.\mathit{update}\ s'\ v''); \mathit{return}\ b\} \\
= & [\![\ \text{bind}\ ]\!] \\
 & \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ (a, v') = m\ (l.\mathit{view}\ s); \mathit{set}\ (l.\mathit{update}\ s\ v'); \mathit{return}\ a\} \mathbin{>\!\!>\!\!=} \lambda a. \\
 & \mathbf{do}\ \{s' \leftarrow \mathit{get}; \mathbf{let}\ (b, v'') = (k\ a)\ (l.\mathit{view}\ s'); \mathit{set}\ (l.\mathit{update}\ s'\ v''); \mathit{return}\ b\} \\
= & [\![\ \varphi\ ]\!] \\
 & \varphi\ m \mathbin{>\!\!>\!\!=} \lambda a. \varphi\ (k\ a) \\
= & [\![\ \text{eta contraction}\ ]\!] \\
 & \varphi\ m \mathbin{>\!\!>\!\!=} (\varphi \cdot k)
\end{array}$$

□

C Proofs from Section 3

Lemma 17. *fstMLens* and *sndMLens* are very well-behaved; moreover, their *mupdate* operations commute in T. ♢

Proof. We consider only *fstMLens*, as *sndMLens* is symmetric. For (MVU), we have:

$$\begin{aligned}
&\quad \mathbf{do}\,\{\mathit{fstMLens}.\mathit{mupdate}\;(s_1,s_2)\;(\mathit{fstMLens}.\mathit{mview}\;(s_1,s_2))\}\\
=&\quad [\![\ \mathit{fstMLens}.\mathit{mview}\ ]\!]\\
&\quad \mathbf{do}\,\{\mathit{fstMLens}.\mathit{mupdate}\;(s_1,s_2)\;s_1\}\\
=&\quad [\![\ \mathit{fstMLens}.\mathit{mupdate}\ ]\!]\\
&\quad \mathbf{do}\,\{\mathit{return}\;(s_1,s_2)\}
\end{aligned}$$

For (MUV), we have:

$$\begin{aligned}
&\quad \mathbf{do}\,\{(s_1'',s_2'') \leftarrow \mathit{fstMLens}.\mathit{mupdate}\;(s_1,s_2)\;s_1';\mathit{return}\;((s_1'',s_2''),\mathit{fstMLens}.\mathit{mview}\;(s_1'',s_2''))\}\\
=&\quad [\![\ \mathit{fstMLens}.\mathit{mupdate}\ ]\!]\\
&\quad \mathbf{do}\,\{\mathbf{let}\;(s_1'',s_2'') = (s_1',s_2);\mathit{return}\;((s_1'',s_2''),\mathit{fstMLens}.\mathit{mview}\;(s_1'',s_2''))\}\\
=&\quad [\![\ \mathit{fstMLens}.\mathit{mview}\ ]\!]\\
&\quad \mathbf{do}\,\{\mathbf{let}\;(s_1'',s_2'') = (s_1',s_2);\mathit{return}\;((s_1'',s_2''),s_1'')\}\\
=&\quad [\![\ \text{substitute } \mathbf{let}\ ]\!]\\
&\quad \mathbf{do}\,\{\mathit{return}\;((s_1',s_2),s_1')\}\\
=&\quad [\![\ \mathit{fstMLens}.\mathit{mupdate}\ ]\!]\\
&\quad \mathbf{do}\,\{(s_1'',s_2'') \leftarrow \mathit{fstMLens}.\mathit{mupdate}\;(s_1,s_2)\;s_1';\mathit{return}\;((s_1'',s_2''),s_1')\}
\end{aligned}$$

And for (MUU), we have:

$$\begin{aligned}
&\quad \mathbf{do}\,\{(s_1'',s_2'') \leftarrow \mathit{fstMLens}.\mathit{mupdate}\;(s_1,s_2)\;s_1';\mathit{fstMLens}.\mathit{mupdate}\;(s_1'',s_2'')\;s_1'\}\\
=&\quad [\![\ \mathit{fstMLens}.\mathit{mupdate}\ ]\!]\\
&\quad \mathbf{do}\,\{\mathbf{let}\;(s_1'',s_2'') = (s_1',s_2);\mathit{return}\;(s_1',s_2'')\;s_1'\}\\
=&\quad [\![\ \text{substitute } \mathbf{let}\ ]\!]\\
&\quad \mathbf{do}\,\{\mathit{return}\;(s_1',s_2')\;s_1'\}\\
=&\quad [\![\ \mathit{fstMLens}.\mathit{mupdate}\ ]\!]\\
&\quad \mathbf{do}\,\{\mathit{fstMLens}.\mathit{mupdate}\;(s_1,s_2)\;s_1'\}
\end{aligned}$$

Finally, the *mupdate* operations clearly commute in T, because they are pure. □

Lemma 52. The monadic asymmetric lenses *fstMLens*, *sndMLens* from Section 3.3 are very well-behaved. ♢

Proof. We prove the lemma for *fstMLens* only; *sndMLens* is dual. For (MVU), we have:

$$\begin{aligned}
&\quad \mathbf{do}\,\{\mathit{fstMLens}.\mathit{mupdate}\;(a,b)\;(\mathit{fstMLens}.\mathit{mview}\;(a,b))\}\\
=&\quad [\![\ \text{definition of } \mathit{fstMLens}.\mathit{mview}\ ]\!]\\
&\quad \mathbf{do}\,\{\mathit{fstMLens}.\mathit{mupdate}\;(a,b)\;a\}
\end{aligned}$$

$$
\begin{array}{ll}
= & \llbracket \text{ definition of } \mathit{fstMLens.mupdate} \ \rrbracket \\
 & \mathbf{do}\ \{\mathit{return}\ (a, b)\}
\end{array}
$$

For (MUV), we have:

$$
\begin{array}{ll}
 & \mathbf{do}\ \{s' \leftarrow \mathit{fstMLens.mupdate}\ (a, b)\ a'; \mathit{return}\ (s', \mathit{fstMLens.mview}\ s')\} \\
= & \llbracket \text{ definition of } \mathit{fstMLens.mupdate} \ \rrbracket \\
 & \mathbf{do}\ \{\mathbf{let}\ s' = (a', b); \mathit{return}\ (s', \mathit{fstMLens.mview}\ s')\} \\
= & \llbracket \text{ definition of } \mathit{fstMLens.mview} \ \rrbracket \\
 & \mathbf{do}\ \{\mathbf{let}\ s' = (a', b); \mathit{return}\ (s', a')\} \\
= & \llbracket \text{ definition of } \mathit{fstMLens.mupdate} \ \rrbracket \\
 & \mathbf{do}\ \{s' \leftarrow \mathit{fstMLens.mupdate}\ (a, b)\ a'; \mathit{return}\ (s', a')\}
\end{array}
$$

Finally, for (MUU) we have:

$$
\begin{array}{ll}
 & \mathbf{do}\ \{s' \leftarrow \mathit{fstMLens.mupdate}\ (a, b)\ a'; \mathit{fstMLens.mupdate}\ s'\ a''\} \\
= & \llbracket \text{ definition of } \mathit{fstMLens.mupdate} \ \rrbracket \\
 & \mathbf{do}\ \{\mathbf{let}\ s' = (a', b); \mathit{fstMLens.mupdate}\ s'\ a''\} \\
= & \llbracket \text{ definition of } \mathit{fstMLens.mupdate} \ \rrbracket \\
 & \mathbf{do}\ \{\mathbf{let}\ s' = (a', b); \mathit{return}\ (a'', b)\} \\
= & \llbracket \text{ definition of } \mathit{fstMLens.mupdate} \ \rrbracket \\
 & \mathbf{do}\ \{\mathit{fstMLens.mupdate}\ (a, b)\ a''\}
\end{array}
$$

□

D Proofs from Section 4

Lemma 19. If $l :: \mathit{MLens}\ T\ S\ V$ is very well-behaved and $l.\mathit{mupdate}\ s\ v$ commutes in T for any s, v, then $\vartheta\ l$ is a monad morphism. ♢

Proof. We first show that $\vartheta\ l$ preserves *return*s:

$$
\begin{array}{ll}
 & \vartheta\ l\ (\mathit{return}\ x) \\
= & \llbracket\ \vartheta\ \rrbracket \\
 & \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ v = l.\mathit{mview}\ s; (a, v') \leftarrow \mathit{lift}\ (\mathit{return}\ x\ v); \\
 & \qquad s' \leftarrow \mathit{lift}\ (l.\mathit{mupdate}\ s\ v'); \mathit{set}\ s'; \mathit{return}\ a\} \\
= & \llbracket\ \mathit{return} \text{ for } \mathit{StateT}\ \rrbracket \\
 & \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ v = l.\mathit{mview}\ s; \mathbf{let}\ (a, v') = (x, v); \\
 & \qquad s' \leftarrow \mathit{lift}\ (l.\mathit{mupdate}\ s\ v'); \mathit{set}\ s'; \mathit{return}\ a\} \\
= & \llbracket\ \text{(MUV)}\ \rrbracket \\
 & \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ v = l.\mathit{mview}\ s; \mathbf{let}\ (a, v') = (x, v); \mathbf{let}\ s' = s; \mathit{set}\ s'; \mathit{return}\ a\} \\
= & \llbracket\ \text{(GS) for } \mathit{StateT}\ \rrbracket \\
 & \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ v = l.\mathit{mview}\ s; \mathit{return}\ x\} \\
= & \llbracket\ \mathit{get} \text{ is discardable}\ \rrbracket \\
 & \mathit{return}\ x
\end{array}
$$

Now we show that $\vartheta\ l$ respects sequential composition:

$$
\begin{array}{ll}
& \vartheta\ l\ (\mathbf{do}\ \{a \leftarrow m; k\ a\}) \\
= & \llbracket\ \vartheta\ \rrbracket \\
& \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ v = l.\mathit{mview}\ s; (b, v''') \leftarrow \mathit{lift}\ (\mathbf{do}\ \{a \leftarrow m; k\ a\}\ v); \\
& \qquad s''' \leftarrow \mathit{lift}\ (l.\mathit{mupdate}\ s\ v'''); \mathit{set}\ s'''; \mathit{return}\ b\} \\
= & \llbracket\ \mathit{lift}; \text{bind for } \mathit{StateT}\ \rrbracket \\
& \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ v = l.\mathit{mview}\ s; (a, v') \leftarrow \mathit{lift}\ (m\ v); (b, v''') \leftarrow \mathit{lift}\ (k\ a\ v'); \\
& \qquad s''' \leftarrow \mathit{lift}\ (l.\mathit{mupdate}\ s\ v'''); \mathit{set}\ s'''; \mathit{return}\ b\} \\
= & \llbracket\ \text{(MUU)}\ \rrbracket \\
& \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ v = l.\mathit{mview}\ s; (a, v') \leftarrow \mathit{lift}\ (m\ v); (b, v''') \leftarrow \mathit{lift}\ (k\ a\ v'); \\
& \qquad s' \leftarrow \mathit{lift}\ (l.\mathit{mupdate}\ s\ v'); s''' \leftarrow \mathit{lift}\ (l.\mathit{mupdate}\ s'\ v'''); \mathit{set}\ s'''; \mathit{return}\ b\} \\
= & \llbracket\ l.\mathit{mupdate}\ s\ v' \text{ commutes in } T\ \rrbracket \\
& \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ v = l.\mathit{mview}\ s; (a, v') \leftarrow \mathit{lift}\ (m\ v); s' \leftarrow \mathit{lift}\ (l.\mathit{mupdate}\ s\ v'); \\
& \qquad (b, v''') \leftarrow \mathit{lift}\ (k\ a\ v'); s''' \leftarrow \mathit{lift}\ (l.\mathit{mupdate}\ s'\ v'''); \mathit{set}\ s'''; \mathit{return}\ b\} \\
= & \llbracket\ \text{(SS) for } \mathit{StateT}\ \rrbracket \\
& \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ v = l.\mathit{mview}\ s; (a, v') \leftarrow \mathit{lift}\ (m\ v); s' \leftarrow \mathit{lift}\ (l.\mathit{mupdate}\ s\ v'); \\
& \qquad (b, v''') \leftarrow \mathit{lift}\ (k\ a\ v'); s''' \leftarrow \mathit{lift}\ (l.\mathit{mupdate}\ s'\ v'''); \mathit{set}\ s'; \mathit{set}\ s'''; \mathit{return}\ b\} \\
= & \llbracket\ \text{Lemma 7}\ \rrbracket \\
& \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ v = l.\mathit{mview}\ s; (a, v') \leftarrow \mathit{lift}\ (m\ v); s' \leftarrow \mathit{lift}\ (l.\mathit{mupdate}\ s\ v'); \mathit{set}\ s'; \\
& \qquad (b, v''') \leftarrow \mathit{lift}\ (k\ a\ v'); s''' \leftarrow \mathit{lift}\ (l.\mathit{mupdate}\ s'\ v'''); \mathit{set}\ s'''; \mathit{return}\ b\} \\
= & \llbracket\ \text{(MUV); (SG) for } \mathit{StateT}\ \rrbracket \\
& \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{let}\ v = l.\mathit{mview}\ s; (a, v') \leftarrow \mathit{lift}\ (m\ v); s' \leftarrow \mathit{lift}\ (l.\mathit{mupdate}\ s\ v'); \mathit{set}\ s'; \\
& \qquad s'' \leftarrow \mathit{get}; \mathbf{let}\ v'' = l.\mathit{mview}\ s'; (b, v''') \leftarrow \mathit{lift}\ (k\ a\ v''); s''' \leftarrow \mathit{lift}\ (l.\mathit{mupdate}\ s''\ v'''); \mathit{set}\ s'''; \\
& \qquad \mathit{return}\ b\} \\
= & \llbracket\ \vartheta\ \rrbracket \\
& \mathbf{do}\ \{a \leftarrow \vartheta\ l\ m; \vartheta\ k\ (k\ a)\}
\end{array}
$$

□

Lemma 53. Simplifying definitions, we have

$$
\begin{array}{ll}
\vartheta\ \mathit{fstMLens}\ m & = \mathbf{do}\ \{(s_1, s_2) \leftarrow \mathit{get}; (c, s_1') \leftarrow \mathit{lift}\ (m\ s_1); \mathit{set}\ (s_1', s_2); \mathit{return}\ c\} \\
\vartheta\ \mathit{sndMLens}\ m & = \mathbf{do}\ \{(s_1, s_2) \leftarrow \mathit{get}; (a, s_2') \leftarrow \mathit{lift}\ (m\ s_2); \mathit{set}\ (s_1, s_2'); \mathit{return}\ a\}
\end{array}
$$

This will be convenient in what follows. ♢

Lemma 54. For arbitrary $f :: \sigma_1 \to \alpha$ and $m :: \mathit{StateT}\ \sigma_2\ \tau\ \beta$, the liftings *left* (*gets f*) and *right m* commute; and symmetrically for *right* (*gets f*) and *left m*. (In fact, this holds for any T-pure computation, not just *gets f*; but we do not need the more general result.) ♢

Proof. We have:

$$
\begin{array}{ll}
& \mathbf{do}\ \{a \leftarrow \mathit{left}\ (\mathit{gets}\ f); b \leftarrow \mathit{right}\ m; \mathit{return}\ (a, b)\} \\
= & \llbracket\ \text{Lemma 53, } \mathit{gets}\ \rrbracket \\
& \mathbf{do}\ \{(s_1, s_2) \leftarrow \mathit{get}; \mathbf{let}\ a = f\ s_1; (b, s_2') \leftarrow \mathit{lift}\ (m\ s_2); \mathit{set}\ (s_1, s_2'); \mathit{return}\ (a, b)\}
\end{array}
$$

$$\begin{array}{ll}
= & [\![\ \text{move } \mathbf{let}\ ]\!] \\
& \mathbf{do}\,\{(s_1, s_2) \leftarrow \mathit{get}; (b, s_2') \leftarrow \mathit{lift}\ (m\ s_2); \mathit{set}\ (s_1, s_2'); \mathbf{let}\ a = f\ s_1; \mathit{return}\ (a, b)\} \\
= & [\![\ \text{(SG) for } \mathit{StateT}\ ]\!] \\
& \mathbf{do}\,\{(s_1, s_2) \leftarrow \mathit{get}; (b, s_2') \leftarrow \mathit{lift}\ (m\ s_2); \mathit{set}\ (s_1, s_2'); (s_1'', s_2'') \leftarrow \mathit{get}; \mathbf{let}\ a = f\ s_1''; \mathit{return}\ (a, b)\} \\
= & [\![\ \text{Lemma 53}\ ]\!] \\
& \mathbf{do}\,\{b \leftarrow \mathit{right}\ m; a \leftarrow \mathit{left}\ (\mathit{gets}\ f); \mathit{return}\ (a, b)\}
\end{array}$$

The symmetric property of course has a symmetric proof too. □

Theorem 22 (transparent composition). Given transparent well-behaved bx

$$\begin{array}{l}
\mathit{bx}_1 :: \mathit{StateTBX}\ S_1\ T\ A\ B \\
\mathit{bx}_2 :: \mathit{StateTBX}\ S_2\ T\ B\ C
\end{array}$$

their composition $\mathit{bx}_1 \fatsemi \mathit{bx}_2 :: \mathit{StateTBX}\ (S_1 \bowtie S_2)\ T\ A\ C$ is also transparent and well-behaved. ♢

Proof. We first have to check that the composition does indeed operate only on the state space $S_1 \bowtie S_2$, by verifying that set_L and set_R maintain this invariant. For set_L, we have:

$$\begin{array}{ll}
& \mathbf{do}\,\{\mathit{set}_L\ a'; \mathit{left}\ (\mathit{bx}_1.\mathit{get}_R)\} \\
= & [\![\ \mathit{set}_L; \mathit{return}\ ]\!] \\
& \mathbf{do}\,\{\mathit{left}\ (\mathit{bx}_1.\mathit{set}_L\ a'); b' \leftarrow \mathit{left}\ (\mathit{bx}_1.\mathit{get}_R); \mathit{right}\ (\mathit{bx}_2.\mathit{set}_L\ b'); b'' \leftarrow \mathit{left}\ (\mathit{bx}_1.\mathit{get}_R); \mathit{return}\ b''\} \\
= & [\![\ \mathit{bx}_1.\mathit{get}_R = \mathit{gets}\ (\mathit{bx}_1.\mathit{read}_R), \text{ and Lemma 54}\ ]\!] \\
& \mathbf{do}\,\{\mathit{left}\ (\mathit{bx}_1.\mathit{set}_L\ a'); b' \leftarrow \mathit{left}\ (\mathit{bx}_1.\mathit{get}_R); b'' \leftarrow \mathit{left}\ (\mathit{bx}_1.\mathit{get}_R); \mathit{right}\ (\mathit{bx}_2.\mathit{set}_L\ b'); \mathit{return}\ b''\} \\
= & [\![\ \mathit{left} \text{ is a monad morphism, and } (\mathrm{G_R G_R}) \text{ for } \mathit{bx}_1\ ]\!] \\
& \mathbf{do}\,\{\mathit{left}\ (\mathit{bx}_1.\mathit{set}_L\ a'); b' \leftarrow \mathit{left}\ (\mathit{bx}_1.\mathit{get}_R); \mathbf{let}\ b'' = b'; \mathit{right}\ (\mathit{bx}_2.\mathit{set}_L\ b'); \mathit{return}\ b''\} \\
= & [\![\ \text{move } \mathbf{let}\ ]\!] \\
& \mathbf{do}\,\{\mathit{left}\ (\mathit{bx}_1.\mathit{set}_L\ a'); b' \leftarrow \mathit{left}\ (\mathit{bx}_1.\mathit{get}_R); \mathit{right}\ (\mathit{bx}_2.\mathit{set}_L\ b'); \mathbf{let}\ b'' = b'; \mathit{return}\ b''\} \\
= & [\![\ \mathit{right} \text{ is a monad morphism, and } (\mathrm{S_L G_L}) \text{ for } \mathit{bx}_2\ ]\!] \\
& \mathbf{do}\,\{\mathit{left}\ (\mathit{bx}_1.\mathit{set}_L\ a'); b' \leftarrow \mathit{left}\ (\mathit{bx}_1.\mathit{get}_R); \mathit{right}\ (\mathit{bx}_2.\mathit{set}_L\ b'); b'' \leftarrow \mathit{right}\ (\mathit{bx}_2.\mathit{get}_L); \mathit{return}\ b''\} \\
= & [\![\ \mathit{set}_L\ ]\!] \\
& \mathbf{do}\,\{\mathit{set}_L\ a'; \mathit{right}\ (\mathit{bx}_2.\mathit{get}_L)\}
\end{array}$$

Of course, set_R is symmetric. Note that get_L (and symmetrically, get_R) are T-pure queries, so do not affect the state:

$$\begin{array}{ll}
& \mathit{get}_L \\
= & [\![\ \mathit{get}_L\ ]\!] \\
& \mathit{left}\ (\mathit{bx}_1.\mathit{get}_L) \\
= & [\![\ \text{Lemma 53}\ ]\!] \\
& \mathbf{do}\,\{(s_1, s_2) \leftarrow \mathit{get}; (c, s_1') \leftarrow \mathit{lift}\ (\mathit{bx}_1.\mathit{get}_L\ s_1); \mathit{set}\ (s_1', s_2); \mathit{return}\ c\} \\
= & [\![\ \mathit{bx}_1 \text{ is transparent}\ ]\!] \\
& \mathbf{do}\,\{(s_1, s_2) \leftarrow \mathit{get}; \mathbf{let}\ (c, s_1') = (\mathit{bx}_1.\mathit{read}_L\ s_1, s_1); \mathit{set}\ (s_1', s_2); \mathit{return}\ c\} \\
= & [\![\ \text{(GS) for } \mathit{StateT}\ ]\!]
\end{array}$$

$$\begin{array}{ll}
& \mathbf{do}\,\{\,(s_1, s_2) \leftarrow \mathit{get}; \mathbf{let}\ c = \mathit{bx}_1.\mathit{read}_L\ s_1; \mathit{return}\ c\,\} \\
= & \llbracket\ \mathit{gets}\ \rrbracket \\
& \mathit{gets}\ (\mathit{read}_L \cdot \mathit{fst})
\end{array}$$

So the composition is transparent, and hence we get the laws $(\mathrm{G_LG_L})$, $(\mathrm{G_RG_R})$, and $(\mathrm{G_LG_R})$ for free: we need only check the laws involving sets. For $(\mathrm{S_LG_L})$, we have:

$$\begin{array}{ll}
& \mathbf{do}\,\{\,\mathit{set}_L\ a'; \mathit{get}_L\,\} \\
= & \llbracket\ \mathit{set}_L,\ \mathit{get}_L\ \rrbracket \\
& \mathbf{do}\,\{\,\mathit{left}\ (\mathit{bx}_1.\mathit{set}_L\ a'); b' \leftarrow \mathit{left}\ (\mathit{bx}_1.\mathit{get}_R); \mathit{right}\ (\mathit{bx}_2.\mathit{set}_L\ b'); \mathit{left}\ (\mathit{bx}_1.\mathit{get}_L)\,\} \\
= & \llbracket\ \text{Lemma 54}\ \rrbracket \\
& \mathbf{do}\,\{\,\mathit{left}\ (\mathit{bx}_1.\mathit{set}_L\ a'); b' \leftarrow \mathit{left}\ (\mathit{bx}_1.\mathit{get}_R); a'' \leftarrow \mathit{left}\ (\mathit{bx}_1.\mathit{get}_L); \mathit{right}\ (\mathit{bx}_2.\mathit{set}_L\ b'); \mathit{return}\ a''\,\} \\
= & \llbracket\ \mathit{left}\ \text{is a monad morphism; } (\mathrm{G_LG_R})\ \text{for}\ \mathit{bx}_1\ \rrbracket \\
& \mathbf{do}\,\{\,\mathit{left}\ (\mathit{bx}_1.\mathit{set}_L\ a'); a'' \leftarrow \mathit{left}\ (\mathit{bx}_1.\mathit{get}_L); b' \leftarrow \mathit{left}\ (\mathit{bx}_1.\mathit{get}_R); \mathit{right}\ (\mathit{bx}_2.\mathit{set}_L\ b'); \mathit{return}\ a''\,\} \\
= & \llbracket\ \mathit{left}\ \text{is a monad morphism; } (\mathrm{S_LG_L})\ \text{for}\ \mathit{bx}_1\ \rrbracket \\
& \mathbf{do}\,\{\,\mathit{left}\ (\mathit{bx}_1.\mathit{set}_L\ a'); \mathbf{let}\ a'' = a'; b' \leftarrow \mathit{left}\ (\mathit{bx}_1.\mathit{get}_R); \mathit{right}\ (\mathit{bx}_2.\mathit{set}_L\ b'); \mathit{return}\ a''\,\} \\
= & \llbracket\ \text{move}\ \mathbf{let};\ \mathit{set}_L\ \rrbracket \\
& \mathbf{do}\,\{\,\mathit{set}_L\ a'; \mathit{return}\ a'\,\}
\end{array}$$

And for $(\mathrm{G_LS_L})$, using the fact that the initial state is in $S_1 \bowtie S_2$:

$$\begin{array}{ll}
& \mathbf{do}\,\{\,a \leftarrow \mathit{get}_L; \mathit{set}_L\ a\,\} \\
= & \llbracket\ \mathit{get}_L,\ \mathit{set}_L\ \rrbracket \\
& \mathbf{do}\,\{\,a \leftarrow \mathit{left}\ (\mathit{bx}_1.\mathit{get}_L); \mathit{left}\ (\mathit{bx}_1.\mathit{set}_L\ a); b' \leftarrow \mathit{left}\ (\mathit{bx}_1.\mathit{get}_R); \mathit{right}\ (\mathit{bx}_2.\mathit{set}_L\ b')\,\} \\
= & \llbracket\ \mathit{left}\ \text{is a monad morphism; } (\mathrm{G_LS_L})\ \text{for}\ \mathit{bx}_1\ \rrbracket \\
& \mathbf{do}\,\{\,b' \leftarrow \mathit{left}\ (\mathit{bx}_1.\mathit{get}_R); \mathit{right}\ (\mathit{bx}_2.\mathit{set}_L\ b')\,\} \\
= & \llbracket\ \text{initial state is consistent, so}\ \mathit{left}\ (\mathit{bx}_1.\mathit{get}_R) = \mathit{right}\ (\mathit{bx}_2.\mathit{get}_L)\ \rrbracket \\
& \mathbf{do}\,\{\,b' \leftarrow \mathit{right}\ (\mathit{bx}_2.\mathit{get}_L); \mathit{right}\ (\mathit{bx}_2.\mathit{set}_L\ b')\,\} \\
= & \llbracket\ \mathit{right}\ \text{is a monad morphism; } (\mathrm{G_LS_L})\ \text{for}\ \mathit{bx}_2\ \rrbracket \\
& \mathit{return}\ ()
\end{array}$$

And of course, $(\mathrm{S_RG_R})$ and $(\mathrm{G_RS_R})$ are symmetric. □

E Proofs from Section 5

Proposition 55. The *identity* bx (Definition 24) is well-behaved, overwritable, and transparent. ♢

Proof. The transparency of *identity* is obvious from its definition ($\mathit{get} = \mathit{gets}\ \mathit{id}$). The well-behavedness and overwritability laws are all immediate from the laws (GG), (GS), (SG), and (SS) of the monad $\mathit{StateT}\ S\ T$. □

Lemma 25. If $h :: S_1 \to S_2$ is invertible, then $\iota\ h$ is a monad isomorphism from $\mathit{StateT}\ S_1\ T$ to $\mathit{StateT}\ S_2\ T$. ♢

Proof. The fact that $\iota\ h$ is a monad morphism follows from the fact that any isomorphism determines a very well-behaved T-lens whose updates commute in T, by Lemma 19. It is straightforward to verify that if h and $inv\ h$ are inverses then so are $\iota\ h$ and $\iota^{-1}\ h$. □

Lemma 56. For any well-behaved bx, we have

$$bx \equiv identity \fatsemi bx \quad \text{and} \quad bx \equiv bx \fatsemi identity \qquad \diamondsuit$$

Proof. For the LHS of (Identity), consider

$$\begin{array}{ll} identity & :: StateTBX\ \ T\ \ A\ \ A\ \ A \\ bx & :: StateTBX\ \ T\ \ S\ \ A\ \ B \end{array}$$

so $identity \fatsemi bx :: StateTBX\ \ T\ (A \bowtie S)\ A\ B$, where

$$\begin{aligned} A \bowtie S &= \{\, (a, s) \mid eval\ (identity.get_R)\ a = eval\ (bx.get_L)\ s \,\} \\ &= \{\, (bx.read_L\ s, s) \mid s \in S \,\} \end{aligned}$$

To define an isomorphism between $StateT\ S\ T$ and $StateT\ (A \bowtie S)\ T$, we need to define an isomorphism $f : S \to (A \bowtie S)$. This is straightforward: the two directions are

$$f\ s = (bx.read_L\ s, s) \qquad f^{-1} = snd$$

We just need to verify compatibility of the isomorphism $StateT\ S\ T \cong StateT\ (A \bowtie S)\ T$ that results with the operations of $identity \fatsemi bx$ and bx, that is:

$$\begin{array}{l} (\iota\ h)\ bx.get_L = (id \fatsemi bx).get_L \\ (\iota\ h)\ bx.get_R = (id \fatsemi bx).get_R \\ (\iota\ h)\ bx.set_L\ a = (id \fatsemi bx).set_L\ a \\ (\iota\ h)\ bx.set_R\ b = (id \fatsemi bx).set_R\ b \end{array}$$

We illustrate the get_L, set_L cases, as they are more interesting.

$$\begin{array}{l} (\iota\ h)\ bx.get_L \\ \quad = \quad [\![\ \text{definition of } \iota\ h \text{ (using } f \text{ and } f^{-1})\ ]\!] \\ \mathbf{do}\ \{\, (a, s) \leftarrow get; \\ \qquad (a', s') \leftarrow lift\ (bx.get_L\ (snd\ (a, s))); \\ \qquad set\ (bx.read_L\ s', s'); \\ \qquad return\ a' \,\} \\ \quad = \quad [\![\ \text{simplifying } snd;\ bx \text{ is transparent}\ ]\!] \\ \mathbf{do}\ \{\, (a, s) \leftarrow get; \\ \qquad (a', s') \leftarrow lift\ (return\ (bx.read_L\ s, s)); \\ \qquad set\ (bx.read_L\ s', s'); \\ \qquad return\ a' \,\} \\ \quad = \quad [\![\ lift \text{ monad morphism}\ ]\!] \end{array}$$

$$\begin{array}{l}
\mathbf{do}\ \{(a,s) \leftarrow \mathit{get};\\
\quad (a',s') \leftarrow \mathit{return}\ (\mathit{bx}.\mathit{read}_L\ s, s);\\
\quad \mathit{set}\ (\mathit{bx}.\mathit{read}_L\ s', s');\\
\quad \mathit{return}\ a'\}\\
\quad = \quad \llbracket \text{ inlining } a' \text{ and } s' \quad \rrbracket\\
\mathbf{do}\ \{(a,s) \leftarrow \mathit{get};\\
\quad \mathit{set}\ (\mathit{bx}.\mathit{read}_L\ s, s);\\
\quad \mathit{return}\ \mathit{bx}.\mathit{read}_L\ s\}\\
\quad = \quad \llbracket\ (a,s) :: (A \bowtie S) \text{ implies } \mathit{bx}.\mathit{read}_L\ s = a \quad \rrbracket\\
\mathbf{do}\ \{(a,s) \leftarrow \mathit{get};\\
\quad \mathit{set}\ (a,s);\\
\quad \mathit{return}\ a\}\\
\quad = \quad \llbracket \text{ (GG) and then (GS) for } \mathit{StateT}\ (A \bowtie S)\ T \quad \rrbracket\\
\mathbf{do}\ \{(a,s) \leftarrow \mathit{get};\\
\quad \mathit{return}\ a\}\\
\quad = \quad \llbracket \text{ introduce trivial binding (where } \mathit{get} :: \mathit{StateT}\ A\ A) \quad \rrbracket\\
\mathbf{do}\ \{(a,s) \leftarrow \mathit{get}; (a', \_) \leftarrow \mathit{lift}\ (\mathit{get}\ a);\\
\quad \mathit{return}\ a'\}\\
\quad = \quad \llbracket \text{ definition of } \mathit{id}.\mathit{get}_L \quad \rrbracket\\
\mathbf{do}\ \{(a,s) \leftarrow \mathit{get}; (a', \_) \leftarrow \mathit{lift}\ (\mathit{id}.\mathit{get}_L\ a);\\
\quad \mathit{return}\ a\}\\
\quad = \quad \llbracket \text{ Form of } \mathit{get}_L \text{ preceding Remark 23} \quad \rrbracket\\
(\mathit{id} \fatsemi \mathit{bx}).\mathit{get}_L
\end{array}$$

Here is the proof for set_L:

$$\begin{array}{l}
(\mathit{id} \fatsemi \mathit{bx}).\mathit{set}_L\ a'\\
\quad = \quad \llbracket \text{ Form of } \mathit{set}_L \text{ preceding Remark 23} \quad \rrbracket\\
\mathbf{do}\ \{(a,s) \leftarrow \mathit{get}; ((), a') \leftarrow \mathit{lift}\ (\mathit{id}.\mathit{set}_L\ a'\ a);\\
\qquad (b, \_) \leftarrow \mathit{lift}\ (\mathit{id}.\mathit{get}_R\ a');\\
\qquad ((), s') \leftarrow \mathit{lift}\ (\mathit{bx}.\mathit{set}_L\ b\ s); \mathit{set}\ (a', s'); \mathit{return}\ ()\}\\
\quad = \quad \llbracket \text{ definition of } \mathit{set}_L, \mathit{get}_R \text{ for } \mathit{id} \quad \rrbracket\\
\mathbf{do}\ \{(a,s) \leftarrow \mathit{get}; ((), a') \leftarrow \mathit{lift}\ (\mathit{set}\ a'\ a);\\
\qquad (b, \_) \leftarrow \mathit{lift}\ (\mathit{get}\ a');\\
\qquad ((), s') \leftarrow \mathit{lift}\ (\mathit{bx}.\mathit{set}_L\ b\ s); \mathit{set}\ (a', s'); \mathit{return}\ ()\}\\
\quad = \quad \llbracket \text{ definitions of } \mathit{set} \text{ and } \mathit{get} \quad \rrbracket\\
\mathbf{do}\ \{(a,s) \leftarrow \mathit{get}; ((), a') \leftarrow \mathit{lift}\ (\mathit{return}\ ((), a'));\\
\qquad (b, \_) \leftarrow \mathit{lift}\ (\mathit{return}\ (a', a'));\\
\qquad ((), s') \leftarrow \mathit{lift}\ (\mathit{bx}.\mathit{set}_L\ b\ s); \mathit{set}\ (a', s')\}\\
\quad = \quad \llbracket\ \mathit{lift}\ (\mathit{return}\ x) = \mathit{return}\ x; \text{ inline resulting } \textbf{let}\text{s} \quad \rrbracket\\
\mathbf{do}\ \{(a,s) \leftarrow \mathit{get};\\
\quad ((), s') \leftarrow \mathit{lift}\ (\mathit{bx}.\mathit{set}_L\ a'\ s);\\
\quad \mathit{set}\ (a', s')\}\\
\quad = \quad \llbracket \text{ introducing binding } (\mathit{return}\ a' :: \mathit{StateT}\ S\ T\ A) \quad \rrbracket
\end{array}$$

$$
\begin{array}{l}
\quad \mathbf{do}\ \{(a,s) \leftarrow \mathit{get};\\
\qquad ((),s') \leftarrow \mathit{lift}\ (\mathit{bx}.\mathit{set}_L\ a'\ s);\\
\qquad (a'',s'') \leftarrow \mathit{lift}\ (\mathit{return}\ a'\ s');\\
\qquad \mathit{set}\ (a'',s'')\}\\
= \quad [\![\ \mathit{lift}\ \text{monad morphism}\ ]\!]\\
\quad \mathbf{do}\ \{(a,s) \leftarrow \mathit{get};\\
\qquad (a'',s'') \leftarrow \mathit{lift}\ (\mathbf{do}\ \{\mathit{bx}.\mathit{set}_L\ a'; \mathit{return}\ a'\}\ s);\\
\qquad \mathit{set}\ (a'',s'')\}\\
= \quad [\![\ (\mathrm{G_L S_L})\ \text{for}\ \mathit{bx}\ ]\!]\\
\quad \mathbf{do}\ \{(a,s) \leftarrow \mathit{get};\\
\qquad (a'',s'') \leftarrow \mathit{lift}\ (\mathbf{do}\ \{\mathit{bx}.\mathit{set}_L\ a'; \mathit{bx}.\mathit{get}_L\}\ s);\\
\qquad \mathit{set}\ (a'',s'')\}\\
= \quad [\![\ \mathit{lift}\ \text{monad morphism}\ ]\!]\\
\quad \mathbf{do}\ \{(a,s) \leftarrow \mathit{get};\\
\qquad ((),s') \leftarrow \mathit{lift}\ (\mathit{bx}.\mathit{set}_L\ a'\ s);\\
\qquad (a'',s'') \leftarrow \mathit{lift}\ (\mathit{bx}.\mathit{get}_L\ s');\\
\qquad \mathit{set}\ (a'',s'')\}\\
= \quad [\![\ \mathit{bx}\ \text{is transparent}\ ]\!]\\
\quad \mathbf{do}\ \{(a,s) \leftarrow \mathit{get};\\
\qquad ((),s') \leftarrow \mathit{lift}\ (\mathit{bx}.\mathit{set}_L\ a'\ s);\\
\qquad (a'',s'') \leftarrow \mathit{lift}\ (\mathit{return}\ (\mathit{bx}.\mathit{read}_L\ s', s'));\\
\qquad \mathit{set}\ (a'',s'')\}\\
= \quad [\![\ \mathit{lift} \cdot \mathit{return} = \mathit{return};\ \text{inlining}\ a''\ \text{and}\ s''\ ]\!]\\
\quad \mathbf{do}\ \{(a,s) \leftarrow \mathit{get};\\
\qquad ((),s') \leftarrow \mathit{lift}\ (\mathit{bx}.\mathit{set}_L\ a'\ s);\\
\qquad \mathit{set}\ (\mathit{bx}.\mathit{read}_L\ s', s')\}\\
= \quad [\![\ \text{introducing trivial}\ \mathit{snd}\ \text{and}\ \mathit{return}\ ]\!]\\
\quad \mathbf{do}\ \{(a,s) \leftarrow \mathit{get};\\
\qquad ((),s') \leftarrow \mathit{lift}\ (\mathit{bx}.\mathit{set}_L\ a'\ (\mathit{snd}\ (a,s)));\\
\qquad \mathit{set}\ (\mathit{bx}.\mathit{read}_L\ s', s');\\
\qquad \mathit{return}\ ()\}\\
= \quad [\![\ \text{definition of}\ \iota\ h\ ]\!]\\
\quad (\iota\ h)\ \mathit{bx}.\mathit{set}_L
\end{array}
$$

Thus $\mathit{identity} \fatsemi \mathit{bx} \equiv \mathit{bx}$. The reasoning for the second equation is symmetric. □

Lemma 57. For any well-behaved bx, we have

$$(\mathit{bx}_1 \fatsemi \mathit{bx}_2) \fatsemi \mathit{bx}_3 \equiv \mathit{bx}_1 \fatsemi (\mathit{bx}_2 \fatsemi \mathit{bx}_3) \qquad \diamondsuit$$

Proof. Consider composing

$$
\begin{array}{l}
\mathit{bx}_1 :: \mathit{StateTBX}\ T\ S_1\ A\ B\\
\mathit{bx}_2 :: \mathit{StateTBX}\ T\ S_2\ B\ C\\
\mathit{bx}_3 :: \mathit{StateTBX}\ T\ S_3\ C\ D
\end{array}
$$

in the following two ways:

$$\begin{array}{l}
bx_1 \fatsemi (bx_2 \fatsemi bx_3) :: \mathit{StateTBX}\ \ T\ (S_1 \bowtie (S_2 \bowtie S_3))\ A\ C \\
(bx_1 \fatsemi bx_2) \fatsemi bx_3 :: \mathit{StateTBX}\ \ T\ ((S_1 \bowtie S_2) \bowtie S_3)\ A\ C
\end{array}$$

Proving (Assoc) amounts to showing that the obvious isomorphism $h :: (a, (b, c)) \to ((a, b), c)$ induces a monad isomorphism $\iota\ h :: \mathit{StateT}\ (S_1 \bowtie (S_2 \bowtie S_3))\ T\ \alpha \to \mathit{StateT}\ ((S_1 \bowtie S_2) \bowtie S_3)\ T\ \alpha$ and checking that

$$\begin{array}{l}
(\iota\ h)\ (bx_1 \fatsemi (bx_2 \fatsemi bx_3)).\mathit{get}_L \quad\ \, = ((bx_1 \fatsemi bx_2) \fatsemi bx_3).\mathit{get}_L \\
(\iota\ h)\ (bx_1 \fatsemi (bx_2 \fatsemi bx_3)).\mathit{get}_R \quad\ \, = ((bx_1 \fatsemi bx_2) \fatsemi bx_3).\mathit{get}_R \\
(\iota\ h)\ (bx_1 \fatsemi (bx_2 \fatsemi bx_3)).\mathit{set}_L\ a = ((bx_1 \fatsemi bx_2) \fatsemi bx_3).\mathit{set}_L\ a \\
(\iota\ h)\ (bx_1 \fatsemi (bx_2 \fatsemi bx_3)).\mathit{set}_R\ b = ((bx_1 \fatsemi bx_2) \fatsemi bx_3).\mathit{set}_R\ b
\end{array}$$

We outline the proofs of the get_L and set_L cases. For get_L, we make use of the following property:

$$(*) \quad (bx_1 \fatsemi (bx_2 \fatsemi bx_3)).\mathit{get}_L\ (s_1, (s_2, s_3)) = \mathit{return}\ (bx_1.\mathit{read}_L\ s_1, (s_1, (s_2, s_3)))$$

This allows us to consider the get_L condition above:

$$\begin{array}{l}
(\iota\ h)\ (bx_1 \fatsemi (bx_2 \fatsemi bx_3)).\mathit{get}_L \\
\quad = \quad \llbracket \text{ definition of } \iota\ h \ \rrbracket \\
\mathbf{do}\ \{ ((s_1, s_2), s_3) \leftarrow \mathit{get}; \\
\qquad (a', (s_1', (s_2', s_3'))) \leftarrow \mathit{lift}\ ((bx_1 \fatsemi (bx_2 \fatsemi bx_3)).\mathit{get}_L\ (s_1, (s_2, s_3))); \\
\qquad \mathit{set}\ ((s_1', s_2'), s_3'); \\
\qquad \mathit{return}\ a' \} \\
\quad = \quad \llbracket \text{ property } (*) \ \rrbracket \\
\mathbf{do}\ \{ ((s_1, s_2), s_3) \leftarrow \mathit{get}; \\
\qquad (a', (s_1', (s_2', s_3'))) \leftarrow \mathit{lift}\ (\mathit{return}\ (bx_1.\mathit{read}_L\ s_1, (s_1, (s_2, s_3)))); \\
\qquad \mathit{set}\ ((s_1', s_2'), s_3'); \\
\qquad \mathit{return}\ a' \} \\
\quad = \quad \llbracket\ \mathit{lift} \cdot \mathit{return} = \mathit{return}; \text{ inlining } a', s_1', s_2', s_3' \ \rrbracket \\
\mathbf{do}\ \{ ((s_1, s_2), s_3) \leftarrow \mathit{get}; \\
\qquad \mathit{set}\ ((s_1, s_2), s_3); \\
\qquad \mathit{return}\ (bx_1.\mathit{read}_L\ s_1, ((s_1, s_2), s_3)) \} \\
\quad = \quad \llbracket \text{ (GG) and (GS) } \rrbracket \\
\mathbf{do}\ \{ ((s_1, s_2), s_3) \leftarrow \mathit{get}; \\
\qquad \mathit{return}\ (bx_1.\mathit{read}_L\ s_1, ((s_1, s_2), s_3)) \}
\end{array}$$

Analogously to property $(*)$, we may similarly show that

$$((bx_1 \fatsemi bx_2) \fatsemi bx_3).\mathit{get}_L\ ((s_1, s_2), s_3) = \mathit{return}\ (bx_1.\mathit{read}_L\ s_1, ((s_1, s_2), s_3))$$

and the previous proof can be adapted to show that $((bx_1 \fatsemi bx_2) \fatsemi bx_3).\mathit{get}_L$ also simplifies into

$$\mathbf{do}\,\{\,((s_1,s_2),s_3) \leftarrow \mathit{get}; \mathit{return}\ (\mathit{bx}_1.\mathit{read}_L\ s_1, ((s_1,s_2),s_3))\,\}$$

which concludes the proof of the get_L condition.

As for set_L, we use the following property:

$$\begin{aligned}
(\dagger)\quad & (\mathit{bx}_1 \fatsemi (\mathit{bx}_2 \fatsemi \mathit{bx}_3)).\mathit{set}_L\ a'\ (s_1,(s_2,s_3)) \\
&= \mathbf{do}\,\{\,((),s_1') \leftarrow \mathit{bx}_1.\mathit{set}_L\ a'\ s_1; \\
&\qquad (a'',s_2') \leftarrow \mathit{bx}_1.\mathit{get}_R\ s_1'; \\
&\qquad ((),s_2') \leftarrow \mathit{bx}_2.\mathit{set}_L\ a''\ s_2; \\
&\qquad (b',s_2') \leftarrow \mathit{bx}_2.\mathit{get}_R\ s_2'; \\
&\qquad ((),s_3') \leftarrow \mathit{bx}_3.\mathit{set}_L\ b'\ s_3; \\
&\qquad \mathit{return}\ ((),(s_1',(s_2',s_3')))\,\}
\end{aligned}$$

and one may prove an analogous form for

$$((\mathit{bx}_1 \fatsemi \mathit{bx}_2) \fatsemi \mathit{bx}_3).\mathit{set}_L\ a'\ ((s_1,s_2),s_3)$$

where the final line instead reads $\mathit{return}\ ((),((s_1',s_2'),s_3'))$. This allows us to show:

$$\begin{aligned}
& (\iota\ h)\ (\mathit{bx}_1 \fatsemi (\mathit{bx}_2 \fatsemi \mathit{bx}_3)).\mathit{set}_L\ a' \\
&\quad = \quad [\![\ \text{definition of } \iota\ h\ ]\!] \\
& \mathbf{do}\,\{\,((s_1,s_2),s_3) \leftarrow \mathit{get}; \\
&\qquad ((),(s_1',(s_2',s_3'))) \leftarrow \mathit{lift}\ ((\mathit{bx}_1 \fatsemi (\mathit{bx}_2 \fatsemi \mathit{bx}_3)).\mathit{set}_L\ a'\ (s_1,(s_2,s_3))); \\
&\qquad \mathit{set}\ ((s_1',s_2'),s_3'); \\
&\qquad \mathit{return}\ ()\,\} \\
&\quad = \quad [\![\ \text{Property } (\dagger)\ ]\!] \\
& \mathbf{do}\,\{\,((s_1,s_2),s_3) \leftarrow \mathit{get}; \\
&\qquad ((),((s_1',s_2'),s_3')) \leftarrow \mathit{lift}\ (((\mathit{bx}_1 \fatsemi \mathit{bx}_2) \fatsemi \mathit{bx}_3).\mathit{set}_L\ a'\ ((s_1,s_2),s_3)); \\
&\qquad \mathit{set}\ ((s_1',s_2'),s_3'); \\
&\qquad \mathit{return}\ ()\,\} \\
&\quad = \quad [\![\ \text{analogous form of Property } (\dagger)\ ]\!] \\
& ((\mathit{bx}_1 \fatsemi \mathit{bx}_2) \fatsemi \mathit{bx}_3).\mathit{set}_L\ a'
\end{aligned}$$

□

Theorem 26. Composition of transparent bx satisfies the identity and associativity laws, modulo $\equiv$.

$$\begin{aligned}
&(\text{Identity})\ \mathit{identity} \fatsemi \mathit{bx} \equiv \mathit{bx} \equiv \mathit{bx} \fatsemi \mathit{identity} \\
&(\text{Assoc})\quad \mathit{bx}_1 \fatsemi (\mathit{bx}_2 \fatsemi \mathit{bx}_3) \equiv (\mathit{bx}_1 \fatsemi \mathit{bx}_2) \fatsemi \mathit{bx}_3
\end{aligned}$$

♢

Proof. By Lemmas 56 and 57. □

F Proofs from Section 6

Proposition 58. The *dual* operator (Definition 27) preserves well-behavedness, overwritability, and transparency. ♢

Proof. Immediate, since those three properties are invariant under transposing left and right. □

Lemma 59. *left* and *right* are monad morphisms. ♢

Proof. Immediate from Lemma 19, because *left* and *right* are definable as $\vartheta\ l$ where (by Lemma 52) l is a very well-behaved lens. Any ordinary lens is essentially an *Id*-lens, and *Id* is commutative, so Lemma 19 holds. □

Proposition 28. If bx_1 and bx_2 are transparent and well-behaved, then $\mathit{pairBX}\ bx_1\ bx_2$ is transparent and well-behaved. ♢

Proof. Let $bx = \mathit{pairBX}\ bx_1\ bx_2$. Then to show ($\mathrm{G_LS_L}$):

$$
\begin{array}{ll}
 & \mathbf{do}\,\{\, a \leftarrow bx.\mathit{get}_L; bx.\mathit{set}_L\ a \,\} \\
= & [\![\ \text{eta-expansion}\ ]\!] \\
 & \mathbf{do}\,\{\, (a_1, a_2) \leftarrow bx.\mathit{get}_L; bx.\mathit{set}_L\ (a_1, a_2) \,\} \\
= & [\![\ \text{Definition}\ ]\!] \\
 & \mathbf{do}\,\{\, a_1 \leftarrow \mathit{left}\ (bx_1.\mathit{get}_L); a_2 \leftarrow \mathit{right}\ (bx_2.\mathit{get}_L); \\
 & \qquad (a_1', a_2') \leftarrow \mathit{return}\ (a_1, a_2); \\
 & \qquad \mathit{left}\ (bx_1.\mathit{set}_L\ a_1'); \mathit{right}\ (bx_2.\mathit{set}_L\ a_2') \,\} \\
= & [\![\ \text{Monad unit}\ ]\!] \\
 & \mathbf{do}\,\{\, a_1 \leftarrow \mathit{left}\ (bx_1.\mathit{get}_L); a_2 \leftarrow \mathit{right}\ (bx_2.\mathit{get}_L); \\
 & \qquad \mathit{left}\ (bx_1.\mathit{set}_L\ a_1); \mathit{right}\ (bx_2.\mathit{set}_L\ a_2) \,\} \\
= & [\![\ \text{Lemma 54, since } bx_2.\mathit{get}_L \text{ is } T\text{-pure}\ ]\!] \\
 & \mathbf{do}\,\{\, a_1 \leftarrow \mathit{left}\ (bx_1.\mathit{get}_L); \mathit{left}\ (bx_1.\mathit{set}_L\ a_1); \\
 & \qquad a_2 \leftarrow \mathit{right}\ (bx_2.\mathit{get}_L); \mathit{right}\ (bx_2.\mathit{set}_L\ a_2) \,\} \\
= & [\![\ \mathit{left}, \mathit{right}\ \text{monad morphisms}\ ]\!] \\
 & \mathbf{do}\,\{\, \mathit{left}\ (\mathbf{do}\,\{\, a_1 \leftarrow bx_1.\mathit{get}_L; bx_1.\mathit{set}_L\ a_1 \,\}); \\
 & \qquad \mathit{right}\ (\mathbf{do}\,\{\, a_2 \leftarrow bx_2.\mathit{get}_L; bx_2.\mathit{set}_L\ a_2 \,\}) \,\} \\
= & [\![\ (\mathrm{G_LS_L})\ \text{twice}\ ]\!] \\
 & \mathbf{do}\,\{\, \mathit{left}\ (\mathit{return}\ ()); \mathit{right}\ (\mathit{return}\ ()) \,\} \\
= & [\![\ \text{monad morphism, unit}\ ]\!] \\
 & \mathit{return}\ ()
\end{array}
$$

Likewise, to show ($\mathrm{S_LG_L}$):

$$
\begin{array}{ll}
 & \mathbf{do}\,\{\, bx.\mathit{set}_L\ a; bx.\mathit{get}_L \,\} \\
= & [\![\ \text{eta-expansion}\ ]\!] \\
 & \mathbf{do}\,\{\, bx.\mathit{set}_L\ (a_1, a_2); bx.\mathit{get}_L \,\} \\
= & [\![\ \text{definition}\ ]\!] \\
 & \mathbf{do}\,\{\, \mathit{left}\ (bx_1.\mathit{set}_L\ a_1); \mathit{right}\ (bx_2.\mathit{set}_L\ a_2); \\
 & \qquad a_1' \leftarrow \mathit{left}\ (bx_1.\mathit{get}_L); a_2' \leftarrow \mathit{right}\ (bx_2.\mathit{get}_L); \\
 & \qquad \mathit{return}\ (a_1', a_2') \,\} \\
= & [\![\ \text{Lemma 54, since } bx_1.\mathit{get}_L\ T\text{-pure}\ ]\!]
\end{array}
$$

$$
\begin{array}{ll}
& \mathbf{do}\,\{\,\mathit{left}\;(\mathit{bx}_1.\mathit{set}_L\;a_1);\,a_1' \leftarrow \mathit{left}\;(\mathit{bx}_1.\mathit{get}_L);\\
& \qquad \mathit{right}\;(\mathit{bx}_2.\mathit{set}_L\;a_2);\,a_2' \leftarrow \mathit{right}\;(\mathit{bx}_2.\mathit{get}_L);\\
& \qquad \mathit{return}\;(a_1', a_2')\,\}\\
= & [\![\; (\mathrm{S_L G_L})\text{ twice }]\!]\\
& \mathbf{do}\,\{\,\mathit{left}\;(\mathit{bx}_1.\mathit{set}_L\;a_1);\,a_1 \leftarrow \mathit{return}\;a_1;\\
& \qquad \mathit{right}\;(\mathit{bx}_2.\mathit{set}_L\;a_2);\,a_2 \leftarrow \mathit{return}\;a_2;\,\mathit{return}\;(a_1, a_2)\,\}\\
= & [\![\; \text{Monad unit }]\!]\\
& \mathbf{do}\,\{\,\mathit{left}\;(\mathit{bx}_1.\mathit{set}_L\;a_1);\,\mathit{right}\;(\mathit{bx}_2.\mathit{set}_L\;a_2);\,\mathit{return}\;(a_1, a_2)\,\}\\
= & [\![\; \text{Definition }]\!]\\
& \mathbf{do}\,\{\,\mathit{bx}.\mathit{set}_L\;(a_1, a_2);\,\mathit{return}\;(a_1, a_2)\,\}\\
= & [\![\; \text{eta-contraction for pairs }]\!]\\
& \mathbf{do}\,\{\,\mathit{bx}.\mathit{set}_L\;a;\,\mathit{return}\;a\,\}
\end{array}
$$

□

Proposition 29. If bx_1 and bx_2 are transparent and well-behaved then $\mathit{sumBX}\ bx_1\ bx_2$ is transparent and well-behaved. ◇

Proof. Let $bx = \mathit{sumBX}\ bx_1\ bx_2$. We first show that bx is transparent. Suppose that bx_1 is transparent, with *read* functions rl_1 and rr_1; and similarly for bx_2. Then

$$
\begin{array}{ll}
& \mathit{bx}.\mathit{get}_L\\
= & [\![\; \text{definition of } \mathit{sumBX}\;]\!]\\
& \mathbf{do}\,\{\,(b, s_1, s_2) \leftarrow \mathit{get};\\
& \quad \mathbf{if}\;b\;\mathbf{then}\;\mathbf{do}\,\{\,(a_1, \_) \leftarrow \mathit{lift}\;(\mathit{bx}_1.\mathit{get}_L\;s_1);\\
& \qquad\qquad\qquad \mathit{return}\;(\mathit{Left}\;a_1)\,\}\\
& \qquad\quad \mathbf{else}\;\;\mathbf{do}\,\{\,(a_2, \_) \leftarrow \mathit{lift}\;(\mathit{bx}_2.\mathit{get}_L\;s_2);\\
& \qquad\qquad\qquad \mathit{return}\;(\mathit{Right}\;a_2)\,\}\,\}\\
= & [\![\; bx_1 \text{ and } bx_2 \text{ are transparent }]\!]\\
& \mathbf{do}\,\{\,(b, s_1, s_2) \leftarrow \mathit{get};\\
& \quad \mathbf{if}\;b\;\mathbf{then}\;\mathbf{do}\,\{\,(a_1, \_) \leftarrow \mathit{lift}\;(\mathit{gets}\;rl_1\;s_1);\\
& \qquad\qquad\qquad \mathit{return}\;(\mathit{Left}\;a_1)\,\}\\
& \qquad\quad \mathbf{else}\;\;\mathbf{do}\,\{\,(a_2, \_) \leftarrow \mathit{lift}\;(\mathit{gets}\;rl_2\;s_2);\\
& \qquad\qquad\qquad \mathit{return}\;(\mathit{Right}\;a_2)\,\}\,\}\\
= & [\![\; \text{definition of } \mathit{gets}\;]\!]\\
& \mathbf{do}\,\{\,(b, s_1, s_2) \leftarrow \mathit{get};\\
& \quad \mathbf{if}\;b\;\mathbf{then}\;\mathbf{do}\,\{\,(a_1, \_) \leftarrow \mathit{lift}\;(\mathit{return}\;(rl_1\;s_1, s_1));\\
& \qquad\qquad\qquad \mathit{return}\;(\mathit{Left}\;a_1)\,\}\\
& \qquad\quad \mathbf{else}\;\;\mathbf{do}\,\{\,(a_2, \_) \leftarrow \mathit{lift}\;(\mathit{return}\;(rl_2\;s_2, s_2));\\
& \qquad\qquad\qquad \mathit{return}\;(\mathit{Right}\;a_2)\,\}\,\}\\
= & [\![\; \mathit{lift} \text{ is a monad morphism }]\!]\\
& \mathbf{do}\,\{\,(b, s_1, s_2) \leftarrow \mathit{get};\\
& \quad \mathbf{if}\;b\;\mathbf{then}\;\mathbf{do}\,\{\,(a_1, \_) \leftarrow \mathit{return}\;(rl_1\;s_1, s_1);\\
& \qquad\qquad\qquad \mathit{return}\;(\mathit{Left}\;a_1)\,\}\\
& \qquad\quad \mathbf{else}\;\;\mathbf{do}\,\{\,(a_2, \_) \leftarrow \mathit{return}\;(rl_2\;s_2, s_2);
\end{array}
$$

$$
\begin{array}{ll}
 & \qquad\qquad\qquad\qquad \mathit{return}\ (\mathit{Right}\ a_2)\}\} \\
= & \llbracket\ \text{monads}\ \rrbracket \\
 & \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get}; \\
 & \qquad \mathbf{if}\ b\ \mathbf{then}\ \mathbf{do}\ \{\mathbf{let}\ a_1 = \mathit{rl}_1\ s_1; \mathit{return}\ (\mathit{Left}\ a_1)\} \\
 & \qquad\qquad \mathbf{else}\ \ \mathbf{do}\ \{\mathbf{let}\ a_2 = \mathit{rl}_2\ s_2; \mathit{return}\ (\mathit{Right}\ a_2)\}\} \\
= & \llbracket\ \mathbf{do}\ \text{notation}\ \rrbracket \\
 & \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get}; \\
 & \qquad \mathbf{if}\ b\ \mathbf{then}\ \mathbf{do}\ \{\mathit{return}\ (\mathit{Left}\ (\mathit{rl}_1\ s_1))\} \\
 & \qquad\qquad \mathbf{else}\ \ \mathbf{do}\ \{\mathit{return}\ (\mathit{Right}\ (\mathit{rl}_2\ s_2))\}\} \\
= & \llbracket\ \text{definition of } \mathit{gets}\ \rrbracket \\
 & \mathbf{do}\ \{\mathit{gets}\ (\lambda(b, s_1, s_2).\, \mathbf{if}\ b\ \mathbf{then}\ \mathit{Left}\ (\mathit{rl}_1\ s_1) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \ \mathit{Right}\ (\mathit{rl}_2\ s_2))\}
\end{array}
$$

Similarly for $bx.get_R$.

Now suppose also that bx_1 and bx_2 are well-behaved; we show that bx is well-behaved too. Because bx is transparent, it satisfies ($\mathrm{G_LG_L}$), ($\mathrm{G_RG_R}$), and ($\mathrm{G_LG_R}$). For ($\mathrm{G_LS_L}$) we have:

$$
\begin{array}{ll}
 & \mathbf{do}\ \{a \leftarrow bx.get_L; bx.set_L\ a\} \\
= & \llbracket\ \text{definition of } bx.get_L\ \rrbracket \\
 & \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get}; \\
 & \qquad \mathbf{if}\ b\ \mathbf{then}\ \mathbf{do}\ \{(a_1, \_) \leftarrow \mathit{lift}\ (bx_1.get_L\ s_1); \\
 & \qquad\qquad\qquad\qquad\qquad \mathbf{let}\ a = \mathit{Left}\ a_1; bx.set_L\ a\} \\
 & \qquad\qquad \mathbf{else}\ \ \mathbf{do}\ \{(a_2, \_) \leftarrow \mathit{lift}\ (bx_2.get_L\ s_2); \\
 & \qquad\qquad\qquad\qquad\qquad \mathbf{let}\ a = \mathit{Right}\ a_2; bx.set_L\ a\} \\
= & \llbracket\ \text{assume } b \text{ is } \mathit{True} \text{ (the } \mathit{False} \text{ case is symmetric)}\ \rrbracket \\
 & \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get}; \\
 & \qquad (a_1, \_) \leftarrow \mathit{lift}\ (bx_1.get_L\ s_1); \\
 & \qquad \mathbf{let}\ a = \mathit{Left}\ a_1; bx.set_L\ a\} \\
= & \llbracket\ \text{definition of } bx.set_L\ \rrbracket \\
 & \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get}; \\
 & \qquad (a_1, \_) \leftarrow \mathit{lift}\ (bx_1.get_L\ s_1); \\
 & \qquad (b, s_1, s_2) \leftarrow \mathit{get}; \\
 & \qquad ((), s_1') \leftarrow \mathit{lift}\ ((bx_1.set_L\ a_1)\ s_1); \\
 & \qquad \mathit{set}\ (\mathit{True}, s_1', s_2)\} \\
= & \llbracket\ \mathit{get} \text{ commutes with lifting}\ \rrbracket \\
 & \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get}; \\
 & \qquad (b, s_1, s_2) \leftarrow \mathit{get}; \\
 & \qquad (a_1, \_) \leftarrow \mathit{lift}\ (bx_1.get_L\ s_1); \\
 & \qquad ((), s_1') \leftarrow \mathit{lift}\ ((bx_1.set_L\ a_1)\ s_1); \\
 & \qquad \mathit{set}\ (\mathit{True}, s_1', s_2)\} \\
= & \llbracket\ \text{(GG)}\ \rrbracket \\
 & \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get};
\end{array}
$$

$$
\begin{array}{ll}
& \qquad (a_1, \_) \leftarrow \mathit{lift}\ (bx_1.\mathit{get}_L\ s_1);\\
& \qquad ((), s_1') \leftarrow \mathit{lift}\ ((bx_1.\mathit{set}_L\ a_1)\ s_1);\\
& \qquad \mathit{set}\ (\mathit{True}, s_1', s_2)\}\\
= & \llbracket\ \mathit{lift} \text{ is a monad morphism }\ \rrbracket\\
& \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get};\\
& \qquad ((), s_1') \leftarrow \mathit{lift}\ (\mathbf{do}\ \{a_1 \leftarrow bx_1.\mathit{get}_L; bx_1.\mathit{set}_L\ a_1\}\ s_1);\\
& \qquad \mathit{set}\ (\mathit{True}, s_1', s_2)\}\\
= & \llbracket\ (\mathrm{G_L S_L}) \text{ for } bx_1\ \rrbracket\\
& \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get};\\
& \qquad ((), s_1') \leftarrow \mathit{lift}\ (\mathit{return}\ ()\ s_1);\\
& \qquad \mathit{set}\ (\mathit{True}, s_1', s_2)\}\\
= & \llbracket\ \text{definition of } \mathit{lift}\ \rrbracket\\
& \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get};\\
& \qquad \mathbf{let}\ s_1' = s_1;\\
& \qquad \mathit{set}\ (\mathit{True}, s_1', s_2)\}\\
= & \llbracket\ \text{substituting } b = \mathit{True} \text{ and } s_1' = s_1\ \rrbracket\\
& \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get};\\
& \qquad \mathit{set}\ (b, s_1, s_2)\}\\
= & \llbracket\ (\mathrm{GS})\ \rrbracket\\
& \mathbf{do}\ \{\mathit{return}\ ()\}
\end{array}
$$

For ($\mathrm{S_L G_L}$), and setting a *Left* value, we have:

$$
\begin{array}{ll}
& \mathbf{do}\ \{bx.\mathit{set}_L\ (\mathit{Left}\ a_1); bx.\mathit{get}_L\}\\
= & \llbracket\ \text{definition of } bx.\mathit{set}_L\ \rrbracket\\
& \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get};\\
& \qquad ((), s_1') \leftarrow \mathit{lift}\ ((bx_1.\mathit{set}_L\ a_1)\ s_1);\\
& \qquad \mathit{set}\ (\mathit{True}, s_1', s_2);\\
& \qquad bx.\mathit{get}_L\}\\
= & \llbracket\ \text{definition of } bx.\mathit{get}_L\ \rrbracket\\
& \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get};\\
& \qquad ((), s_1') \leftarrow \mathit{lift}\ ((bx_1.\mathit{set}_L\ a_1)\ s_1);\\
& \qquad \mathit{set}\ (\mathit{True}, s_1', s_2);\\
& \qquad (b, s_1'', s_2') \leftarrow \mathit{get};\\
& \qquad \mathbf{if}\ b\ \mathbf{then}\ \mathbf{do}\ \{(a_1', \_) \leftarrow \mathit{lift}\ (bx_1.\mathit{get}_L\ s_1'');\\
& \qquad\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{Left}\ a_1')\}\\
& \qquad\qquad \mathbf{else}\ \ \mathbf{do}\ \{(a_2, \_) \leftarrow \mathit{lift}\ (bx_2.\mathit{get}_L\ s_2');\\
& \qquad\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{Right}\ a_2)\}\}\\
= & \llbracket\ (\mathrm{SG})\ \rrbracket\\
& \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get};\\
& \qquad ((), s_1') \leftarrow \mathit{lift}\ ((bx_1.\mathit{set}_L\ a_1)\ s_1);\\
& \qquad \mathit{set}\ (\mathit{True}, s_1', s_2);\\
& \qquad \mathbf{let}\ (b, s_1'', s_2') = (\mathit{True}, s_1', s_2);\\
& \qquad \mathbf{if}\ b\ \mathbf{then}\ \mathbf{do}\ \{(a_1', \_) \leftarrow \mathit{lift}\ (bx_1.\mathit{get}_L\ s_1'');
\end{array}
$$

$$\begin{array}{l}
\qquad\qquad\qquad\qquad \mathit{return}\ (\mathit{Left}\ a_1')\} \\
\qquad\qquad \mathbf{else}\ \ \mathbf{do}\ \{(a_2, \_) \leftarrow \mathit{lift}\ (bx_2.\mathit{get}_L\ s_2'); \\
\qquad\qquad\qquad\qquad \mathit{return}\ (\mathit{Right}\ a_2)\}\}
\end{array}$$

$$\begin{array}{l}
= \quad \llbracket\ \text{substituting; conditional}\ \rrbracket \\
\ \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get}; \\
\qquad ((), s_1') \leftarrow \mathit{lift}\ ((bx_1.\mathit{set}_L\ a_1)\ s_1); \\
\qquad \mathit{set}\ (\mathit{True}, s_1', s_2); \\
\qquad (a_1', \_) \leftarrow \mathit{lift}\ (bx_1.\mathit{get}_L\ s_1'); \\
\qquad \mathit{return}\ (\mathit{Left}\ a_1')\}
\end{array}$$

$$\begin{array}{l}
= \quad \llbracket\ \mathit{set}\ \text{commutes with lifting; naming the wildcard;}\ \rrbracket \\
\ \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get}; \\
\qquad ((), s_1') \leftarrow \mathit{lift}\ ((bx_1.\mathit{set}_L\ a_1)\ s_1); \\
\qquad (a_1', s_1'') \leftarrow \mathit{lift}\ (bx_1.\mathit{get}_L\ s_1'); \\
\qquad \mathit{set}\ (\mathit{True}, s_1', s_2); \\
\qquad \mathit{return}\ (\mathit{Left}\ a_1')\}
\end{array}$$

$$\begin{array}{l}
= \quad \llbracket\ bx_1\ \text{is transparent, so}\ s_1'' = s_1'\ \rrbracket \\
\ \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get}; \\
\qquad ((), s_1') \leftarrow \mathit{lift}\ ((bx_1.\mathit{set}_L\ a_1)\ s_1); \\
\qquad (a_1', s_1'') \leftarrow \mathit{lift}\ (bx_1.\mathit{get}_L\ s_1'); \\
\qquad \mathit{set}\ (\mathit{True}, s_1'', s_2); \\
\qquad \mathit{return}\ (\mathit{Left}\ a_1')\}
\end{array}$$

$$\begin{array}{l}
= \quad \llbracket\ \mathit{lift}\ \text{is a monad morphism}\ \rrbracket \\
\ \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get}; \\
\qquad (a_1', s_1'') \leftarrow \mathit{lift}\ (\mathbf{do}\ \{bx_1.\mathit{set}_L\ a_1; bx_1.\mathit{get}_L\}\ s_1); \\
\qquad \mathit{set}\ (\mathit{True}, s_1'', s_2); \\
\qquad \mathit{return}\ (\mathit{Left}\ a_1')\}
\end{array}$$

$$\begin{array}{l}
= \quad \llbracket\ (\mathrm{S_L G_L})\ \text{for}\ bx_1\ \rrbracket \\
\ \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get}; \\
\qquad (a_1', s_1'') \leftarrow \mathit{lift}\ (\mathbf{do}\ \{bx_1.\mathit{set}_L\ a_1; \mathit{return}\ a_1\}\ s_1); \\
\qquad \mathit{set}\ (\mathit{True}, s_1'', s_2); \\
\qquad \mathit{return}\ (\mathit{Left}\ a_1')\}
\end{array}$$

$$\begin{array}{l}
= \quad \llbracket\ \mathit{lift}\ \text{is a monad morphism}\ \rrbracket \\
\ \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get}; \\
\qquad ((), s_1') \leftarrow \mathit{lift}\ ((bx_1.\mathit{set}_L\ a_1)\ s_1); \\
\qquad (a_1', s_1'') \leftarrow \mathit{lift}\ (\mathit{return}\ a_1\ s_1'); \\
\qquad \mathit{set}\ (\mathit{True}, s_1'', s_2); \\
\qquad \mathit{return}\ (\mathit{Left}\ a_1')\}
\end{array}$$

$$\begin{array}{l}
= \quad \llbracket\ \mathit{return}\ \text{for}\ \mathit{StateT}\text{:}\ s_1'' = s_1'\ \rrbracket \\
\ \mathbf{do}\ \{(b, s_1, s_2) \leftarrow \mathit{get}; \\
\qquad ((), s_1') \leftarrow \mathit{lift}\ ((bx_1.\mathit{set}_L\ a_1)\ s_1); \\
\qquad (a_1', \_) \leftarrow \mathit{lift}\ (\mathit{return}\ a_1\ s_1'); \\
\qquad \mathit{set}\ (\mathit{True}, s_1', s_2);
\end{array}$$

$$
\begin{array}{ll}
 & \qquad \mathit{return}\ (\mathit{Left}\ a_1')\,\} \\
= & \llbracket\ \mathit{set} \text{ commutes with lifting }\ \rrbracket \\
 & \mathbf{do}\,\{\,(b, s_1, s_2) \leftarrow \mathit{get}; \\
 & \qquad ((), s_1') \leftarrow \mathit{lift}\ ((\mathit{bx}_1.\mathit{set}_L\ a_1)\ s_1); \\
 & \qquad \mathit{set}\ (\mathit{True}, s_1', s_2); \\
 & \qquad (a_1', s_1'') \leftarrow \mathit{lift}\ (\mathit{return}\ a_1\ s_1'); \\
 & \qquad \mathit{return}\ (\mathit{Left}\ a_1')\,\} \\
= & \llbracket\ \text{definitions of } \mathit{return} \text{ and } \mathit{lift}\ \rrbracket \\
 & \mathbf{do}\,\{\,(b, s_1, s_2) \leftarrow \mathit{get}; \\
 & \qquad ((), s_1') \leftarrow \mathit{lift}\ ((\mathit{bx}_1.\mathit{set}_L\ a_1)\ s_1); \\
 & \qquad \mathit{set}\ (\mathit{True}, s_1', s_2); \\
 & \qquad \mathbf{let}\ (a_1', s_1'') = (a_1, s_1'); \\
 & \qquad \mathit{return}\ (\mathit{Left}\ a_1')\,\} \\
= & \llbracket\ \text{definition of } \mathit{bx}.\mathit{set}_L\text{; substituting } a_1' = a_1\ \rrbracket \\
 & \mathbf{do}\,\{\,\mathit{bx}_1.\mathit{set}_L\ (\mathit{Left}\ a_1); \mathit{return}\ (\mathit{Left}\ a_1)\,\}
\end{array}
$$

Of course, setting a *Right* value, and $(\mathrm{G_R S_R})$ and $(\mathrm{S_R G_R})$, are symmetric. □

Proposition 30. If *bx* is transparent and well-behaved, then so is *listIBX bx*. ◇

Proof. Suppose *bx* is transparent and well-behaved. We first show that *listIBX bx* is transparent; we consider only $(\mathit{listIBX}\ \mathit{bx}).\mathit{get}_L$, as get_R is symmetric.

$$
\begin{array}{ll}
 & (\mathit{listIBX}\ \mathit{bx}).\mathit{get}_L \\
= & \llbracket\ \text{definition of } \mathit{listIBX}\ \rrbracket \\
 & \mathbf{do}\,\{\,(n, \mathit{cs}) \leftarrow \mathit{get}; \mathit{mapM}\ (\mathit{lift} \cdot \mathit{eval}\ \mathit{bx}.\mathit{get}_L)\ (\mathit{take}\ n\ \mathit{cs})\,\} \\
= & \llbracket\ \mathit{bx} \text{ is transparent }\ \rrbracket \\
 & \mathbf{do}\,\{\,(n, \mathit{cs}) \leftarrow \mathit{get}; \mathit{mapM}\ (\mathit{lift} \cdot \mathit{eval}\ (\mathit{gets}\ (\mathit{bx}.\mathit{read}_L)))\ (\mathit{take}\ n\ \mathit{cs})\,\} \\
= & \llbracket\ \text{definition of } \mathit{eval}\ \rrbracket \\
 & \mathbf{do}\,\{\,(n, \mathit{cs}) \leftarrow \mathit{get}; \mathit{mapM}\ (\mathit{lift} \cdot \mathit{return} \cdot \mathit{bx}.\mathit{read}_L)\ (\mathit{take}\ n\ \mathit{cs})\,\} \\
= & \llbracket\ \mathit{lift} \text{ is a monad morphism }\ \rrbracket \\
 & \mathbf{do}\,\{\,(n, \mathit{cs}) \leftarrow \mathit{get}; \mathit{mapM}\ (\mathit{return} \cdot \mathit{bx}.\mathit{read}_L)\ (\mathit{take}\ n\ \mathit{cs})\,\} \\
= & \llbracket\ \mathit{mapM}\ (\mathit{return} \cdot f) = \mathit{return} \cdot \mathit{map}\ f\ \rrbracket \\
 & \mathbf{do}\,\{\,(n, \mathit{cs}) \leftarrow \mathit{get}; \mathit{return}\ (\mathit{map}\ (\mathit{bx}.\mathit{read}_L\ (\mathit{take}\ n\ \mathit{cs})))\,\} \\
= & \llbracket\ \text{definition of } \mathit{gets}\ \rrbracket \\
 & \mathit{gets}\ (\lambda(n, \mathit{cs}).\ \mathit{map}\ (\mathit{bx}.\mathit{read}_L)\ (\mathit{take}\ n\ \mathit{cs}))
\end{array}
$$

Now to show that *listIBX* preserves well-behavedness. Note that *sets* simplifies when its two list arguments have the same length, to:

$$
\mathit{sets}\ s\ i\ \mathit{as}\ \mathit{cs} = \mathit{mapM}\ (\mathit{uncurry}\ s)\ (\mathit{zip}\ \mathit{as}\ \mathit{cs})
$$

Then for $(\mathrm{G_L S_L})$, we have:

$$
\begin{array}{ll}
 & \mathbf{do}\,\{\,\mathit{as} \leftarrow (\mathit{listIBX}\ \mathit{bx}).\mathit{get}_L; (\mathit{listIBX}\ \mathit{bx}) \cdot \mathit{set}_L\ \mathit{as}\,\} \\
= & \llbracket\ \text{definition of } \mathit{listIBX}\ \rrbracket
\end{array}
$$

$$
\begin{array}{ll}
& \mathbf{do}\,\{(n,\mathit{cs}) \leftarrow \mathit{get};\\
& \qquad \mathit{as} \leftarrow \mathit{mapM}\ (\mathit{lift}\cdot \mathit{eval}\ \mathit{bx}.\mathit{get}_L)\ (\mathit{take}\ n\ \mathit{cs});\\
& \qquad (\_,\mathit{cs}) \leftarrow \mathit{get};\\
& \qquad \mathit{cs}' \leftarrow \mathit{lift}\ (\mathit{sets}\ (\mathit{exec}\cdot \mathit{bx}.\mathit{set}_L)\ \mathit{bx}.\mathit{init}_L\ \mathit{as}\ \mathit{cs});\\
& \qquad \mathit{set}\ (\mathit{length}\ \mathit{as},\mathit{cs}')\}\\
= & \llbracket\ \mathit{get}\text{ commutes with liftings; (GG) }\rrbracket\\
& \mathbf{do}\,\{(n,\mathit{cs}) \leftarrow \mathit{get};\\
& \qquad \mathit{as} \leftarrow \mathit{mapM}\ (\mathit{lift}\cdot \mathit{eval}\ \mathit{bx}.\mathit{get}_L)\ (\mathit{take}\ n\ \mathit{cs});\\
& \qquad \mathit{cs}' \leftarrow \mathit{lift}\ (\mathit{sets}\ (\mathit{exec}\cdot \mathit{bx}.\mathit{set}_L)\ \mathit{bx}.\mathit{init}_L\ \mathit{as}\ \mathit{cs});\\
& \qquad \mathit{set}\ (\mathit{length}\ \mathit{as},\mathit{cs}')\}\\
= & \llbracket\ \mathit{bx}\text{ is transparent, as above }\rrbracket\\
& \mathbf{do}\,\{(n,\mathit{cs}) \leftarrow \mathit{get};\\
& \qquad \mathbf{let}\ \mathit{as} = \mathit{map}\ (\mathit{bx}.\mathit{read}_R)\ (\mathit{take}\ n\ \mathit{cs});\\
& \qquad \mathit{cs}' \leftarrow \mathit{lift}\ (\mathit{sets}\ (\mathit{exec}\cdot \mathit{bx}.\mathit{set}_L)\ \mathit{bx}.\mathit{init}_L\ \mathit{as}\ \mathit{cs});\\
& \qquad \mathit{set}\ (\mathit{length}\ \mathit{as},\mathit{cs}')\}\\
= & \llbracket\ \mathit{length}\ \mathit{as} = \mathit{length}\ (\mathit{take}\ n\ \mathit{cs}) = n\text{, so }\mathit{sets}\text{ simplifies }\rrbracket\\
& \mathbf{do}\,\{(n,\mathit{cs}) \leftarrow \mathit{get};\\
& \qquad \mathbf{let}\ \mathit{as} = \mathit{map}\ (\mathit{bx}.\mathit{read}_R)\ (\mathit{take}\ n\ \mathit{cs});\\
& \qquad \mathit{cs}' \leftarrow \mathit{lift}\ (\mathit{mapM}\ (\mathit{uncurry}\ (\mathit{exec}\cdot \mathit{bx}.\mathit{set}_L))\ (\mathit{zip}\ \mathit{as}\ \mathit{cs}));\\
& \qquad \mathit{set}\ (n,\mathit{cs}')\}\\
= & \llbracket\ \mathit{zip}\ (\mathit{map}\ f\ \mathit{xs})\ \mathit{xs} = \mathit{map}\ (\lambda x.\,(f\ x, x))\ \mathit{xs}\ \rrbracket\\
& \mathbf{do}\,\{(n,\mathit{cs}) \leftarrow \mathit{get};\\
& \qquad \mathit{cs}' \leftarrow \mathit{lift}\ (\mathit{mapM}\ (\mathit{uncurry}\ (\mathit{exec}\cdot \mathit{bx}.\mathit{set}_L))\\
& \qquad\qquad\qquad\qquad (\mathit{map}\ (\lambda c.\,(\mathit{bx}.\mathit{read}_R\ c, c))\ \mathit{cs}));\\
& \qquad \mathit{set}\ (n,\mathit{cs}')\}\\
= & \llbracket\ \mathit{exec}\ (\mathit{bx}.\mathit{set}_L\ (\mathit{bx}.\mathit{read}_R\ c))\ c = \mathit{return}\ c\text{, by }(\mathrm{G_LS_L})\ \rrbracket\\
& \mathbf{do}\,\{(n,\mathit{cs}) \leftarrow \mathit{get};\\
& \qquad \mathbf{let}\ \mathit{cs}' = \mathit{cs};\\
& \qquad \mathit{set}\ (n,\mathit{cs}')\}\\
= & \llbracket\text{ substituting }\mathit{cs}' = \mathit{cs}\text{; (GS) }\rrbracket\\
& \mathit{return}\ ()
\end{array}
$$

And for $(\mathrm{S_LG_L})$, we note first that

$$
\begin{array}{ll}
& \mathbf{do}\,\{c'' \leftarrow \mathit{lift}\ (\mathit{uncurry}\ (\mathit{exec}\cdot \mathit{bx}.\mathit{set}_L)\ (a, c));\\
& \qquad \mathbf{let}\ a' = \mathit{bx}.\mathit{read}_L\ c''; \mathit{return}\ (a', c'')\}\\
= & \llbracket\ \mathit{uncurry}, \mathit{exec};\ \mathit{bx}\text{ is transparent }\rrbracket\\
& \mathbf{do}\,\{((), c') \leftarrow \mathit{lift}\ ((\mathit{bx}.\mathit{set}_L\ a)\ c);\\
& \qquad (a', c'') \leftarrow \mathit{lift}\ (\mathit{bx}.\mathit{get}_L\ c'); \mathit{return}\ (a', c'')\}\\
= & \llbracket\ \mathit{lift}\text{ is a monad morphism }\rrbracket\\
& \mathbf{do}\,\{(a', c'') \leftarrow \mathit{lift}\ (\mathbf{do}\,\{\mathit{bx}.\mathit{set}_L\ a; \mathit{bx}.\mathit{get}_L\}\ c);\\
& \qquad \mathit{return}\ (a', c'')\}\\
= & \llbracket\ (\mathrm{S_LG_L})\text{ for }\mathit{bx}\ \rrbracket
\end{array}
$$

$$\begin{array}{ll}
& \mathbf{do}\ \{ (a', c'') \leftarrow \mathit{lift}\ (\mathbf{do}\ \{ \mathit{bx}.\mathit{set}_L\ a; \mathit{return}\ a \}\ c); \\
& \qquad \mathit{return}\ (a', c'') \} \\
= & \llbracket \text{ monads: } a' \text{ will be bound to } a \ \rrbracket \\
& \mathbf{do}\ \{ (\_, c'') \leftarrow \mathit{lift}\ (\mathbf{do}\ \{ \mathit{bx}.\mathit{set}_L\ a; \mathit{return}\ a \}\ c); \\
& \qquad \mathbf{let}\ a' = a; \mathit{return}\ (a', c'') \} \\
= & \llbracket \text{ reversing the above steps } \rrbracket \\
& \mathbf{do}\ \{ c'' \leftarrow \mathit{lift}\ (\mathit{uncurry}\ (\mathit{exec} \cdot \mathit{bx}.\mathit{set}_L)\ (a, c)); \\
& \qquad \mathbf{let}\ a' = a; \mathit{return}\ (a', c'') \}
\end{array}$$

and similarly

$$\begin{array}{ll}
& \mathbf{do}\ \{ c'' \leftarrow \mathit{lift}\ (\mathit{bx}.\mathit{init}_L\ a); \\
& \qquad \mathbf{let}\ a' = \mathit{bx}.\mathit{read}_L\ c''; \mathit{return}\ (a', c'') \} \\
= & \llbracket\ \mathit{bx} \text{ is transparent } \rrbracket \\
& \mathbf{do}\ \{ c' \leftarrow \mathit{lift}\ (\mathit{bx}.\mathit{init}_L\ a); \\
& \qquad (a', c'') \leftarrow \mathit{lift}\ (\mathit{bx}.\mathit{get}_L\ c'); \mathit{return}\ (a', c'') \} \\
= & \llbracket\ \mathit{lift} \text{ is a monad morphism } \rrbracket \\
& \mathbf{do}\ \{ (a', c'') \leftarrow \mathit{lift}\ (\mathbf{do}\ \{ c' \leftarrow \mathit{bx}.\mathit{init}_L\ a; \mathit{bx}.\mathit{get}_L\ c' \}); \\
& \qquad \mathit{return}\ (a', c'') \} \\
= & \llbracket\ (\mathrm{I_L G_L}) \text{ for } \mathit{bx} \ \rrbracket \\
& \mathbf{do}\ \{ (a', c'') \leftarrow \mathit{lift}\ (\mathbf{do}\ \{ c' \leftarrow \mathit{bx}.\mathit{init}_L\ a; \mathit{return}\ (a, c') \}); \\
& \qquad \mathit{return}\ (a', c'') \} \\
= & \llbracket \text{ monads: } a' \text{ will be bound to } a \ \rrbracket \\
& \mathbf{do}\ \{ (\_, c'') \leftarrow \mathit{lift}\ (\mathbf{do}\ \{ c' \leftarrow \mathit{bx}.\mathit{init}_L\ a; \mathit{return}\ (a, c') \}); \\
& \qquad \mathbf{let}\ a' = a; \mathit{return}\ (a', c'') \} \\
= & \llbracket \text{ reversing the above steps } \rrbracket \\
& \mathbf{do}\ \{ c'' \leftarrow \mathit{lift}\ (\mathit{bx}.\mathit{init}_L\ a); \\
& \qquad \mathbf{let}\ a' = a; \mathit{return}\ (a', c'') \}
\end{array}$$

and therefore (by induction on as):

$$\begin{array}{ll}
& \mathbf{do}\ \{ \mathit{cs}' \leftarrow \mathit{lift}\ (\mathit{sets}\ (\mathit{exec} \cdot \mathit{bx}.\mathit{set}_L)\ \mathit{bx}.\mathit{init}_L\ \mathit{as}\ \mathit{cs}); \\
& \qquad \mathbf{let}\ \mathit{as}' = \mathit{map}\ (\mathit{bx}.\mathit{read}_L\ (\mathit{take}\ (\mathit{length}\ \mathit{as})\ \mathit{cs}')); k\ \mathit{as}'\ \mathit{cs}' \} \\
= & \llbracket \text{ either way, } \mathit{as}' \text{ gets bound to } \mathit{as} \ \rrbracket \\
& \mathbf{do}\ \{ \mathit{cs}' \leftarrow \mathit{lift}\ (\mathit{sets}\ (\mathit{exec} \cdot \mathit{bx}.\mathit{set}_L)\ \mathit{bx}.\mathit{init}_L\ \mathit{as}\ \mathit{cs}); \\
& \qquad k\ \mathit{as}\ \mathit{cs}' \}
\end{array}$$

Then we have:

$$\begin{array}{ll}
& \mathbf{do}\ \{ (\mathit{listIBX}\ \mathit{bx}) \cdot \mathit{set}_L\ \mathit{as}; (\mathit{listIBX}\ \mathit{bx}).\mathit{get}_L \} \\
= & \llbracket \text{ definition of } \mathit{listIBX} \ \rrbracket \\
& \mathbf{do}\ \{ (\_, \mathit{cs}) \leftarrow \mathit{get}; \\
& \qquad \mathit{cs}' \leftarrow \mathit{lift}\ (\mathit{sets}\ (\mathit{exec} \cdot \mathit{bx}.\mathit{set}_L)\ \mathit{bx}.\mathit{init}_L\ \mathit{as}\ \mathit{cs}); \\
& \qquad \mathit{set}\ (\mathit{length}\ \mathit{as}, \mathit{cs}');
\end{array}$$

$$
\begin{array}{ll}
 & \qquad (n, \mathit{cs}) \leftarrow \mathit{get};\\
 & \qquad \mathit{mapM}\ (\mathit{lift} \cdot \mathit{eval}\ \mathit{bx}.\mathit{get}_L)\ (\mathit{take}\ n\ \mathit{cs})\,\}\\
= & \llbracket\ \text{(SG)}\ \rrbracket\\
 & \textbf{do}\ \{\,(\_, \mathit{cs}) \leftarrow \mathit{get};\\
 & \qquad \mathit{cs}' \leftarrow \mathit{lift}\ (\mathit{sets}\ (\mathit{exec} \cdot \mathit{bx}.\mathit{set}_L)\ \mathit{bx}.\mathit{init}_L\ \mathit{as}\ \mathit{cs});\\
 & \qquad \mathit{set}\ (\mathit{length}\ \mathit{as}, \mathit{cs}');\\
 & \qquad \mathit{mapM}\ (\mathit{lift} \cdot \mathit{eval}\ \mathit{bx}.\mathit{get}_L)\ (\mathit{take}\ (\mathit{length}\ \mathit{as})\ \mathit{cs}')\,\}\\
= & \llbracket\ \mathit{bx}\ \text{is transparent, as above}\ \rrbracket\\
 & \textbf{do}\ \{\,\mathit{cs} \leftarrow \mathit{get};\\
 & \qquad \mathit{cs}' \leftarrow \mathit{lift}\ (\mathit{sets}\ (\mathit{exec} \cdot \mathit{bx}.\mathit{set}_L)\ \mathit{bx}.\mathit{init}_L\ \mathit{as}\ \mathit{cs});\\
 & \qquad \mathit{set}\ (\mathit{length}\ \mathit{as}, \mathit{cs}');\\
 & \qquad \mathit{return}\ (\mathit{map}\ (\mathit{bx}.\mathit{read}_L\ (\mathit{take}\ (\mathit{length}\ \mathit{as})\ \mathit{cs}')))\,\}\\
= & \llbracket\ \text{observation above}\ \rrbracket\\
 & \textbf{do}\ \{\,\mathit{cs} \leftarrow \mathit{get};\\
 & \qquad \mathit{cs}' \leftarrow \mathit{lift}\ (\mathit{sets}\ (\mathit{exec} \cdot \mathit{bx}.\mathit{set}_L)\ \mathit{bx}.\mathit{init}_L\ \mathit{as}\ \mathit{cs});\\
 & \qquad \mathit{set}\ (\mathit{length}\ \mathit{as}, \mathit{cs}');\\
 & \qquad \mathit{return}\ \mathit{as}\,\}
\end{array}
$$

Finally, for $(\mathrm{I_L G_L})$ we have:

$$
\begin{array}{ll}
 & \textbf{do}\ \{\,\mathit{cs} \leftarrow (\mathit{listIBX}\ \mathit{bx}).\mathit{init}_L\ \mathit{as};\\
 & \qquad (\mathit{listIBX}\ \mathit{bx}).\mathit{get}_L\ (\mathit{length}\ \mathit{as}, \mathit{cs})\,\}\\
= & \llbracket\ \text{definition of}\ \mathit{listIBX};\ \mathit{bx}\ \text{is transparent}\ \rrbracket\\
 & \textbf{do}\ \{\,\mathit{cs} \leftarrow \mathit{mapM}\ (\mathit{bx}.\mathit{init}_L)\ \mathit{as};\\
 & \qquad \lambda(n, \mathit{cs}).\ \mathit{gets}\ (\mathit{map}\ (\mathit{bx}.\mathit{read}_L))\ (\mathit{length}\ \mathit{as}, \mathit{cs})\,\}\\
= & \llbracket\ \text{definition of}\ \mathit{gets}\ \rrbracket\\
 & \textbf{do}\ \{\,\mathit{cs} \leftarrow \mathit{mapM}\ (\mathit{bx}.\mathit{init}_L)\ \mathit{as};\\
 & \qquad \mathit{return}\ (\mathit{map}\ (\mathit{bx}.\mathit{read}_L)\ (\mathit{take}\ (\mathit{length}\ \mathit{as})\ \mathit{cs}), \mathit{cs})\,\}\\
= & \llbracket\ \mathit{length}\ \mathit{cs} = \mathit{length}\ \mathit{as}\ \text{by definition of}\ \mathit{mapM}\ \rrbracket\\
 & \textbf{do}\ \{\,\mathit{cs} \leftarrow \mathit{mapM}\ (\mathit{bx}.\mathit{init}_L)\ \mathit{as};\\
 & \qquad \mathit{return}\ (\mathit{map}\ (\mathit{bx}.\mathit{read}_L)\ (\mathit{take}\ (\mathit{length}\ \mathit{cs})\ \mathit{cs}), \mathit{cs})\,\}\\
= & \llbracket\ \mathit{take}\ (\mathit{length}\ \mathit{cs}) = \mathit{cs}\ \rrbracket\\
 & \textbf{do}\ \{\,\mathit{cs} \leftarrow \mathit{mapM}\ (\mathit{bx}.\mathit{init}_L)\ \mathit{as};\\
 & \qquad \mathit{return}\ (\mathit{map}\ (\mathit{bx}.\mathit{read}_L)\ \mathit{cs}, \mathit{cs})\,\}\\
= & \llbracket\ (\mathrm{I_L G_L})\ \text{for}\ \mathit{bx}\ \rrbracket\\
 & \textbf{do}\ \{\,\mathit{cs} \leftarrow \mathit{mapM}\ (\mathit{bx}.\mathit{init}_L)\ \mathit{as}; \mathit{return}\ (\mathit{as}, \mathit{cs})\,\}\\
= & \llbracket\ \text{definition of}\ \mathit{listIBX}\ \text{again}\ \rrbracket\\
 & \textbf{do}\ \{\,\mathit{cs} \leftarrow ((\mathit{listIBX}\ \mathit{bx}).\mathit{init}_L\ \mathit{as}); \mathit{return}\ (\mathit{as}, \mathit{cs})\,\}
\end{array}
$$

The proofs on the right are of course symmetric, so omitted. □

Proposition 32. If $f\ c :: \mathit{StateTBX}\ (\mathit{Reader}\ C)\ S\ A\ B$ is transparent and well-behaved for any $c :: C$, then $\mathit{switch}\ f :: \mathit{StateTBX}\ (\mathit{Reader}\ C)\ S\ A\ B$ is well-behaved, but not necessarily transparent. ♢

Proof. The failure of transparency is illustrated by taking f to be any non-constant function. For example, take

$$\begin{aligned}
\tau &= \mathit{StateTBX}\ \mathit{Id}\ (A, A)\\
\alpha &= (A, A)\\
\beta &= A\\
\gamma &= \mathit{Bool}\\
f\ b &= \mathbf{if}\ b\ \mathbf{then}\ \mathit{fstBX}\ \mathbf{else}\ \mathit{sndBX}
\end{aligned}$$

Then the get_L operation of $\mathit{switch}\, f$ is of the form

$$\mathbf{do}\,\{b \leftarrow \mathit{lift}\ \mathit{ask}; (f\ b).\mathit{get}_L\}$$

which is equivalent to

$$\mathbf{do}\,\{b \leftarrow \mathit{lift}\ \mathit{ask}; \mathbf{if}\ b\ \mathbf{then}\ \mathit{fstBX}.\mathit{get}_L\ \mathbf{else}\ \mathit{sndBX}.\mathit{get}_L\}$$

which is clearly not $(\mathit{Reader}\ \mathit{Bool})$-pure.

We now consider the preservation of well-behavedness. Clearly, the *get* operations commute so $(\mathrm{G_LG_L})$, $(\mathrm{G_RG_R})$ and $(\mathrm{G_LG_R})$ hold. As usual, we prove the laws for the left side only; the rest are symmetric.

To show $(\mathrm{G_LS_L})$:

$$\begin{aligned}
&\mathbf{do}\,\{a \leftarrow (\mathit{switch}\, f).\mathit{get}_L; (\mathit{switch}\, f).\mathit{set}_L\ a\}\\
={}& \quad \llbracket \text{ Definition } \rrbracket\\
&\mathbf{do}\,\{c \leftarrow \mathit{lift}\ \mathit{ask}; a \leftarrow (f\ c).\mathit{get}_L; c' \leftarrow \mathit{lift}\ \mathit{ask}; (f\ c').\mathit{set}_L\ a\}\\
={}& \quad \llbracket\ \mathit{lift}\ \mathit{ask} \text{ commutes with any } (\mathit{Reader}\ \gamma)\text{-pure operation } \rrbracket\\
&\mathbf{do}\,\{c \leftarrow \mathit{lift}\ \mathit{ask}; c' \leftarrow \mathit{lift}\ \mathit{ask}; a \leftarrow (f\ c).\mathit{get}_L; (f\ c').\mathit{set}_L\ a\}\\
={}& \quad \llbracket\ \mathit{lift}\ \mathit{ask} \text{ is copyable } \rrbracket\\
&\mathbf{do}\,\{c \leftarrow \mathit{lift}\ \mathit{ask}; a \leftarrow (f\ c).\mathit{get}_L; (f\ c').\mathit{set}_L\ a\}\\
={}& \quad \llbracket\ (\mathrm{G_LS_L})\ \rrbracket\\
&\mathbf{do}\,\{c \leftarrow \mathit{lift}\ \mathit{ask}; \mathit{return}\ ()\}\\
={}& \quad \llbracket\ \mathit{lift}\ \mathit{ask} \text{ is discardable } \rrbracket\\
&\mathit{return}\ ()
\end{aligned}$$

To show $(\mathrm{S_LG_L})$:

$$\begin{aligned}
&\mathbf{do}\,\{(\mathit{switch}\, f).\mathit{set}_L\ a; (\mathit{switch}\, f).\mathit{get}_L\}\\
={}& \quad \llbracket \text{ Definition } \rrbracket\\
&\mathbf{do}\,\{c \leftarrow \mathit{lift}\ \mathit{ask}; (f\ c).\mathit{set}_L\ a; c' \leftarrow \mathit{lift}\ \mathit{ask}; (f\ c').\mathit{get}_L\}\\
={}& \quad \llbracket\ \mathit{lift}\ \mathit{ask} \text{ commutes with any } (\mathit{Reader}\ \gamma)\text{-pure operation } \rrbracket\\
&\mathbf{do}\,\{c \leftarrow \mathit{lift}\ \mathit{ask}; c' \leftarrow \mathit{lift}\ \mathit{ask}; (f\ c).\mathit{set}_L\ a; (f\ c').\mathit{get}_L\}\\
={}& \quad \llbracket\ \mathit{lift}\ \mathit{ask} \text{ is copyable } \rrbracket\\
&\mathbf{do}\,\{c \leftarrow \mathit{lift}\ \mathit{ask}; (f\ c).\mathit{set}_L\ a; (f\ c).\mathit{get}_L\}\\
={}& \quad \llbracket\ (\mathrm{S_LG_L})\ \rrbracket
\end{aligned}$$

$$\begin{array}{ll}
& \mathbf{do}\,\{\, c \leftarrow \mathit{lift}\ \mathit{ask}; (f\ c).\mathit{set}_L\ a; \mathit{return}\ a \,\} \\
= & [\![\ \text{Definition} \ ]\!] \\
& \mathbf{do}\,\{\, (\mathit{switch}\ f).\mathit{set}_L\ a; \mathit{return}\ a \,\}
\end{array}$$

□

Proposition 33. Suppose $f :: A \rightarrow \mathit{Maybe}\ B$ and $g :: B \rightarrow \mathit{Maybe}\ A$ are partial inverses; that is, for any a, b we have $f\ a = \mathit{Just}\ b$ if and only if $g\ b = \mathit{Just}\ a$, and that err is a zero element for monad T. Then $\mathit{partialBX}\ \mathit{err}\ f\ g :: \mathit{StateTBX}\ T\ S\ A\ B$ is well-behaved, where $S = \{\, (a, b) \mid f\ a = \mathit{Just}\ b \wedge g\ b = \mathit{Just}\ a \,\}$. ♢

Proof. Suppose f, g are partial inverses and err a zero element of T, and let

$$\mathit{bx} = \mathit{partialBX}\ \mathit{err}\ f\ g :: \mathit{StateTBX}\ T\ S\ A\ B$$

The laws ($\mathrm{G_L G_L}$), ($\mathrm{G_R G_R}$) and ($\mathrm{G_L G_R}$) are immediate because the get_L and get_R operations are clearly T-pure. It is also straightforward to verify that the operations maintain the invariant that the states (a, b) satisfy $f\ a = \mathit{Just}\ b \wedge g\ b = \mathit{Just}\ a$, because the get operations do not change the state and the set operations either yield an error, or set the state to (a, b) where $f\ a = \mathit{Just}\ b$ (and therefore $g\ b = \mathit{Just}\ a$, since f, g are partial inverses).

For ($\mathrm{G_L S_L}$), we proceed as follows:

$$\begin{array}{ll}
& \mathbf{do}\,\{\, a \leftarrow \mathit{bx}.\mathit{get}_L; \mathit{bx}.\mathit{set}_L\ a \,\} \\
= & [\![\ \text{definition of } \mathit{bx}, \mathit{gets}, \mathit{fst}, \text{monad unit} \ ]\!] \\
& \mathbf{do}\,\{\, (a, b) \leftarrow \mathit{get}; \\
& \qquad \mathbf{case}\ f\ a\ \mathbf{of} \\
& \qquad\quad \mathit{Just}\ b' \rightarrow \mathit{set}\ (a, b') \\
& \qquad\quad \mathit{Nothing} \rightarrow \mathit{lift}\ \mathit{err} \,\} \\
= & [\![\ \text{The state } (a, b) \text{ is consistent, so } f\ a = \mathit{Just}\ b \ ]\!] \\
& \mathbf{do}\,\{\, a \leftarrow \mathit{gets}\ \mathit{fst}; \\
& \qquad \mathbf{case}\ \mathit{Just}\ b\ \mathbf{of} \\
& \qquad\quad \mathit{Just}\ b' \rightarrow \mathit{set}\ (a, b') \\
& \qquad\quad \mathit{Nothing} \rightarrow \mathit{lift}\ \mathit{err} \,\} \\
= & [\![\ \mathbf{case} \text{ simplification} \ ]\!] \\
& \mathbf{do}\,\{\, (a, b) \leftarrow \mathit{get}; \\
& \qquad \mathit{set}\ (a, b) \,\} \\
= & [\![\ (\mathrm{GS}) \ ]\!] \\
& \mathit{return}\ ()
\end{array}$$

For ($\mathrm{S_L G_L}$), there are two cases. If $f\ a = \mathit{Nothing}$, we reason as follows:

$$\begin{array}{ll}
& \mathbf{do}\,\{\, \mathit{bx}.\mathit{set}_L\ a; \mathit{bx}.\mathit{get}_L \,\} \\
= & [\![\ \text{Definition of } \mathit{get}_L, \mathit{set}_L, \mathit{gets}\ \mathit{fst} \ ]\!] \\
& \mathbf{do}\,\{\, \mathbf{case}\ f\ a\ \mathbf{of} \\
& \qquad\quad \mathit{Just}\ b' \rightarrow \mathit{set}\ (a, b') \\
& \qquad\quad \mathit{Nothing} \rightarrow \mathit{lift}\ \mathit{err};
\end{array}$$

$$\begin{array}{ll}
& \qquad (a', b') \leftarrow \mathit{get}; \mathit{return}\ a' \} \\
= & [\![\ f\ a = \mathit{Nothing}, \text{simplify } \mathbf{case}\ ]\!] \\
& \mathbf{do}\ \{ \mathit{lift}\ \mathit{err}; \\
& \qquad (a', b') \leftarrow \mathit{get}; \mathit{return}\ a' \} \\
= & [\![\ \mathit{err} \text{ is a zero element }]\!] \\
& \mathbf{do}\ \{ \mathit{lift}\ \mathit{err} \} \\
= & [\![\ \mathit{err} \text{ is a zero element }]\!] \\
& \mathbf{do}\ \{ \mathit{lift}\ \mathit{err}; \mathit{return}\ a \} \\
= & [\![\ \text{reverse previous steps }]\!] \\
& \mathbf{do}\ \{ \mathbf{case}\ f\ a\ \mathbf{of} \\
& \qquad\quad \mathit{Just}\ b' \rightarrow \mathit{set}\ (a, b') \\
& \qquad\quad \mathit{Nothing} \rightarrow \mathit{lift}\ \mathit{err}; \\
& \qquad \mathit{return}\ a' \} \\
= & [\![\ \text{definition }]\!] \\
& \mathbf{do}\ \{ \mathit{bx}.\mathit{set}_L\ a; \mathit{return}\ a \}
\end{array}$$

On the other hand, if $f\ a = \mathit{Just}\ b$ for some b then we reason as follows:

$$\begin{array}{ll}
& \mathbf{do}\ \{ \mathit{bx}.\mathit{set}_L\ a; \mathit{bx}.\mathit{get}_L \} \\
= & [\![\ \text{Definition of } \mathit{get}_L, \mathit{set}_L, \mathit{gets}\ \mathit{fst}\ ]\!] \\
& \mathbf{do}\ \{ \mathbf{case}\ f\ a\ \mathbf{of} \\
& \qquad\quad \mathit{Just}\ b' \rightarrow \mathit{set}\ (a, b') \\
& \qquad\quad \mathit{Nothing} \rightarrow \mathit{lift}\ \mathit{err}; \\
& \qquad (a', b') \leftarrow \mathit{get}; \mathit{return}\ a' \} \\
= & [\![\ f\ a = \mathit{Just}\ b, \text{simplify } \mathbf{case}\ ]\!] \\
& \mathbf{do}\ \{ \mathit{set}\ (a, b); \\
& \qquad (a', b') \leftarrow \mathit{get}; \mathit{return}\ a' \} \\
= & [\![\ \text{(SG) }]\!] \\
& \mathbf{do}\ \{ \mathit{set}\ (a, b); \\
& \qquad \mathit{return}\ a' \} \\
= & [\![\ \text{reverse previous steps }]\!] \\
& \mathbf{do}\ \{ \mathbf{case}\ f\ a\ \mathbf{of} \\
& \qquad\quad \mathit{Just}\ b' \rightarrow \mathit{set}\ (a, b') \\
& \qquad\quad \mathit{Nothing} \rightarrow \mathit{lift}\ \mathit{err}; \\
& \qquad \mathit{return}\ a' \} \\
= & [\![\ \text{definition }]\!] \\
& \mathbf{do}\ \{ \mathit{bx}.\mathit{set}_L\ a; \mathit{return}\ a \}
\end{array}$$

□

Proposition 34. Assume that *ok*, *as* and *bs* satisfy the following equations:

$$\begin{array}{l}
a \in \mathit{as}\ b \Rightarrow \mathit{ok}\ a\ b \\
b \in \mathit{bs}\ a \Rightarrow \mathit{ok}\ a\ b
\end{array}$$

Then *nondetBX ok bs as* is well-behaved. ◇

Proof. For well-definedness on the state space $\{(a, b) \mid \mathit{ok}\ a\ b\}$, we reason as follows. Suppose $\mathit{ok}\ a\ b$ holds. Then clearly, after doing a *get* the state is unchanged and this continues to hold. After a *set*, if the new value of a' satisfies $\mathit{ok}\ a'\ b$ then the updated state will be (a', b), so the invariant is maintained. Otherwise, the updated state will be (a', b') where $b' \in \mathit{bs}\ a'$, so by assumption $\mathit{ok}\ a'\ b'$ holds.

For $(\mathrm{G_L S_L})$ we reason as follows:

$$
\begin{array}{ll}
 & \mathbf{do}\,\{a \leftarrow (\mathit{nondetBX}\ \mathit{as}\ \mathit{bs}).\mathit{get}_L; (\mathit{nondetBX}\ \mathit{as}\ \mathit{bs}).\mathit{set}_L\ a\} \\
= & [\![\ \text{definition}\ ]\!] \\
 & \mathbf{do}\,\{(a, b) \leftarrow \mathit{get}; \mathbf{let}\ a' = a; (a'', b'') \leftarrow \mathit{get}; \\
 & \qquad \mathbf{if}\ \mathit{ok}\ a'\ b''\ \mathbf{then}\ \mathit{set}\ (a', b'') \\
 & \qquad \mathbf{else\ do}\,\{b' \leftarrow \mathit{lift}\ (\mathit{bs}\ a'); \mathit{set}\ (a', b')\}\} \\
= & [\![\ \text{inline } \mathbf{let}\ ]\!] \\
 & \mathbf{do}\,\{(a, b) \leftarrow \mathit{get}; (a'', b'') \leftarrow \mathit{get}; \\
 & \qquad \mathbf{if}\ \mathit{ok}\ a\ b''\ \mathbf{then}\ \mathit{set}\ (a', b'') \\
 & \qquad \mathbf{else\ do}\,\{b' \leftarrow \mathit{lift}\ (\mathit{bs}\ a); \mathit{set}\ (a, b')\}\} \\
= & [\![\ (\mathrm{GG})\ ]\!] \\
 & \mathbf{do}\,\{(a, b) \leftarrow \mathit{get}; \\
 & \qquad \mathbf{if}\ \mathit{ok}\ a\ b\ \mathbf{then}\ \mathit{set}\ (a, b) \\
 & \qquad \mathbf{else\ do}\,\{b' \leftarrow \mathit{lift}\ (\mathit{bs}\ a); \mathit{set}\ (a, b')\}\} \\
= & [\![\ \mathit{ok}\ a\ b \mathrel{==} \mathit{True}\ ]\!] \\
 & \mathbf{do}\,\{(a, b) \leftarrow \mathit{get}; \mathit{set}\ (a, b)\} \\
= & [\![\ (\mathrm{GS})\ ]\!] \\
 & \mathit{return}\ ()
\end{array}
$$

Note that we rely on the invariant (not explicit in the type) that the state values (a, b) satisfy $\mathit{ok}\ a\ b = \mathit{True}$.

For $(\mathrm{S_L G_L})$ the reasoning is as follows:

$$
\begin{array}{ll}
 & \mathbf{do}\,\{\mathit{bx}.\mathit{set}_L\ a; \mathit{bx}.\mathit{get}_L\} \\
= & [\![\ \text{Definition}\ ]\!] \\
 & \mathbf{do}\,\{(a, b) \leftarrow \mathit{get} \\
 & \qquad \mathbf{if}\ \mathit{ok}\ a'\ b\ \mathbf{then}\ \mathit{set}\ (a', b) \\
 & \qquad \mathbf{else\ do}\,\{b' \leftarrow \mathit{lift}\ (\mathit{bs}\ a'); \mathit{set}\ (a', b')\}; \\
 & \qquad (a'', b'') \leftarrow \mathit{get}; \mathit{return}\ a''\} \\
= & [\![\ (\mathrm{SG})\ ]\!] \\
 & \mathbf{do}\,\{(a, b) \leftarrow \mathit{get} \\
 & \qquad \mathbf{if}\ \mathit{ok}\ a'\ b\ \mathbf{then}\ \mathit{set}\ (a', b) \\
 & \qquad \mathbf{else\ do}\,\{b' \leftarrow \mathit{lift}\ (\mathit{bs}\ a'); \mathit{set}\ (a', b')\}; \\
 & \qquad \mathit{return}\ a'\}
\end{array}
$$

□

Proposition 35. If A and B are types equipped with a correct notion of equality (so $a = b$ if and only if $(a \mathrel{==} b) = \mathit{True}$), and $\mathit{bx} :: \mathit{StateTBX}\ T\ S\ A\ B$ then $\mathit{signalBX}\ \mathit{sigA}\ \mathit{sigB}\ \mathit{bx} :: \mathit{StateTBX}\ T\ S\ A\ B$ is well-behaved. ♢

Proof. First, observe that the *get* operations are defined so as to obviously be T-pure, and therefore $(\mathrm{G_L G_R})$ holds. Let $bx' = \mathit{signalBX}\ \mathit{sigA}\ \mathit{sigB}\ bx$.

For $(\mathrm{G_L S_L})$, we proceed as follows::

$$
\begin{array}{ll}
 & \mathbf{do}\ \{\, a \leftarrow bx'.get_L; bx'.set_L\ a \,\} \\
= & [\![\ \text{Definitions}\ ]\!] \\
 & \mathbf{do}\ \{\, a \leftarrow bx.get_L; a' \leftarrow bx.get_L; bx.set_L\ a; \\
 & \quad \mathit{lift}\ (\mathbf{if}\ a \not\doteq a'\ \mathbf{then}\ \mathit{sigA}\ a'\ \mathbf{else}\ \mathit{return}\ ())\,\} \\
= & [\![\ bx.get_L\ \text{copyable}\ ]\!] \\
 & \mathbf{do}\ \{\, a \leftarrow bx.get_L; bx.set_L\ a; \\
 & \quad \mathit{lift}\ (\mathbf{if}\ a \not\doteq a'\ \mathbf{then}\ \mathit{sigA}\ a'\ \mathbf{else}\ \mathit{return}\ ())\,\} \\
= & [\![\ a \not\doteq a \doteq \mathit{False}\ ]\!] \\
 & \mathbf{do}\ \{\, a \leftarrow bx.get_L; bx.set_L\ a; \mathit{lift}\ (\mathit{return}\ ())\,\} \\
= & [\![\ (\mathrm{G_L S_L})\ ]\!] \\
 & \mathbf{do}\ \{\, \mathit{return}\ (); \mathit{lift}\ (\mathit{return}\ ())\,\} \\
= & [\![\ \text{monad unit, } \mathit{lift}\ \text{monad morphism}\ ]\!] \\
 & \mathit{return}\ ()
\end{array}
$$

$(\mathrm{S_L G_L})$:

$$
\begin{array}{ll}
 & \mathbf{do}\ \{\, bx'.set_L\ a; bx'.get_L \,\} \\
= & [\![\ \text{Definition}\ ]\!] \\
 & \mathbf{do}\ \{\, a' \leftarrow bx.get_L; bx.set_L\ a; \\
 & \qquad \mathit{lift}\ (\mathbf{if}\ a \not\doteq a'\ \mathbf{then}\ \mathit{sigA}\ a'\ \mathbf{else}\ \mathit{return}\ ()); \\
 & \qquad bx.get_L \,\} \\
= & [\![\ \text{Monad unit}\ ]\!] \\
 & \mathbf{do}\ \{\, a' \leftarrow bx.get_L; bx.set_L\ a; \\
 & \qquad \mathit{lift}\ (\mathbf{if}\ a \not\doteq a'\ \mathbf{then}\ \mathit{sigA}\ a'\ \mathbf{else}\ \mathit{return}\ ()); \\
 & \qquad a'' \leftarrow bx.get_L; \mathit{return}\ a'' \,\} \\
= & [\![\ \text{Lemma 7}\ ]\!] \\
 & \mathbf{do}\ \{\, a' \leftarrow bx.get_L; bx.set_L\ a; a'' \leftarrow bx.get_L; \\
 & \qquad \mathit{lift}\ (\mathbf{if}\ a \not\doteq a'\ \mathbf{then}\ \mathit{sigA}\ a'\ \mathbf{else}\ \mathit{return}\ ()); \mathit{return}\ a'' \,\} \\
= & [\![\ (\mathrm{S_L G_L})\ ]\!] \\
 & \mathbf{do}\ \{\, a' \leftarrow bx.get_L; bx.set_L\ a; a'' \leftarrow \mathit{return}\ a; \\
 & \qquad \mathit{lift}\ (\mathbf{if}\ a \not\doteq a'\ \mathbf{then}\ \mathit{sigA}\ a'\ \mathbf{else}\ \mathit{return}\ ()); \mathit{return}\ a'' \,\} \\
= & [\![\ \text{Monad unit}\ ]\!] \\
 & \mathbf{do}\ \{\, a' \leftarrow bx.get_L; bx.set_L\ a; \\
 & \qquad \mathit{lift}\ (\mathbf{if}\ a \not\doteq a'\ \mathbf{then}\ \mathit{sigA}\ a'\ \mathbf{else}\ \mathit{return}\ ()); \mathit{return}\ a \,\} \\
= & [\![\ \text{Definition}\ ]\!] \\
 & \mathbf{do}\ \{\, (\mathit{signalBX}\ \mathit{sigA}\ \mathit{sigB}\ bx).set_L\ a; \mathit{return}\ a \,\}
\end{array}
$$

□

Proposition 36. For any f, g, the dynamic bx *dynamicBX* f g is well-behaved. ◇

Proof. Let $bx = dynamicBX\ f\ g$ for some f, g. For ($\mathrm{S_L G_L}$), by construction, an invocation of $bx.set_L\ a'$ ends by setting the state to $((a', b'), fs, bs)$ for some b', fs, bs, and a subsequent $bx.get_L$ will return a'. More formally, we proceed as follows:

```
   do {bx.setL a'; bx.getL}
=    ⟦ definition ⟧
   do {((a, b), fs, bs) ← get;
         if a == a' then return () else
            do b' ← case lookup (a', b) fs of
                    Just b'  → return b'
                    Nothing → lift (f a' b)
                set ((a', b'), ((a', b), b') : fs, bs);
         ((a'', b''), fs'', gs'') ← get; return a''}
```

We now consider three sub-cases.

First, if $a == a'$ then

```
   do {((a, b), fs, bs) ← get;
         if a == a' then return () else
            do b' ← case lookup (a', b) fs of
                    Just b'  → return b'
                    Nothing → lift (f a' b)
                set ((a', b'), ((a', b), b') : fs, bs);
         ((a'', b''), fs'', gs'') ← get; return a''}
=    ⟦ a == a' ⟧
   do {((a, b), fs, bs) ← get;
         return ();
         ((a'', b''), fs'', gs'') ← get; return a''}
=    ⟦ (GG) ⟧
   do {((a, b), fs, bs) ← get;
         return a}
=    ⟦ reversing previous steps ⟧
   do {((a, b), fs, bs) ← get;
         if a == a' then return () else
            do b' ← case lookup (a', b) fs of
                    Just b'  → return b'
                    Nothing → lift (f a' b)
                set ((a', b'), ((a', b), b') : fs, bs);
         return a}
=    ⟦ definition, a = a' ⟧
   do {bx.setL a'; return a'}
```

Second, if $a \not\equiv a'$ and $((a', b), b') \in fs$ for some b', then $lookup\ (a', b)\ fs = Just\ b'$ holds, so:

```
  do {((a, b), fs, bs) ← get;
        if a ≡ a′ then return () else
           do b′ ← case lookup (a′, b) fs of
                     Just b′   → return b′
                     Nothing → lift (f a′ b)
                 set ((a′, b′), ((a′, b), b′) : fs, bs);
        ((a″, b″), fs″, gs″) ← get; return a″}
=   ⟦ a ≢ a′, case simplification ⟧
  do {((a, b), fs, bs) ← get;
        b′ ← return b′;
        set ((a′, b′), ((a′, b), b′) : fs, bs);
        ((a″, b″), fs″, gs″) ← get; return a″}
=   ⟦ monad unit ⟧
  do {((a, b), fs, bs) ← get;
        set ((a′, b′), ((a′, b), b′) : fs, bs);
        ((a″, b″), fs″, gs″) ← get; return a″}
=   ⟦ (GG) ⟧
  do {((a, b), fs, bs) ← get;
        set ((a′, b′), ((a′, b), b′) : fs, bs);
        return a′}
=   ⟦ reversing previous steps ⟧
  do {((a, b), fs, bs) ← get
        if a ≡ a′ then return () else
           do b′ ← case lookup (a′, b) fs of
                     Just b′   → return b′
                     Nothing → lift (f a′ b)
                 set ((a′, b′), ((a′, b), b′) : fs, bs);
        return a′}
=   ⟦ definition ⟧
  do {bx.set_L a′; return a′}
```

Finally, if $a \not\equiv a'$ and there is no b' such that $((a', b), b') \in fs$, then $lookup\ (a', b)\ fs = Nothing$, and:

```
  do {((a, b), fs, bs) ← get;
        if a ≡ a′ then return () else
           do b′ ← case lookup (a′, b) fs of
                     Just b′   → return b′
                     Nothing → lift (f a′ b)
                 set ((a′, b′), ((a′, b), b′) : fs, bs);
        ((a″, b″), fs″, gs″) ← get; return a″}
=   ⟦ a ≢ a′, lookup (a′, b) = Nothing ⟧
  do {((a, b), fs, bs) ← get;
```

```
            b' ← lift (f a' b);
            set ((a', b'), ((a', b), b') : fs, bs);
            ((a'', b''), fs'', gs'') ← get; return a''}
  =   ⟦ (SG) ⟧
    do {((a, b), fs, bs) ← get;
            b' ← lift (f a' b);
            set ((a', b'), ((a', b), b') : fs, bs);
            return a'}
  =   ⟦ reversing previous steps ⟧
    do {((a, b), fs, bs) ← get
            if a == a' then return () else
               do b' ← case lookup (a', b) fs of
                         Just b'  → return b'
                         Nothing → lift (f a' b)
                     set ((a', b'), ((a', b), b') : fs, bs);
            return a'}
  =   ⟦ definition ⟧
    do {bx.set_L a'; return a'}
```

Therefore, $(\mathrm{S_L G_L})$ holds in all three cases.

For $(\mathrm{G_L S_L})$, an invocation of $bx.get_L$ in a state $((a, b), fs, bs)$ returns a, and by construction a subsequent $bx.set_L\ a$ has no effect.

More formally, we proceed as follows.

```
    do {a ← bx.get_L; bx.set_L a}
  =   ⟦ Definition ⟧
    do {((a, _), _, _) ← get;
            ((a_0, b_0), fs, bs) ← get;
            if a_0 == a then return () else
               do b' ← case lookup (a, b_0) fs of
                         Just b'  → return b'
                         Nothing → lift (f a b)
                     set ((a, b'), ((a, b), b') : fs, bs)}
  =   ⟦ (GG) ⟧
    do {((a_0, b_0), fs, bs) ← get;
            if a_0 == a_0 then return () else
               do b' ← case lookup (a, b_0) fs of
                         Just b'  → return b'
                         Nothing → lift (f a_0 b)
                     set ((a_0, b'), ((a_0, b), b') : fs, bs)}
  =   ⟦ a_0 = a_0 ⟧
    do {((a_0, b_0), fs, bs) ← get;
            return ()}
```

$$= \quad [\![\ \text{(GG)}\ ]\!]$$

return () ☐

G Code

This appendix includes all the code discussed in the paper, along with other convenience definitions that were not discussed in the paper and alternative definitions of e.g. composition that we explored while writing the paper. In this appendix, we have reinstated the standard Haskell use of **newtype**s etc.

G.1 SetBX

```
 {-# LANGUAGE RankNTypes, ImpredicativeTypes #-}
module BX where
import Control.Monad.State as State
import Control.Monad.Reader as Reader
data BX m a b = BX {
   mgetl :: m a,
   msetl :: a → m (),
   mgetr :: m b,
   msetr :: b → m ()
}
mputlr :: Monad m ⇒ BX m a b → a → m b
mputlr bx a = msetl bx a ≫ mgetr bx
mputrl :: Monad m ⇒ BX m a b → b → m a
mputrl bx b = msetr bx b ≫ mgetl bx
```

Identity.

```
idMBX :: MonadState a m ⇒ BX m a a
idMBX = BX get put get put
```

Duality.

```
coMBX :: BX m a b → BX m b a
coMBX bx = BX (mgetr bx) (msetr bx) (mgetl bx) (msetl bx)
```

Natural transformations

```
type f →̇ g = ∀a.f a → g a
type g₁ ←̇ f →̇ g₂ = (f →̇ g₁, f →̇ g₂)
type f₁ →̇ g ←̇ f₂ = (f₁ →̇ g, f₂ →̇ g)
```

```
type NTSquare f g1 g2 h = (g1 ←̇ f →̇ g2, g1 →̇ h ←̇ g2)
```

Composition

```
compMBX :: (m1 →̇ n ←̇ m2) → BX m1 a b → BX m2 b c →
             BX n a c
compMBX (φ, ψ) bx1 bx2 = BX (φ (mgetl bx1))
                            (λa. φ (msetl bx1 a))
                            (ψ (mgetr bx2))
                            (λa. ψ (msetr bx2 a))
```

Variant, assuming monad morphisms *l* and *r*

```
compMBX' :: (Monad m1, Monad m2, Monad n) ⇒
  (m1 →̇ n ←̇ m2) →
  BX m1 a b → BX m2 b c → BX n a c
compMBX' (l, r) bx1 bx2 =
  BX (l (mgetl bx1))
     (λa. do { b ← l (do { msetl bx1 a; mgetr bx1 });
               r (msetl bx2 b) })
     (r (mgetr bx2))
     (λc. do { b ← r (do { msetr bx2 c; mgetl bx2 });
               l (msetr bx1 b) })
```

G.2 Isomorphisms

```
module Iso where
```

Some isomorphisms.

```
data Iso a b = Iso { to :: a → b, from :: b → a }
assocIso :: Iso ((a, b), c) (a, (b, c))
assocIso = Iso (λ((a, b), c). (a, (b, c))) (λ(a, (b, c)). ((a, b), c))
swapIso :: Iso (a, b) (b, a)
swapIso = Iso (λ(a, b). (b, a)) (λ(a, b). (b, a))
unitlIso :: Iso a ((), a)
unitlIso = Iso (λa. ((), a)) (λ((), a). a)
unitrIso :: Iso a (a, ())
unitrIso = Iso (λa. (a, ())) (λ(a, ()). a)
```

G.3 Lenses

```
module Lens where
data Lens a b = Lens { view :: a → b,
                       update :: a → b → a,
                       create :: b → a }
idLens :: Lens a a
idLens = Lens (λa. a) (\_a → a) (λa. a)
compLens :: Lens b c → Lens a b → Lens a c
compLens l2 l1 = Lens (view l2 · view l1)
                      (λa c. update l1 a (update l2 (view l1 a) c))
                      (create l1 · create l2)
fstLens :: b → Lens (a, b) a
fstLens b = Lens fst (λ(a, b) a′. (a′, b)) (λa. (a, b))
sndLens :: a → Lens (a, b) b
sndLens a = Lens snd (λ(a, b) b′. (a, b′)) (λb. (a, b))
```

G.4 Monadic Lenses

```
module MLens where
import Lens
data MLens m a b = MLens { mview :: a → b,
                           mupdate :: a → b → m a,
                           mcreate :: b → m a }
idMLens :: Monad m ⇒ MLens m a a
idMLens = MLens (λa. a) (\_a → return a) (λa. return a)
(;) :: Monad m ⇒ MLens m b c → MLens m a b → MLens m a c
(;) l2 l1 = MLens (mview l2 · mview l1)
                  (λa c. do { b ← mupdate l2 (mview l1 a) c; mupdate l1 a b })
                  (λc. do { b ← mcreate l2 c; mcreate l1 b })
lens2MLens :: Monad m ⇒ Lens a b → MLens m a b
lens2MLens l = MLens (view l)
                     (λa b. return (update l a b))
                     (return · create l)
```

G.5 Relational BX

```
module RelBX where
import Lens
```

pointed types that have a distinguished element

```
class Pointed a where
  point :: a
```

Relational bx

```
data RelBX a b = RelBX {consistent :: a → b → Bool,
                        fwd :: a → b → b,
                        bwd :: a → b → a}
```

Lenses from relational bx

```
lens2rel :: Eq b ⇒ Lens a b → RelBX a b
lens2rel l = RelBX (λa b. view l a == b)
                   (λa b. view l a)
                   (λa b. update l a b)
```

Relational BX form spans of lenses provided types pointed

```
rel2lensSpan :: (Pointed a, Pointed b) ⇒ RelBX a b → (Lens (a, b) a, Lens (a, b) b)
rel2lensSpan bx = (Lens fst
                        (λ(_, b) a. (a, fwd bx a b))
                        (λa. (point, point)),
                   Lens snd
                        (λ(a, _) b. (bwd bx a b, b))
                        (λb. (point, point)))
```

G.6 Symmetric Lenses

```
module SLens where
import Lens
import RelBX
```

Symmetric lenses (with explicit points)

```
data SLens c a b = SLens {putr :: (a, c) → (b, c),
                          putl :: (b, c) → (a, c),
                          missing :: c}
```

Dual

```
dualSL sl = SLens (putl sl) (putr sl) missing
```

From asymmetric lenses

```
lens2symlens :: Lens a b → SLens (Maybe a) a b
lens2symlens l = SLens putr putl Nothing
  where putr (a, _)    = (view l a, Just a)
        putl (b′, ma) = let a′ = case ma of
                                   Nothing → create l b′
                                   a → update l a′ b′
                        in (create l b′, Just a′)
```

From relational bx

```
rel2symlens :: (Pointed a, Pointed b) ⇒ RelBX a b → SLens (a, b) a b
rel2symlens bx = SLens (λ(a′, (a, b)). let b′ = fwd bx a′ b
                                        in (b′, (a′, b′)))
                       (λ(b′, (a, b)). let a′ = bwd bx a b′
                                        in (a′, (a′, b′)))
                       (point, point)
```

To asymmetric lens, on the left...

```
symlens2lensL :: SLens c a b → Lens (a, b, c) a
symlens2lensL sl = Lens (λ(a, b, c). a)
                        (λ(_, _, c). fixup c)
                        (fixup (missing sl))
  where fixup c a′ = let (b′, c′) = putr sl (a′, c) in (a′, b′, c′)
```

...and on the right

```
symlens2lensR :: SLens c a b → Lens (a, b, c) b
symlens2lensR sl = Lens (λ(a, b, c). b)
                        (λ(_, _, c). fixup c)
                        (fixup (missing sl))
  where fixup c b′ = let (a′, c′) = putl sl (b′, c) in (a′, b′, c′)
```

Spans and cospans: used to simplify some definitions.

```
type Span c y1 x y2 = (c x y1, c x y2)
type Cospan c y1 z y2 = (c y1 z, c y2 z)
```

To a span

```
symlens2lensSpan :: SLens c a b → Span Lens a (a, b, c) b
symlens2lensSpan sl = (symlens2lensL sl, symlens2lensR sl)
```

From a span

```
lensSpan2symlens :: Span Lens a c b → SLens (Maybe c) a b
lensSpan2symlens (l1, l2) = SLens (λ(a, mc).
                                      let c' = case mc of Nothing → create l1 a
                                                          Just c → update l1 c a
                                      in (view l2 c', Just c'))
                                    (λ(b, mc).
                                      let c' = case mc of Nothing → create l2 b
                                                          Just c → update l2 c b
                                      in (view l1 c', Just c'))
                                    Nothing
```

G.7 Monadic Symmetric Lenses

```
module SMLens where
import MLens
```

Symmetric lenses (with explicit points)

```
data SMLens m c a b = SMLens {mputr :: (a, c) → m (b, c),
                              mputl :: (b, c) → m (a, c),
                              missing :: c}
```

Dual

```
dualSL sl = SMLens (mputl sl) (mputr sl) missing
```

To asymmetric MLens, on the left...

```
symMLens2MLensL :: Monad m ⇒ SMLens m c a b → MLens m (a, b, c) a
symMLens2MLensL sl = MLens (λ(a, b, c). a)
                           (λ(_, _, c). fixup c)
                           (fixup (missing sl))
  where fixup c a' = do {(b', c') ← mputr sl (a', c); return (a', b', c')}
```

...and on the right

```
symMLens2MLensR :: Monad m ⇒ SMLens m c a b → MLens m (a, b, c) b
symMLens2MLensR sl = MLens (λ(a, b, c). b)
                           (λ(_, _, c). fixup c)
                           (fixup (missing sl))
  where fixup c b' = do {(a', c') ← mputl sl (b', c); return (a', b', c')}
```

Spans and cospans: used to simplify some definitions.

```
type Span c y1 x y2 = (c x y1, c x y2)
type Cospan c y1 z y2 = (c y1 z, c y2 z)
```

To a span

```
symlens2lensSpan :: Monad m ⇒ SMLens m c a b → Span (MLens m) a (a, b, c) b
symlens2lensSpan sl = (symMLens2MLensL sl, symMLens2MLensR sl)
```

and from a span

```
lensSpan2symlens :: Monad m ⇒ Span (MLens m) a c b → SMLens m (Maybe c) a b
lensSpan2symlens (l1, l2)
  = SMLens (λ(a, mc). do c' ← case mc of Nothing → mcreate l1 a
                                         Just c → mupdate l1 c a
                         return (mview l2 c', Just c'))
           (λ(b, mc). do c' ← case mc of Nothing → mcreate l2 b
                                         Just c → mupdate l2 c b
                         return (mview l1 c', Just c'))
           Nothing
```

Composition (naive)

```
(;) :: Monad m ⇒ SMLens m c1 a b → SMLens m c2 b c → SMLens m (c1, c2) a c
(;) sl1 sl2 = SMLens mputR mputL mMissing where
  mputR (a, (c1, c2)) = do (b, c1') ← mputr sl1 (a, c1)
                           (c, c2') ← mputr sl2 (b, c2)
                           return (c, (c1', c2'))
  mputL (c, (c1, c2)) = do (b, c2') ← mputl sl2 (c, c2)
                           (a, c1') ← mputl sl1 (b, c1)
                           return (a, (c1', c2'))
  mMissing            = (missing sl1, missing sl2)
```

G.8 StateTBX

```
 {-# LANGUAGE RankNTypes, FlexibleContexts #-}
module StateTBX where
import Control.Monad.State as State
import Control.Monad.Id as Id
import BX
import Iso
import Lens
import RelBX
import SLens as SLens
```

```
import MLens as MLens
import SMLens as SMLens
```

The interface

```
data StateTBX m s a b = StateTBX {
  getl :: StateT s m a,
  setl :: a → StateT s m (),
  initl :: a → m s,
  getr :: StateT s m b,
  setr :: b → StateT s m (),
  initr :: b → m s
}
```

Variations on initialisation

```
init2run :: Monad m ⇒ (a → m s) → StateT s m x → a → m (x, s)
init2run init m a = do {s ← init a; runStateT m s}
runl :: Monad m ⇒ StateTBX m s a b → StateT s m x → a → m (x, s)
runl bx = init2run (initl bx)
runr :: Monad m ⇒ StateTBX m s a b → StateT s m x → b → m (x, s)
runr bx = init2run (initr bx)
run2init :: Monad m ⇒ (∀x.StateT s m x → a → m (x, s)) → a → m s
run2init run a = do {((), s) ← run (return ()) a; return s}
```

An alternative 'PutBX' or push–pull style

```
put_L^R :: Monad m ⇒ StateTBX m s a b → a → StateT s m b
put_L^R bx a = do {setl bx a; getr bx}
put_R^L :: Monad m ⇒ StateTBX m s a b → b → StateT s m a
put_R^L bx b = do {setr bx b; getl bx}
```

Identity is easy

```
idBX :: Monad m ⇒ StateTBX m a a a
idBX = StateTBX get put return get put return
```

Duality

```
coBX :: StateTBX m s a b → StateTBX m s b a
coBX bx = StateTBX (getr bx) (setr bx) (initr bx)
                   (getl bx) (setl bx) (initl bx)
```

Monad morphisms injecting *StateT s m* (respectively *StateT t m*) into *StateT (s, t) m*.

```
left :: Monad m ⇒ StateT s m a → StateT (s, t) m a
left ma = do {(s, t) ← get;
              (a, s') ← lift (runStateT ma s);
              put (s', t);
              return a}
right :: Monad m ⇒ StateT t m a → StateT (s, t) m a
right ma = do {(s, t) ← get;
               (a, t') ← lift (runStateT ma t);
               put (s, t');
               return a}
```

Composition: given l :: *StateTBX m s a b* and l' :: *StateTBX m s b c*, we want

```
compBX l l' :: StateTBX m (s, t) a c
```

satisfying the monad and bx laws

```
ϑ :: Monad m ⇒ MLens m s v → StateT v m →̇ StateT s m
ϑ l m = do s ← get
           let v = mview l s
           (a, v') ← lift (runStateT m v)
           s' ← lift (mupdate l s v')
           put s'
           return a
```

The m-lenses induced by two composable bxs.

```
mlensL :: Monad m ⇒ StateTBX m s1 a b →
                    StateTBX m s2 b c →
                    MLens m (s1, s2) s1
mlensL bx1 bx2 = MLens view update create where
  view (s1, s2)        = s1
  update (s1, s2) s1'  = do b ← evalStateT (getr bx1) s1'
                            s2' ← execStateT (setl bx2 b) s2
                            return (s1', s2')
  create s1            = do b ← evalStateT (getr bx1) s1
                            s2 ← initl bx2 b
                            return (s1, s2)
mlensR :: Monad m ⇒ StateTBX m s1 a b →
                    StateTBX m s2 b c →
                    MLens m (s1, s2) s2
mlensR bx1 bx2 = MLens view update create where
  view (s1, s2)        = s2
  update (s1, s2) s2'  = do (b, _) ← runStateT (getl bx2) s2'
```

```
                      ((), s1') ← runStateT (setr bx1 b) s1
                      return (s1', s2')
create s2        = do b ← evalStateT (getl bx2) s2
                      s1 ← initr bx1 b
                      return (s1, s2)
```

Composition in terms of m-lenses

```
compBX :: (Monad m) ⇒ StateTBX m s1 a b →
                      StateTBX m s2 b c →
                      StateTBX m (s1, s2) a c
compBX bx1 bx2 =
  StateTBX (φ (getl bx1)) (φ · (setl bx1))
           (λa. do (b, s) ← runl bx1 (getr bx1) a
                   t ← initl bx2 b
                   return (s, t))
           (ψ (getr bx2)) (ψ · (setr bx2))
           (λc. do (b, t) ← runr bx2 (getl bx2) c
                   s ← initr bx1 b
                   return (s, t))
  where φ = ϑ (mlensL bx1 bx2)
        ψ = ϑ (mlensR bx1 bx2)
```

Alternative definition using *left* and *right*

```
compBX' :: (Monad m) ⇒ StateTBX m s a b →
                       StateTBX m t b c →
                       StateTBX m (s, t) a c
compBX' bx1 bx2 =
  StateTBX (left (getl bx1))
           (λa. do b ← left (setl bx1 a ≫ getr bx1)
                   right (setl bx2 b))
           (λa. do (b, s) ← runl bx1 (getr bx1) a
                   t ← initl bx2 b
                   return (s, t))
           (right (getr bx2))
           (λc. do b ← right (setr bx2 c ≫ getl bx2)
                   left (setr bx1 b))
           (λc. do (b, t) ← runr bx2 (getl bx2) c
                   s ← initr bx1 b
                   return (s, t))
```

Direct definition

```
compBX0 :: (Monad m) ⇒ StateTBX m s a b →
                       StateTBX m t b c →
```

```
                                    StateTBX m (s, t) a c
compBX0 bx1 bx2 =
  StateTBX (do {(s, t) ← get; lift (evalStateT (getl bx1) s)})
    (λa. do (s, t) ← get
            s' ← lift (execStateT (setl bx1 a) s)
            b' ← lift (evalStateT (getr bx1) s')
            t' ← lift (execStateT (setl bx2 b') t)
            put (s', t'))
    (λa. do (b, s) ← runl bx1 (getr bx1) a
            t ← initl bx2 b
            return (s, t))
    (do {(s, t) ← get; lift (evalStateT (getr bx2) t)})
    (λc. do (s, t) ← get
            t' ← lift (execStateT (setr bx2 c) t)
            b' ← lift (evalStateT (getl bx2) t')
            s' ← lift (execStateT (setr bx1 b') s)
            put (s', t'))
    (λc. do (b, t) ← runr bx2 (getl bx2) c
            s ← initr bx1 b
            return (s, t))
```

Isomorphisms

```
iso2BX :: Monad m ⇒ Iso a b → StateTBX m a a b
iso2BX iso = StateTBX get put return
                      (do {a ← get; return (to iso a)})
                      (λb. do {put (from iso b)})
                      (λb. return (from iso b))

assocBX :: Monad m ⇒  StateTBX m ((a, b), c) ((a, b), c) (a, (b, c))
assocBX = iso2BX assocIso

swapBX :: Monad m ⇒ StateTBX m (a, b) (a, b) (b, a)
swapBX = iso2BX swapIso

unitlBX :: Monad m ⇒ StateTBX m a a ((), a)
unitlBX = iso2BX unitlIso

unitrBX :: Monad m ⇒ StateTBX m a a (a, ())
unitrBX = iso2BX unitrIso
```

Lenses

```
lens2BX :: Monad m ⇒ Lens a b → StateTBX m a a b
lens2BX l = StateTBX get put return
                     (do {a ← get; return (view l a)})
                     (λb. do {a ← get; put (update l a b)})
```

```
                                  (λb. return (create l b))
lensSpan2BX :: Monad m ⇒ Lens c a → Lens c b → StateTBX m c a b
lensSpan2BX l1 l2 = StateTBX (do c ← get
                                  return (view l1 c))
                             (λa. do c ← get
                                     put (update l1 c a))
                             (λa. return (create l1 a))
                             (do c ← get
                                  return (view l2 c))
                             (λb. do c ← get
                                     put (update l2 c b))
                             (λb. return (create l2 b))
```

Monadic lenses

```
mlens2BX :: Monad m ⇒ MLens m a b → StateTBX m a a b
mlens2BX l = StateTBX get put return view upd create where
  view     = gets (mview l)
  upd b    = do {             a ← get;
                              a′ ← lift (mupdate l a b);
                              put a′}
  create b = mcreate l b
mlensSpan2BX :: Monad m ⇒ MLens m c a → MLens m c b →
                          StateTBX m c a b
mlensSpan2BX l1 l2 = StateTBX viewL updL createL
                              viewR updR createR where
  viewL     = gets (mview l1)
  updL a    = do c ← get
                 c′ ← lift (mupdate l1 c a)
                 put c′
  createL a = mcreate l1 a
  viewR     = gets (mview l2)
  updR b    = do c ← get
                 c′ ← lift (mupdate l2 c b)
                 put c′
  createR b = mcreate l2 b
```

Relational bxs.

```
rel2BX :: (Monad m, Pointed a, Pointed b) ⇒
            RelBX a b → StateTBX m (a, b) a b
rel2BX bx = StateTBX (do {(a, b) ← get; return a})
  (λa′. do {(a, b) ← get; put (a′, fwd bx a′ b)})
```

```
    (λa. return (a, point))
    (do {(a, b) ← get; return b})
    (λb′. do {(a, b) ← get; put (bwd bx a b′, b′)})
    (λb. return (point, b))
```

Symmetric lenses

```
symlens2bx :: Monad m ⇒ SLens c a b → StateTBX m (a, b, c) a b
symlens2bx l = StateTBX (do (a, b, c) ← get
                            return a)
                        (λa′. do (a, b, c) ← get
                                 let (b′, c′) = putr l (a′, c)
                                 put (a′, b′, c′))
                        (λa. do  let (b, c) = putr l (a, missing l)
                                 return (a, b, c))
                        (do (a, b, c) ← get
                            return (b))
                        (λb′. do (a, b, c) ← get
                                 let (a′, c′) = putl l (b′, c)
                                 put (a′, b′, c′))
                        (λb. do  let (a, c) = putl l (b, missing l)
                                 return (a, b, c))
bx2symlens :: StateTBX Id c a b → SLens (Maybe c) a b
bx2symlens bx = SLens (λ(a, mc).
                        let m = (setl bx a ≫ getr bx) in
                        let (b′, c′) =
                          case mc of
                            Nothing → runIdentity (runl bx m a);
                            Just c → runIdentity (runStateT m c)
                        in (b′, Just c′))
                      (λ(b, mc).
                        let m = (setr bx b ≫ getl bx) in
                        let (a′, c′) =
                          case mc of
                            Nothing → runIdentity (runr bx m b);
                            Just c → runIdentity (runStateT m c)
                        in (a′, Just c′))
                      Nothing
```

Monadic symmetric lenses

```
symMLens2bx :: Monad m ⇒ SMLens m c a b → StateTBX m (a, b, c) a b
symMLens2bx l = StateTBX (do (a, b, c) ← get
                             return a)
```

```
                    (λa′. do (a, b, c) ← get
                             (b′, c′) ← lift (mputr l (a′, c))
                             put (a′, b′, c′))
                    (λa. do  (b, c) ← mputr l (a, missing l)
                             return (a, b, c))
                    (do (a, b, c) ← get
                        return b)
                    (λb′. do (a, b, c) ← get
                             (a′, c′) ← lift (mputl l (b′, c))
                             put (a′, b′, c′))
                    (λb. do  (a, c) ← mputl l (b, missing l)
                             return (a, b, c))
bx2symMLens :: Monad m ⇒ StateTBX m c a b →
               SMLens m (Maybe c) a b
bx2symMLens bx = SMLens mputlr mputrl missing where
  mputlr (a, ms) = do s ← case ms of
                            Just s′     → return s′
                            Nothing     → initl bx a
                      (b, s′) ← (runStateT (do {setl bx a; getr bx}) s)
                      return (b, Just s′)
  mputrl (b, ms) = do s ← case ms of
                            Just s′     → return s′
                            Nothing     → initr bx b
                      (a, s′) ← runStateT (do {setr bx b; getl bx}) s
                      return (a, Just s′)
  missing = Nothing
```

Constants

```
constBX :: Monad τ ⇒ α → StateTBX τ α () α
constBX a = StateTBX (return ())
                     (const (return ()))
                     (const (return a))
                     get put return
```

Pairs

```
fstBX :: (Monad m) ⇒ b → StateTBX m (a, b) (a, b) a
fstBX b₀ = StateTBX  (get)
                     (put)
                     return
                     (gets fst)
                     (λa. do (_, b) ← get
                             put (a, b))
```

```
                             (λa. return (a, b0))
sndBX :: Monad m ⇒ a → StateTBX m (a, b) (a, b) b
sndBX a0 = StateTBX (get)
                    (put)
                    return
                    (gets snd)
                    (λb. do (a, _) ← get
                            put (a, b))
                    (λb. return (a0, b))
```

Products

```
pairBX :: Monad m ⇒ StateTBX m s1 a1 b1 →
                    StateTBX m s2 a2 b2 →
                    StateTBX m (s1, s2) (a1, a2) (b1, b2)
pairBX bx1 bx2 = StateTBX (do a1 ← left (getl bx1)
                              a2 ← right (getl bx2)
                              return (a1, a2))
                          (λ(a1, a2). do left (setl bx1 a1)
                                         right (setl bx2 a2))
                          (λ(a1, a2). do s1 ← initl bx1 a1
                                         s2 ← initl bx2 a2
                                         return (s1, s2))
                          (do b1 ← left (getr bx1)
                              b2 ← right (getr bx2)
                              return (b1, b2))
                          (λ(b1, b2). do left (setr bx1 b1)
                                         right (setr bx2 b2))
                          (λ(b1, b2). do s1 ← initr bx1 b1
                                         s2 ← initr bx2 b2
                                         return (s1, s2))
```

Sums

```
inlBX :: Monad m ⇒ x → StateTBX m (x, Maybe y) x (Either x y)
inlBX initX = StateTBX getA setA initA getB setB initB
  where getA            = do {(x, _) ← get; return x}
        getB            = do (x, my) ← get
                             case my of
                               Just y → return (Right y)
                               Nothing → return (Left x)
        setA x'         = do {(x, my) ← get; put (x', my)}
        setB (Left x)   = do {put (x, Nothing)}
        setB (Right y)  = do {(x, _) ← get; put (x, Just y)}
```

```
          initA x          = return (x, Nothing)
          initB (Left x)   = return (x, Nothing)
          initB (Right y)  = return (initX, Just y)
inrBX :: Monad m ⇒ y → StateTBX m (y, Maybe x) y (Either x y)
inrBX initY = StateTBX getA setA initA getB setB initB
  where getA             = do {(y, _) ← get; return y}
        getB             = do (y, mx) ← get
                              case mx of
                                Just x → return (Left x)
                                Nothing → return (Right y)
        setA y'          = do {(y, mx) ← get; put (y', mx)}
        setB (Left x)    = do {(y, _) ← get; put (y, Just x)}
        setB (Right y)   = do {put (y, Nothing)}
        initA y          = return (y, Nothing)
        initB (Left x)   = return (initY, Just x)
        initB (Right y)  = return (y, Nothing)
sumBX :: Monad m ⇒ StateTBX m s1 a1 b1 →
          StateTBX m s2 a2 b2 →
          StateTBX m (Bool, s1, s2) (Either a1 a2) (Either b1 b2)
sumBX bx1 bx2 = StateTBX getA setA initA getB setB initB
  where getA          = do    (b, s1, s2) ← get;
                              if b then
                                do (a1, _) ← lift (runStateT (getl bx1) s1)
                                   return (Left a1)
                              else do (a2, _) ← lift (runStateT (getl bx2) s2)
                                      return (Right a2)
      getB            = do    (b, s1, s2) ← get;
                              if b then
                                do (b1, _) ← lift (runStateT (getr bx1) s1)
                                   return (Left b1)
                              else do (b2, _) ← lift (runStateT (getr bx2) s2)
                                      return (Right b2)
      setA (Left a1)  = do    (b, s1, s2) ← get
                              ((), s1') ← lift (runStateT (setl bx1 a1) s1)
                              put (True, s1', s2)
      setA (Right a2) = do    (b, s1, s2) ← get
                              ((), s2') ← lift (runStateT (setl bx2 a2) s2)
                              put (False, s1, s2')
      setB (Left b1)  = do    (b, s1, s2) ← get
                              ((), s1') ← lift (runStateT (setr bx1 b1) s1)
                              put (True, s1', s2)
      setB (Right b2) = do    (b, s1, s2) ← get
```

```
                                 ((), s_2') ← lift (runStateT (setr bx_2 b_2) s_2)
                                 put (False, s_1, s_2')
    initA (Left a_1)   = do      s_1 ← initl bx_1 a_1
                                 return (True, s_1, ⊥)
    initA (Right a_2)  = do      s_2 ← initl bx_2 a_2
                                 return (False, ⊥, s_2)
    initB (Left b_1)   = do      s_1 ← initr bx_1 b_1
                                 return (True, s_1, ⊥)
    initB (Right b_2)  = do      s_2 ← initr bx_2 b_2
                                 return (False, ⊥, s_2)
```

List

```
listBX :: Monad m ⇒ StateTBX m s a b →
                     StateTBX m (Int, [s]) [a] [b]
listBX bx = StateTBX get_L set_L init_L get_R set_R init_R
  where get_L =      do {(n, cs) ← get;
                         mapM (lift · evalStateT (getl bx)) (take n cs)}
    get_R     =      do {(n, cs) ← get;
                         mapM (lift · evalStateT (getr bx)) (take n cs)}
    set_L as  =      do {(_, cs) ← get;
                         cs' ← sets (setl bx) (initl bx) cs as;
                         put (length as, cs')}
    set_R bs  =      do {(_, cs) ← get;
                         cs' ← sets (setr bx) (initr bx) cs bs;
                         put (length bs, cs')}
    init_L as =      do {cs ← mapM (initl bx) as;
                         return (length as, cs)}
    init_R bs =      do {cs ← mapM (initr bx) bs;
                         return (length bs, cs)}
    sets set init [] []             = return []
    sets set init (c : cs) (x : xs) = do c' ← lift (execStateT (set x) c)
                                         cs' ← sets set init cs xs
                                         return (c' : cs')
    sets set init cs []             = return cs
    sets set init [] xs             = lift (mapM init xs)
```

G.9 Composers example

```
{-# LANGUAGE MultiParamTypeClasses, ScopedTypeVariables #-}

module Composers where
import Data.List as List
```

```
import Data.Set as Set
import Control.Monad.State as State
import Control.Monad.Id
import SLens
import StateTBX
```

Here is a version of the familiar Composers example [5], see the Bx wiki; versions of this have been used in many papers including e.g. the Symmetric Lens paper [13].

Assumption: Name is a key in all our datastructures: the user is required not to give as argument any view that contains more than one element for a given name.

NB We are not saying this version is better than any other version: it's just an illustration.

```
composers :: SLens [(Name, Dates)]
                   (Set (Name, Nation, Dates))
                   [(Name, Nation)]
composers = SLens {putr = putMN, putl = putNM,
                      missing = [ ]}
  where
    putMN (m, c) = (n, c')
      where
        n = selectNNfromNND tripleList
        c' = selectNDfromNND tripleList
        tripleList = h c [ ] (Set.toList m)
        h [ ] sel leftover = reverse sel ++ sort leftover
        h ((nn, _) : cs) ss ls = h cs (ps ++ ss) ns
          where (ps, ns) = selectNNDonKey nn ls
    putNM (n, c) = (m, c')
      where
      m = Set.fromList tripleList
      c' = selectNDfromNND tripleList
      tripleList = k n [ ] c
      k [ ] selected _ = List.reverse selected
      k (h@(nn, _) : nts) ss ls = k nts (newTriple : ss) ns
        where (ps, ns) = selectNDonKey nn ls
          newTriple = newTripleFromList h (λ(_, d). d) ps
```

where the useful 'select' statements are packaged as follows

```
type NND = (Name, Nation, Dates)
selectNNDonKey :: Name → [NND] → ([NND], [NND])
selectNNDonKey n = List.partition (λ(nn, _, _). nn == n)

type ND = (Name, Dates)
selectNDonKey :: Name → [ND] → ([ND], [ND])
selectNDonKey n = List.partition (λ(nn, _). nn == n)
```

```
selectNDfromNND :: [NND] → [ND]
selectNDfromNND = List.map (λ(nn, nt, dd). (nn, dd))
type NN = (Name, Nation)
selectNNfromNND :: [NND] → [NN]
selectNNfromNND = List.map (λ(nn, nt, dd). (nn, nt))
mkNNDfromNN :: [NN] → [NND]
mkNNDfromNN = List.map (λ(nn, nt). (nn, nt, dates0))
```

This last helper function abstracts how to make a new triple

```
newTripleFromList :: NN → (a → Dates) → [a] → NND
newTripleFromList (nn, nt) _ [] = (nn, nt, dates0)
newTripleFromList (nn, nt) f (a: _) = (nn, nt, f a)
```

Now here is the same functionality as a bx. There are no effects other than the state ones induced directly by the BX, so the underlying monad is the Identity monad.

```
composersBx :: StateTBX Id
                         [(Name, Nation, Dates)]
                         (Set (Name, Nation, Dates))
                         [(Name, Nation)]
composersBx = StateTBX getl setl initl getr setr initr
  where
    getl = state (λl. (Set.fromList l, l))
    setl = (λm. state (λl. ((), f (Set.toList m) [] l)))
    initl = (λm. return (Set.toList m))
    f leftovers upd [] = (reverse upd) ++ sort leftovers
    f leftovers upd ((nn, na, nd) : rs) = f ns (ps ++ upd) rs
      where (ps, ns) = selectNNDonKey nn leftovers
    getr = state (λl. (selectNNfromNND l, l))
    setr = (λn. state (λl. ((), g n [] l)))
    initr = (λn. return (mkNNDfromNN n))
    g [] updated stateNotSeenYet = reverse updated
    g (h@(nn, na) : todo) updated stateNotSeenYet =
      g todo (newTriple : updated) ns
      where (ps, ns) = selectNNDonKey nn stateNotSeenYet
        newTriple = newTripleFromList h (λ(_, _, d). d) ps
```

Now let's see how to use both the symmetric lens and the bx versions, and demonstrate them behaving the same.

1. Initialise both with no composers at all.

   ```
   (m₁, c₁) = putl composers ([], missing composers)
   s₁ = runIdentity (initr composersBx [])
   ```

2. Now suppose the owner of the left-hand view likes JS Bach.

   ```
   jsBach = (Name "J. S. Bach", Nation "German",
             Dates (Just (Date "1685", Date "1750")))
   onlyBach = Set.fromList ([jsBach])
   ```

   Putting this into the symmetric lens version:

   ```
   (n1sl, c2) = putr composers (onlyBach, c1)
   ```

   and into the bx version (the underscore represents the result of the monadic computation; we could use *evalState* if we didn't like it, but this is just standard Haskell-monad-cruft, nothing to do with our formalism specifically:

   ```
   (_, s2) = runStaten (do {setl composersBx onlyBach}) s1
   ```

3. Let's check that what the owner of the right-hand view sees is the same in both cases. n1 is that, for the symmetric lens (we got told, whether we liked it or not). For the bx:

   ```
   (n1bx, s3) = runStaten (do {n ← getr composersBx; return n}) s2
   ok1 = (n1sl == n1bx)
   ```

4. The RH view owner also likes John Tavener:

   ```
   johnTavener = (Name "John Tavener", Nation "British")
   ```

   and decides to append:

   ```
   bachTavener = n1sl ++ [johnTavener]
   ```

   Putting this into the symmetric lens version:

   ```
   (m1sl, c3) = putl composers (bachTavener, c2)
   ```

   and into the bx version:

   ```
   (_, s4) = runStaten (do {setr composersBx bachTavener}) s3
   ```

   yields the same result for the LH view owner:

   ```
   (m1bx, s5) = runStaten (do {m ← getl composersBx; return m}) s4
   ok2 = (m1sl == m1bx)
   ```

   (Note that Haskell's Set equality compares the contents of Sets ignoring multiplicity and order.)

5. The LH owner looks up Tavener's dates:

$$\mathit{datesJT} = \mathit{Dates}\;(\mathit{Just}\;(\mathit{Date}\;\texttt{"1944"}, \mathit{Date}\;\texttt{"2013"}))$$

   and fixes their view:

$$\begin{aligned}
&(\mathit{yesJT}, \mathit{noJT}) = \mathsf{Set}.\mathit{partition}\;(\lambda(\mathit{nn}, \mathit{na}, \mathit{dd}).\,\mathit{nn} == \mathit{Name}\;\texttt{"John Tavener"})\;\mathit{m1sl}\\
&\mathit{fixedYesJT} = \mathsf{Set}.\mathit{map}\;(\lambda(\mathit{nn}, \mathit{na}, \mathit{dd}).\,(\mathit{nn}, \mathit{na}, \mathit{datesJT}))\;\mathit{yesJT}\\
&m_2 = \mathsf{Set}.\mathit{union}\;\mathit{noJT}\;\mathit{fixedYesJT}
\end{aligned}$$

   and puts it back in the symmetric lens version:

$$(\mathit{n2sl}, \mathit{c4}) = \mathit{putr}\;\mathit{composers}\;(m_2, \mathit{c3})$$

   and in the bx version:

$$(\_, \mathit{s6}) = \mathit{runStaten}\;(\mathbf{do}\;\{\mathit{setl}\;\mathit{composersBx}\;m_2\})\;\mathit{s5}$$

   Checking result from the other side:

$$\begin{aligned}
&(\mathit{n2bx}, \mathit{s7}) = \mathit{runStaten}\;(\mathbf{do}\;\{n \leftarrow \mathit{getr}\;\mathit{composersBx}; \mathit{return}\;n\})\;\mathit{s6}\\
&\mathit{ok3} = (\mathit{n2sl} == \mathit{n2bx})
\end{aligned}$$

6. Back to the RH view owner

$$\begin{aligned}
&\mathit{n3} = ((\mathit{Name}\;\texttt{"Hendrik Andriessen"}, \mathit{Nation}\;\texttt{"Dutch"}) : \mathit{n2sl})\\
&\qquad +\!\!+\;[(\mathit{Name}\;\texttt{"J-B Lully"}, \mathit{Nation}\;\texttt{"French"})]\\
&(\mathit{m3sl}, \mathit{c5}) = \mathit{putl}\;\mathit{composers}\;(\mathit{n3}, \mathit{c4})\\
&(\_, \mathit{s8}) = \mathit{runStaten}\;(\mathbf{do}\;\{\mathit{setr}\;\mathit{composersBx}\;\mathit{n3}\})\;\mathit{s6}
\end{aligned}$$

To note:

- we have shown alternating sets on the two sides, as that is natural for symmetric lenses; for bx, any order of sets works equally well (and there is no need to wonder about what complement to use).
- We've shown the fine-grained version to facilitate comparison, but we can also combine steps:

$$\begin{aligned}
&(n', s') = \mathit{runIdentity}\;(\mathit{runr}\;\mathit{composersBx}\;(\mathbf{do}\;\{\\
&\quad \mathit{setl}\;\mathit{composersBx}\;\mathit{onlyBach};\\
&\quad n \leftarrow \mathit{getr}\;\mathit{composersBx};\\
&\quad \mathit{return}\;n\})[\,])
\end{aligned}$$

  etc.

Auxiliary definitions; only the *Show* instance for Dates is noteworthy

```
newtype Name = Name {unName :: String}
  deriving (Eq, Ord)
instance Show Name where
  show (Name n) = n
newtype Nation = Nation {unNation :: String}
  deriving (Eq, Ord)
instance Show Nation where
  show (Nation n) = n
newtype Date = Date {unDate :: String}
  deriving (Eq, Ord)
instance Show Date where
  show (Date d) = d
newtype Dates = Dates {unDates :: Maybe (Date, Date)}
  deriving (Eq, Ord)
instance Show Dates where
  show (Dates d) = h d
    where h Nothing = "????"
      h (Just (dob, dod)) = show dob ++ "--" ++ show dod
dates0 :: Dates
dates0 = Dates Nothing
```

G.10 Examples

```
{-# LANGUAGE RankNTypes, FlexibleContexts #-}
module Examples where
import Control.Monad.State as State
import Control.Monad.Reader as Reader
import Control.Monad.Writer as Writer
import Control.Monad.Id as Id
import Data.Map as Map (Map)
import BX
import StateTBX
```

Failure.

```
inv :: StateTBX Maybe Float Float Float
inv = StateTBX get setL initL (gets (λa. 1 / a)) setR initR
  where setL a   = try put a
        setR b   = try put (1 / b)
```

```
        try m a = if a ≠ 0.0 then m a else lift Nothing
        init_L a  = if a ≠ 0.0 then Just a else Nothing
        init_R a  = if a ≠ 0.0 then Just (1 / a) else Nothing
```

A generalization

```
  divZeroBX :: (Fractional a, Eq a, Monad m) ⇒ (∀x.m x) → StateTBX m a a a
  divZeroBX divZero = StateTBX get set_L init_L (gets (λa. (1 / a))) set_R init_R
    where
      set_L a   = do {lift (guard (a ≠ 0)); put a}
      set_R b   = do {lift (guard (b ≠ 0)); put (1 / b)}
      init_L a  = do {guard (a ≠ 0); return a}
      init_R b  = do {guard (b ≠ 0); return (1 / b)}
      guard b = if b then return () else divZero
```

Uses *readS* to trap errors

```
  readSome :: (Read a, Show a) ⇒
                StateTBX Maybe (a, String) a String
  readSome = StateTBX (gets fst) set_L init_L (gets snd) set_R init_R
    where set_L a′ = put (a′, show a′)
          set_R b′ = do (_, b) ← get
                        if b ≠ b′ then return ()
                        else case reads b of
                          ((a′, ""): _) → put (a′, b)
                          _ → lift Nothing
          init_L a = return (a, show a)
          init_R b = do a ← choices [a | (a, "") ← reads b]
                        return (a, b)
          choices [] = mzero
          choices (a : as) = return a ‘mplus‘ choices as
```

A generalization

```
  readableBX :: (Read a, Show a, MonadPlus m) ⇒
                  StateTBX m (a, String) a String
  readableBX = StateTBX get_A set_A initA get_B set_B initB
    where get_A    = do {(a, s) ← get; return a}
          get_B    = do {(a, s) ← get; return s}
          set_A a′ = put (a′, show a′)
          set_B b′ = do (_, b) ← get
                        case ((b == b′), reads b) of
                          (True, _) → return ()
                          (_, (a′, ""): _) → put (a′, b)
```

```
                   _ → lift mzero
      initA a = return (a, show a)
      initB b = case reads b of
        (a, ""): _ → return (a, b)
        _ → mzero
```

The JTL example from the paper (Example 1)

```
nondetBX :: (a → b → Bool) → (a → [b]) → (b → [a]) → StateTBX [ ] (a, b) a b
nondetBX ok bs as = StateTBX getL setL initL getR setR initR where
  getL    = do {(a, b) ← get; return a}
  getR    = do {(a, b) ← get; return b}
  setL a′ = do (a, b) ← get
               if ok a′ b then put (a′, b) else
                 do {b′ ← lift (bs a′); put (a′, b′)}
  setR b′ = do (a, b) ← get
               if ok a b′ then put (a, b′) else
                 do {a′ ← lift (as b′); put (a′, b′)}
  initL a = do {b ← bs a; return (a, b)}
  initR b = do {a ← as b; return (a, b)}
```

Switching between two lenses on the same state space, based on a boolean flag

```
switchBX :: MonadReader Bool m ⇒ StateTBX m s a b →
                                 StateTBX m s a b →
                                 StateTBX m s a b
switchBX bx1 bx2 = StateTBX getA setA initA getB setB initB
  where getA        = switch (getl bx1) (getl bx2)
    getB            = switch (getr bx2) (getr bx2)
    setA a          = switch (setl bx1 a) (setl bx2 a)
    setB b          = switch (setr bx1 b) (setr bx2 b)
    initA a         = switch (initl bx1 a) (initl bx2 a)
    initB b         = switch (initr bx1 b) (initr bx2 b)
    switch m1 m2 = do {b ← ask; if b then m1 else m2}
```

Generalized version

```
switchBX′ :: MonadReader c m ⇒ (c → StateTBX m s a b) →
                               StateTBX m s a b
switchBX′ f = StateTBX getA setA initA getB setB initB
  where getA = switch getl
    getB     = switch getr
    setA a   = switch (λbx. setl bx a)
    setB b   = switch (λbx. setr bx b)
```

```
initA a     = switch (λbx. initl bx a)
initB b     = switch (λbx. initr bx b)
switch op = do { c ← ask; op (f c)}
```

Logging BX: writes the sequence of sets

```
loggingBX :: (Eq a, Eq b, MonadWriter (Either a b) m) ⇒
               StateTBX m s a b → StateTBX m s a b
loggingBX bx = StateTBX (getl bx) set_A (initl bx) (getr bx) set_B (initr bx)
  where set_A a′ = do a ← getl bx
                      if a ≢ a′ then tell (Left a′) else return ()
                      setl bx a′
        set_B b′ = do b ← getr bx
                      if b ≢ b′ then tell (Right b′) else return ()
                      setr bx b′
```

I/O: user interaction

```
interactiveBX :: (Read a, Read b) ⇒
                   (a → b → Bool) → StateTBX IO (a, b) a b
interactiveBX r = StateTBX get_A set_A initA get_B set_B initB
  where get_A    = do {(a, b) ← get; return a}
        set_A a′  = do {(a, b) ← get; fixB a′ b}
        fixA a b = if   r a b
                   then put (a, b)
                   else do {a′ ← lift (initA b); fixA a′ b}
        initA b  = do print "Please restore consistency:"
                      str ← getLine
                      return (read str)
        get_B    = do {(a, b) ← get; return b}
        set_B b′  = do {(a, b) ← get; fixA a b′}
        fixB a b = if   r a b
                   then put (a, b)
                   else do {b′ ← lift (initB b); fixA a b′}
        initB a  = do print "Please restore consistency:"
                      str ← getLine
                      return (read str)
```

and signalling changes

```
signalBX :: (Eq a, Eq b, Monad m) ⇒
              (a → m ()) → (b → m ()) →
              StateTBX m s a b → StateTBX m s a b
signalBX sigA sigB t = StateTBX (getl t) set_L (initl t)
```

```
                                  (getr t) setR (initr t) where
    setL a' = do {     a ← getl t; setl t a';
                       lift (if a ≢ a' then sigA a' else return ())}
    setR b' = do {     b ← getr t; setr t b';
                       lift (if b ≢ b' then sigB b' else return ())}
  alertBX :: (Eq a, Eq b) ⇒ StateTBX IO s a b → StateTBX IO s a b
  alertBX = signalBX (\_ → putStrLn "Left")
                     (\_ → putStrLn "Right")
```

where

```
  fst3 (a, _, _) = a
  snd3 (_, a, _) = a
  thd3 (_, _, a) = a
```

Model-transformation-by-example (Example 2 from the paper)

```
  dynamicBX' :: (Eq α, Eq β, Monad τ) ⇒
                (α → β → τ β) → (α → β → τ α) →
                StateTBX τ ((α, β), [((α, β), β)], [((α, β), α)]) α β
  dynamicBX' f g = StateTBX (gets (fst · fst3)) setL ⊥
                            (gets (snd · fst3)) setR ⊥
    where setL a' = do ((a, b), fs, bs) ← get
                       b' ← case lookup (a', b) fs of
                          Just b'  → return b'
                          Nothing → lift (f a' b)
                       put ((a', b'), ((a', b), b') : ((a', b'), b') : fs, ((a', b'), a') : bs)
          setR b' = do ((a, b), fs, bs) ← get
                       a' ← case lookup (a, b') bs of
                          Just a'  → return a'
                          Nothing → lift (g a b')
                       put ((a', b'), ((a', b'), b') : fs, ((a, b'), a') : ((a', b'), a') : bs)
```

Some test cases

```
  l0 :: Monad m ⇒ b → c →
       StateTBX m ((a, b), (c, a)) (a, b) (c, a)
  l0 b c = fstBX b `compBX` coBX (sndBX c)
  l = l0 "b" "c"
  foo = runl l (do setr l ("x", "y")
                   (a, b) ← getl l
                   lift (print a)
                   lift (print b)
                   setl l ("z", "w")
```

```
                (c, d) ← getr l
                lift (print c)
                lift (print d)) ("a", "b")


l′ :: (Read a, Ord a) ⇒ StateTBX IO (a, a) a a
l′ = interactiveBX (⟨⟩)
bar = runStateT (do setr l′ "abc"
                    a ← getl l′
                    lift (print a)
                    setl l′ "def"
                    b ← getr l′
                    lift (print b))
                ("abc", "xyz")


baz = let bx = (divZeroBX (fail "divZero"))
   in runl bx (do setr bx 17.0
                  a ← getl bx
                  lift (print a)
                  setl bx 42.0
                  a ← getr bx
                  lift (print a)
                  setl bx 0.0
                  lift (print "foo")) 1.0


xyzzy = let bx = listBX (divZeroBX (fail "divZero"))
   in runl bx (do b ← getr bx
                  lift (print b)
                  setr bx [5.0, 6.0, 7.0, 8.0]
                  a ← getl bx
                  lift (print a))
              [1.0, 2.0, 3.0]
```