Reflections on monadic lenses

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

[1] University of Oxford, `firstname.lastname@cs.ox.ac.uk`
[2] University of Edinburgh, `firstname.lastname@ed.ac.uk`

Abstract. Bidirectional transformations (bx) have primarily been modeled as pure functions, and do not account for the possibility of the side-effects that are available in most programming languages. Recently several formulations of bx that use monads to account for effects have been proposed, both among practitioners and in academic research. The combination of bx with effects turns out to be surprisingly subtle, leading to problems with some of these proposals and increasing the complexity of others. This paper reviews the proposals for monadic lenses to date, and offers some improved definitions, paying particular attention to the obstacles to naively adding monadic effects to existing definitions of pure bx such as lenses and symmetric lenses, and the subtleties of equivalence of symmetric bidirectional transformations in the presence of effects.

1 Introduction

Programming with multiple concrete representations of the same conceptual information is a commonplace, and challenging, problem. It is commonplace because data is everywhere, and not all of it is relevant or appropriate for every task: for example, one may want to work with only a subset of one's full email account on a mobile phone or other low-bandwidth device. It is challenging because the most direct approach to mapping data across sources A and B is to write separate functions, one mapping to B and one to A, following some (not always explicit) specification of what it means for an A value and a B value to be *consistent*. Keeping these transformations coherent with each other, and with the specification, is a considerable maintenance burden, yet it remains the main approach found in practice.

Over the past decade, a number of promising proposals for easing programming such *bidirectional transformations* have emerged, including *lenses* [6], bx based on consistency relations [14], *symmetric lenses* [8], and a number of variants and extensions (e.g. [12, 9]). Most of these proposals consist of an interface with pure functions and some equational laws that characterize good behavior; the interaction of bidirectionality with other effects has received comparatively little attention.

Some programmers and researchers have already proposed ways to combine elenses and monadic effects [4, 12]. Recently, we have proposed symmetric notions of bidirectional computation based on *entangled state monads* [3, 1] and *coalgebras* [2]. As a result, there are now several alternative proposals for bidirectional transformations with effects, and the relationships among them are not well understood.

In this paper we summarize and compare the existing proposals, offer some new alternatives, and attempt to provide general and useful definitions of "monadic lenses" and "symmetric monadic lenses". Perhaps surprisingly, it appears challenging even to define the composition of lenses in the presence of effects, especially in the symmetric case. We first review the definition of pure lenses and two prior proposals for extending them with monadic effects. These definitions have some limitations, and we propose a new definition of monadic lens that overcomes them.

Next we consider the symmetric case. The effectful bx and coalgebraic bx in our previous work are symmetric, but their definitions rely on relatively heavyweight machinery (monad transformers and morphisms, coalgebra). It seems natural to ask whether just adding monadic effects to symmetric lenses in the style of Hofmann et al. would also work. We show that, as for asymmetric lenses, adding monadic effects to symmetric lenses is challenging, and give examples illustrating the problems with the most obvious generalization. We then briefly discuss our recent work on symmetric forms of bx with monadic effects [3, 1, 2]. Defining composition for these approaches also turns out to be tricky, and our definition of monadic lenses arose out of exploring this space. The essence of composition of symmetric monadic bx, we now believe, can be presented most easily in terms of monadic lenses, by considering *spans*, an approach also advocated (in the pure case) by Johnson and Rosebrugh [9].

Symmetric pure bx need to be equipped with a notion of equivalence, to abstract away inessential differences of representation of their "state" or "complement" spaces. As noted by Hofmann et al. [8] and Johnson and Rosebrugh [9], isomorphism of state spaces is unsatisfactory, and there are competing proposals for equivalence of symmetric lenses and spans. In the case of spans of monadic lenses, the right notion of equivalence seem even less obvious. We compare three, increasingly coarse, equivalences of spans based on isomorphism (following [1]), span equivalence (following [9], and bisimulation (following [8, 2]). In addition, we show a (we think surprising) result: in the pure case, span equivalence and bisimulation equivalence coincide.

In this paper we employ Haskell-like notation to describe and compare formalisms, with a few conventions: we write function composition $f \cdot g$ with a centred dot, and use a lowered dot for field lookup $x.f$, in contrast to Haskell's notation $f\ x$. Throughout the paper, we introduce a number of different representations of lenses, and rather than pedantically disambiguating them all, we freely redefine identifiers as we go. We assume familiarity with common uses of monads in Haskell to encapsulate effects (following Wadler [17]), and with the **do**-notation (following Wadler's monad comprehensions [16]).

2 Asymmetric monadic lenses

Recall that a *lens* [6, 7] is a pair of functions, usually called *get* and *put*:

$$\textbf{data}\ \alpha \rightsquigarrow \beta = \mathit{Lens}\ \{ \mathit{get} :: \alpha \to \beta, \mathit{put} :: \alpha \to \beta \to \alpha \}$$

satisfying (at least) the following *well-behavedness* laws:

$$\begin{array}{ll} (\mathsf{GetPut}) & \mathit{put}\ a\ (\mathit{get}\ a) = a \\ (\mathsf{PutGet}) & \mathit{get}\ (\mathit{put}\ a\ b) = b \end{array}$$

The idea is that a lens of type $S \rightsquigarrow V$ maintains a source of type S, providing a view of type V onto it; the well-behavedness laws capture the intuition that the view faithfully reflects the source: if we "get" a b from a source a and then "put" the same b value back into a, this leaves a unchanged; and if we "put" a b into a source a and then "get" from the result, we get b itself. Lenses have been investigated extensively; see for example Foster et al. [7] for a recent tutorial overview. For the purposes of this paper, we just recall the definition of *composition* of lenses:

$$\begin{array}{l}(;) :: (\alpha \rightsquigarrow \beta) \to (\beta \rightsquigarrow \gamma) \to (\alpha \rightsquigarrow \gamma)\\ l_1\,; l_2 = \mathit{Lens}\;(l_2.\mathit{get} \cdot l_1.\mathit{get})\;(\lambda a\; c \to l_1.\mathit{put}\; a\;(l_2.\mathit{put}\;(l_1.\mathit{get}\; a)\; c))\end{array}$$

which preserves well-behavedness. Finally, lenses are often equipped with a *create* function

$$\textbf{data}\;\alpha \rightsquigarrow \beta = \mathit{Lens}\;\{\mathit{get} :: \alpha \to \beta, \mathit{put} :: \alpha \to \beta \to \alpha, \mathit{create} :: \beta \to \alpha\}$$

satisfying an additional law:

$$(\mathsf{CreateGet}) \quad \mathit{get}\;(\mathit{create}\; b) = b$$

2.1 A naive approach

As a first attempt, consider simply adding a monadic effect μ to the result types of *get* and *put*.

$$\textbf{data}\;[\alpha \rightsquigarrow_0 \beta]_\mu = \mathit{MLens}_0\;\{\mathit{mget} :: \alpha \to \mu\;\beta, \mathit{mput} :: \alpha \to \beta \to \mu\;\alpha\}$$

Such an approach has been considered and discussed in some recent Haskell libraries and online discussions [4]. A natural question arises immediately: what laws should a lens $l :: [S \rightsquigarrow_0 V]_M$ satisfy? The following generalizations of the laws appear natural:

$$\begin{array}{lll}(\mathsf{MGetPut_0}) & \textbf{do}\;\{b \leftarrow \mathit{mget}\;a; \mathit{mput}\;a\;b\} & = \mathit{return}\;a\\ (\mathsf{MPutGet_0}) & \textbf{do}\;\{a' \leftarrow \mathit{mput}\;a\;b; \mathit{mget}\;a'\} & = \textbf{do}\;\{a' \leftarrow \mathit{mput}\;a\;b; \mathit{return}\;b\}\end{array}$$

that is, if we "get" b from a and then "put" the same b value back into a, this has the same effect as just returning a (and doing nothing else), and if we "put" a value b and then "get" the result, this has the same effect as just returning b after doing the "put". The obvious generalization of composition from the pure case for these operations is:

$$\begin{array}{l}(;) :: [\alpha \rightsquigarrow_0 \beta]_\mu \to [\beta \rightsquigarrow_0 \gamma]_\mu \to [\alpha \rightsquigarrow_0 \gamma]_\mu\\ l_1\,; l_2 = \mathit{MLens}_0\;(\lambda a \to \textbf{do}\;\{b \leftarrow l_1.\mathit{mget}\;a; l_2.\mathit{mget}\;b\})\\ \qquad\qquad\qquad\qquad (\lambda a\;c \to \textbf{do}\;\{b \leftarrow l_1.\mathit{mget}\;a; b' \leftarrow l_2.\mathit{mput}\;b\;c; l_1.\mathit{mput}\;a\;b'\})\end{array}$$

This proposal has at least two apparent problems. First, the $(\mathsf{MGetPut_0})$ law appears to sharply constrain *mget*: indeed, if *mget a* has an irreversible side-effect then $(\mathsf{MGetPut})$ cannot hold. This suggests that *mget* must either be pure, or have side-effects that are reversible by *mput*, ruling out behaviors such as performing I/O during *mget*. Second, it appears difficult to compose these structures in a way that preserves the laws, unless we again make fairly draconian assumptions about μ. In order to show $(\mathsf{MGetPut_0})$ for the composition $l_1\,;l_2$, it seems necessary to be able to commute $l_2.\mathit{mget}$ with $l_1.\mathit{mget}$ and we also need to know that doing $l_1.\mathit{mget}$ twice is the same as doing it just once. Likewise, to show $(\mathsf{MPutGet_0})$ we need to commute $l_2.\mathit{mget}$ with $l_1.\mathit{mput}$.

2.2 Monadic put-lenses

Pacheco et al. [12] proposed a variant of lenses called *monadic putback-oriented lenses*. For the purposes of this paper, the putback-orientation of their approach is irrelevant: we focus on their use of monads, and we provide a slightly simplified version of their definition:

$$\textbf{data}\ [\alpha \rightsquigarrow_1 \beta]_\mu = \mathit{MLens}_1\ \{\mathit{mget} :: \alpha \to \beta, \mathit{mput} :: \alpha \to \beta \to \mu\ \alpha\}$$

(The main difference from their version is that we remove the *Maybe* type constructors from the return type of *mget* and the first argument of *mput*.) Pacheco et al. state laws for these monadic lenses. First, they assume that the monad μ has a *monad membership* operation

$$(\in) :: \alpha \to \mu\ \alpha \to \mathit{Bool}$$

satisfying the following two laws:

$$\begin{array}{ll} (\in\text{-ID}) & x \in \mathit{return}\ x \quad \Leftrightarrow \mathit{True} \\ (\in\text{-}\gg\!=) & y \in (m \gg\!= f) \Leftrightarrow \exists x\ .\ x \in m \wedge y \in (f\ x) \end{array}$$

Then the laws for MLens_1 (taken from [12] Prop. 3, p49) are as follows:

$$\begin{array}{ll} (\mathsf{MGetPut}_1) & v = \mathit{mget}\ s \qquad \Longrightarrow \mathit{mput}\ s\ v = \mathit{return}\ s \\ (\mathsf{MPutGet}_1) & s' \in \mathit{mput}\ s\ v' \Longrightarrow v' = \mathit{mget}\ s' \end{array}$$

(In the first law we correct an apparent typo in the original paper, as well as removing the *Just* constructors from both laws.) By making *mget* pure, this definition avoids the immediate problems with composition discussed above, and Pacheco et al. outline a proof that their laws are preserved by composition. However, it is not obvious how to generalize their approach beyond monads that admit a sensible $\in$ operation.

Many interesting monads do have a sensible $\in$ operation (e.g. *Maybe*, $[\,]$). Pacheco et al. suggest that $\in$ can be defined for any monad as $x \in m \equiv (\exists h : h\,m = x)$, where h is what they call a '(polymorphic) algebra' for m, that is, a polymorphic function $h :: m\ a \to a$. However, this definition doesn't appear satisfactory for monads such as *IO*, for which there is no such (pure) function: the ($\in$-ID) law can never hold in this case. It is not clear that we can define a useful $\in$ operation directly for *IO* either: given that $m :: \mathit{IO}\ s$ could ultimately return any a-value, it seems safe (if perhaps overly conservative) to define $x \in m = \mathit{True}$ for any x and m. This satisfies the $\in$ laws (at least, if we make a simplifying assumption that all types are inhabited), and indeed, it seems to be the only thing we could write in Haskell that would satisfy the laws, since we have no way of looking inside the monadic computation $m :: \mathit{IO}\ s$ to find out what its eventual return value is. But then the ($\mathsf{MPutGet}_1$) law, whose precondition is always true, forces the view space to be trivial. These complications suggest, at least, that it would be advantageous to find a definition of monadic lenses that makes sense (and is preserved under composition) for any monad.

2.3 Monadic lenses

We propose the following definition of monadic lens:

Definition 2.1 (monadic lens). A *monadic lens* from source type S to view type V in which the put operation may have effects from monad M (or 'M-lens from S to V'), is represented by the type $[S \rightsquigarrow V]_M$, where

$$\mathbf{data}\ [\alpha \rightsquigarrow \beta]_\mu = \mathit{MLens}\ \{\mathit{mget} :: \alpha \to \beta, \mathit{mput} :: \alpha \to \beta \to \mu\ \alpha\}$$

(dropping the μ from the return type of *mget*, compared to the definition in Section 2.1). We say that M-lens l is *well-behaved* if it satisfies

$$\begin{array}{ll} (\mathsf{MGetPut}) & \mathbf{do}\ \{l.\mathit{mput}\ s\ (l.\mathit{mget}\ s)\} = \mathit{return}\ s \\ (\mathsf{MPutGet}) & \mathbf{do}\ \{s' \leftarrow l.\mathit{mput}\ s\ v; k\ s'\ (l.\mathit{mget}\ s')\} = \mathbf{do}\ \{s' \leftarrow l.\mathit{mput}\ s\ v; k\ s'\ v\} \end{array} \diamondsuit$$

Note that in (MPutGet), we use a continuation $k :: \beta \to \alpha \to \mu\ \gamma$ to quantify over all possible subsequent computations in which s' and $l.\mathit{mget}\ s'$ might appear. In fact, using the laws of monads and simply-typed lambda calculus we can prove this law from just the special case $k = \lambda b\ a \to \mathit{return}\ (b, a)$, so in the sequel when we prove (MPutGet) we may just prove this case while using the strong form freely in the proof.

The ordinary asymmetric lenses are exactly the monadic lenses over $\mu = \mathit{Id}$; the laws then specialise to the standard equational laws.

Definition 2.2. We can also define an operation that lifts a pure lens to a monadic lens:

$$\begin{array}{l} \mathit{lens2mlens} :: \mathit{Monad}\ \mu \Rightarrow \alpha \rightsquigarrow \beta \to [\alpha \rightsquigarrow \beta]_\mu \\ \mathit{lens2mlens}\ l = \mathit{MLens}\ (l.\mathit{get})\ (\lambda a\ b \to \mathit{return}\ (l.\mathit{put}\ a\ b)) \end{array} \diamondsuit$$

Lemma 2.3. If $l :: \mathit{Lens}\ \alpha\ \beta$ is well-behaved, then so is *lens2mlens l*. ♢

Example 2.4. To illustrate, some simple pure lenses include:

$$\begin{array}{l} \mathit{idLens} :: \alpha \rightsquigarrow \alpha \\ \mathit{idLens} = \mathit{Lens}\ (\lambda a \to a)\ (\lambda\_\ a \to a) \\ \mathit{fstLens} :: (\alpha, \beta) \rightsquigarrow \alpha \\ \mathit{fstLens} = \mathit{MLens}\ \mathit{fst}\ (\lambda (s_1, s_2)\ s_1' \to (s_1', s_2)) \\ \mathit{sndLens} :: (\alpha, \beta) \rightsquigarrow \beta \\ \mathit{sndLens} = \mathit{Lens}\ \mathit{snd}\ (\lambda (s_1, s_2)\ s_2' \to (s_1, s_2')) \end{array}$$

Many more examples of pure lenses are to be found in the literature [6, 7], all of which lift to well-behaved monadic lenses. ♢

As more interesting examples, we present asymmetric versions of the partial and logging lenses presented by Abou-Saleh et al. [1]. Pure lenses (as usually defined) are total, which means that *get* must be surjective and *put* must be defined for all source and view pairs. One way to accommodate partiality is to adjust the return type of *get* to *Maybe b* or give *put* the return type *Maybe a* to allow for failure if we attempt to put a *b*-value that is not in the range of *get*. In either case, the laws need to be adjusted somehow. Monadic lenses allow for partiality without requiring such an ad hoc change. A trivial example is

$$\begin{array}{l} \mathit{constMLens} :: b \to [a \leadsto b]_{\mathit{Maybe}} \\ \mathit{constMLens}\ b = \mathit{MLens}\ (\mathit{const}\ b)\ (\lambda a\ b' \to \mathbf{if}\ b == b'\ \mathbf{then}\ \mathit{Just}\ a\ \mathbf{else}\ \mathit{Nothing}) \end{array}$$

which is well-behaved (since both sides of (MPutGet) fail if the view is changed to a value different from b). Of course, this example also illustrates that the *mget* function of a monadic lens need not be surjective.

As a more interesting example, consider:

$$\begin{array}{l} \mathit{absLens} :: [\mathit{Int} \leadsto \mathit{Int}]_{\mathit{Maybe}} \\ \mathit{absLens} = \mathit{MLens}\ \mathit{abs} \\ \qquad\qquad (\lambda a\ b \to \mathbf{if}\ b < 0\ \mathbf{then}\ \mathit{Nothing}\ \mathbf{else}\ \mathit{Just}\ (\mathbf{if}\ a < 0\ \mathbf{then}\ -b\ \mathbf{else}\ b)) \end{array}$$

In the *get* direction, this lens maps a source number to its absolute value; in the reverse direction, it fails if the view b is negative, otherwise uses the sign of the previous source a to determine the sign of the updated source.

The following *logging lens* takes a pure lens l and, whenever the source value a changes, records the previous a value.

$$\begin{array}{l} \mathit{logLens} :: \mathit{Eq}\ \alpha \Rightarrow \alpha \leadsto \beta \to [\alpha \leadsto \beta]_{\mathit{Writer}\ \alpha} \\ \mathit{logLens}\ l = \mathit{MLens}\ (l.\mathit{get})\ (\lambda a\ b \to \\ \quad \mathbf{let}\ a' = l.\mathit{put}\ a\ b\ \mathbf{in\ do}\ \{\mathbf{if}\ a \not= a'\ \mathbf{then}\ \mathit{tell}\ a\ \mathbf{else}\ \mathit{return}\ (); \mathit{return}\ a'\} \end{array}$$

We presented a number of more involved examples of effectful bx in [1]; all of them can be reformulated as monadic lenses.

Next, we consider composition of monadic lenses.

$$\begin{array}{l} (;) :: \mathit{Monad}\ \mu \Rightarrow [a \leadsto b]_\mu \to [b \leadsto c]_\mu \to [a \leadsto c]_\mu \\ l_1 ; l_2 = \mathit{MLens}\ (l_2.\mathit{mget} \cdot l_1.\mathit{mget})\ \mathit{mput}\ \mathbf{where} \\ \quad \mathit{mput}\ a\ c = \mathbf{do}\ \{b \leftarrow l_2.\mathit{mput}\ (l_1.\mathit{mget}\ a)\ c; l_1.\mathit{mput}\ a\ b\} \end{array}$$

Note that we consider only the simple case in which the lenses share a common monad μ. Composing lenses with effects in different monads would require determining how to compose the monads, which is nontrivial [11, 10].

Theorem 2.5. If $l_1 :: [A \leadsto B]_M$ and $l_2 :: [B \leadsto C]_M$ are well-behaved, then so is $l_1 ; l_2$.
◇

Initialization In practical use, it is usually also necessary to equip lenses with an *initialization* mechanism. Indeed, as already mentioned, Pacheco et al.'s monadic put-lenses make the α argument optional (using *Maybe*), to allow for initialization when only a β is available; we chose to exclude this from our version of monadic lenses above.

We propose the following alternative:

$$\mathbf{data}\ [\alpha \leadsto \beta]_\mu = \mathit{MLens}\ \{\mathit{mget} :: \alpha \to \beta, \mathit{mput} :: \alpha \to \beta \to \mu\ \alpha, \mathit{mcreate} :: \beta \to \mu\ \alpha\}$$

and we consider such initializable monadic lenses to be well-behaved when they satisfy the following additional law:

$$(\mathsf{MCreateGet}) \quad \textbf{do}\ \{a \leftarrow \mathit{mcreate}\ b; k\ a\ (\mathit{mget}\ a)\} = \textbf{do}\ \{a \leftarrow \mathit{mcreate}\ b; k\ a\ b\}$$

As with $(\mathsf{MPutGet})$, this property follows from the special case $k = \lambda x\ y \rightarrow \mathit{return}\ (x,y)$, and we will use this fact freely.

This approach, in our view, helps keep the (GetPut) and (PutGet) laws simple and clear, and avoids the need to wrap *mput*'s first argument in *Just* whenever it is called. Moreover, the *mcreate* operation of the composition of two lenses is easy to define: it is just the Kleisli composition $\lambda c \rightarrow \textbf{do}\ \{b \leftarrow l_2.\mathit{mcreate}\ c; l_1.\mathit{mcreate}\ b\}$ of the *mcreate* operations of the two lenses being composed (in reverse order). The proof of $(\mathsf{MCreateGet})$ for $l_1\,; l_2$ is similar to that for $(\mathsf{MPutGet})$.

When the distinction is important, we use the term *full* for well-behaved lenses equipped with a *create* operation. It is easy to show that the source and view types of a full lens must either both be empty or both non-empty, and that the *get* operation of a full lens is surjective.

3 Symmetric monadic lenses and spans

Hofmann *et al.* [8] proposed *symmetric lenses* that use a *complement* to store (at least) the information that is not present in both views.

$$\textbf{data}\ \alpha \overset{\gamma}{\longleftrightarrow} \beta = \mathit{SLens}\ \{\mathit{put}_R :: (\alpha,\gamma) \rightarrow (\beta,\gamma), \mathit{put}_L :: (\beta,\gamma) \rightarrow (\alpha,\gamma), \mathit{missing} :: \gamma\}$$

Informally, put_R turns an α into a β, modifying a complement γ as it goes, and symmetrically for put_L; and *missing* is an initial complement, to get the ball rolling. Well-behavedness for symmetric lenses amounts to the following equational laws:

$$\begin{array}{ll}
(\mathsf{PutRL}) & \textbf{let}\ (b,c') = \mathit{sl}.\mathit{put}_R\ (a,c)\ \textbf{in}\ \mathit{sl}.\mathit{put}_L\ (b,c') \\
 & \quad = \textbf{let}\ (b,c') = \mathit{sl}.\mathit{put}_R\ (a,c)\ \textbf{in}\ (a,c') \\
(\mathsf{PutLR}) & \textbf{let}\ (a,c') = \mathit{sl}.\mathit{put}_L\ (b,c)\ \textbf{in}\ \mathit{sl}.\mathit{put}_R\ (a,c') \\
 & \quad = \textbf{let}\ (a,c') = \mathit{sl}.\mathit{put}_L\ (b,c)\ \textbf{in}\ (b,c')
\end{array}$$

Furthermore, the composition of two symmetric lenses preserves well-behavedness, and can be defined as follows:

$$\begin{array}{l}
(;) :: (\alpha \overset{\sigma_1}{\longleftrightarrow} \beta) \rightarrow (\beta \overset{\sigma_2}{\longleftrightarrow} \gamma) \rightarrow (\alpha \overset{(\sigma_1,\sigma_2)}{\longleftrightarrow} \gamma) \\
l_1\,; l_2 = \mathit{SLens}\ \mathit{put}_R\ \mathit{put}_L\ (l_1.\mathit{missing}, l_2.\mathit{missing})\ \textbf{where} \\
\quad \mathit{put}_R\ (a,(s_1,s_2)) = \textbf{let}\ (b,s_1') = \mathit{put}_R\ (a,s_1) \\
\quad \qquad\qquad\qquad\qquad\quad\ \ (c,s_2') = \mathit{put}_R\ (b,s_2) \\
\quad \qquad\qquad\qquad\qquad \textbf{in}\ (c,(s_1',s_2')) \\
\quad \mathit{put}_L\ (c,(s_1,s_2)) = \textbf{let}\ (b,s_2') = \mathit{put}_L\ (c,s_2) \\
\quad \qquad\qquad\qquad\qquad\quad\ \ (a,s_1') = \mathit{put}_L\ (b,s_1) \\
\quad \qquad\qquad\qquad\qquad \textbf{in}\ (a,(s_1',s_2'))
\end{array}$$

We can define an *identity* symmetric lens as follows:

$$idSLens :: X \xleftrightarrow{()} X$$
$$idSLens = SLens\ id\ id\ ()$$

It is natural to wonder if symmetric lens composition satisfies identity and associativity laws making symmetric lenses into a category. This is complicated by the fact that the complement types of the composition $idSLens;sl$ and of sl differ, so it is not even type-correct to ask whether $idSLens;sl$ and sl are equal. To make it possible to relate the behavior of symmetric lenses with different complement types, Hofmann *et al.* defined equivalence of symmetric lenses as follows:

Definition 3.1. 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 3.2 (Symmetric lens equivalence). Two symmetric lenses $sl_1 :: SLens\ C_1\ X\ Y$ and $sl_2 :: SLens\ C_2\ X\ Y$ are considered *equivalent* ($sl_1 \equiv_{\mathrm{sl}} sl_2$) 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. $sl_1.put_L \sim_r sl_2.put_L$. ♢

Hofmann *et al.* show that $\equiv_{\mathrm{sl}}$ is an equivalence relation; moreover it is sufficiently strong to validate identity, associativity and congruence laws:

Theorem 3.3 ([8]). If $sl_1 :: X \xleftrightarrow{C_1} Y$ and $sl_2 :: Y \xleftrightarrow{C_2} Z$ are well-behaved, then so is $sl_1 ; sl_2$. In addition, composition satisfies the laws:

(Identity)	$sl ; idSLens \equiv_{\mathrm{sl}} sl \equiv_{\mathrm{sl}} idSLens ; sl$
(Assoc)	$sl_1 ; (sl_2 ; sl_3) \equiv_{\mathrm{sl}} (sl_1 ; sl_2) ; sl_3$
(Cong)	$sl_1 \equiv_{\mathrm{sl}} sl_1' \wedge sl_2 \equiv_{\mathrm{sl}} sl_2' \Longrightarrow sl_1 ; sl_2 \equiv_{\mathrm{sl}} sl_1' ; sl_2'$ ♢

3.1 Naive monadic symmetric 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 M:

Definition 3.4. A *monadic symmetric lens* from A to B with complement type C and effects M consists of two functions converting A to B and vice versa, each also operating on C and possibly having effects in M, and a complement value *missing* used for initialisation:

$$\begin{aligned}\mathbf{data}\ [\alpha \xleftrightarrow{\gamma} \beta]_\mu = SMLens\ \{ & mput_R :: (\alpha, \gamma) \to \mu\ (\beta, \gamma), \\ & mput_L :: (\beta, \gamma) \to \mu\ (\alpha, \gamma), \\ & missing :: \gamma \}\end{aligned}$$

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

$$\begin{aligned}(\mathsf{PutRLM})\quad & \mathbf{do}\ \{ (b, c') \leftarrow sl.mput_R\ (a, c); sl.mput_L\ (b, c') \} \\ & \quad = \mathbf{do}\ \{ (b, c') \leftarrow sl.mput_R\ (a, c); return\ (a, c') \} \\ (\mathsf{PutLRM})\quad & \mathbf{do}\ \{ (a, c') \leftarrow sl.mput_L\ (b, c); sl.mput_R\ (a, c') \} \\ & \quad = \mathbf{do}\ \{ (a, c') \leftarrow sl.mput_L\ (b, c); return\ (b, c') \}\end{aligned}$$
♢

The above monadic generalisation of symmetric lenses appears natural, but it turns out to have some idiosyncrasies, similar to those of the naive version of monadic lenses we considered in Section 2.1.

Composition and well-behavedness Consider the following candidate definition of composition for monadic symmetric lenses:

$$
\begin{array}{l}
(;) :: \mathit{Monad}\ \mu \Rightarrow [\alpha \overset{\sigma_1}{\longleftrightarrow} \beta]_\mu \to [\beta \overset{\sigma_2}{\longleftrightarrow} \gamma]_\mu \to [\alpha \overset{(\sigma_1,\sigma_2)}{\longleftrightarrow} \gamma]_\mu \\
sl_1\,;sl_2 = \mathit{SMLens}\ put_R\ put_L\ missing\ \mathbf{where} \\
\quad put_R\ (a,(s_1,s_2)) = \mathbf{do}\ \{\,(b,s'_1) \leftarrow sl_1.mput_R\ (a,s_1); \\
\qquad\qquad\qquad\qquad\quad (c,s'_2) \leftarrow sl_2.mput_R\ (b,s_2); \\
\qquad\qquad\qquad\qquad\quad return\ (c,(s'_1,s'_2))\,\} \\
\quad put_L\ (c,(s_1,s_2)) = \mathbf{do}\ \{\,(b,s'_2) \leftarrow sl_2.mput_L\ (c,s_2); \\
\qquad\qquad\qquad\qquad\quad (a,s'_1) \leftarrow sl_1.mput_L\ (b,s_1); \\
\qquad\qquad\qquad\qquad\quad return\ (a,(s'_1,s'_2))\,\} \\
\quad missing \qquad\quad = (sl_1.missing, sl_2.missing)
\end{array}
$$

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 3.5. Consider the following construction:

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

The lens *setBool True* has no effect on the complement or values, but sets the state to *True*. Both *setBool True* and *setBool False* are well-behaved, but their composition (in either direction) is not: (PutRLM) fails for *setBool True*;*setBool False* because *setBool True* and *setBool False* share a single *Bool* state value. ♢

Proposition 3.6. *setBool b* is well-behaved for $b \in \{True, False\}$, but *setBool True* ; *setBool False* is not well-behaved. ♢

Composition does preserve well-behavedness for monads satisfying a suitable *commutativity* property, but this rules out many interesting monads, such as *State* and *IO*. Hence, we consider a third formulation.

3.2 Entangled state monads

The types of the $mput_R$ and $mput_L$ operations of symmetric lenses can be seen (modulo mild reordering) as stateful operations in the *state monad* $State\ \gamma\ \alpha = \gamma \to (\alpha, \gamma)$, where the state $\gamma = C$. Inspired by this observation (which we later realized was also anticipated by Hofmann et al.), we considered generalising these operations and their laws to an arbitrary monad in a sequence of papers [3, 1, 2]. In our initial workshop paper, we proposed the following definition:

$$
\begin{array}{l}
\mathbf{data}\ [\alpha \between \beta]_\mu = SetBX\ \{get_L :: \mu\ \alpha, set_L :: \alpha \to \mu\ (), \\
\qquad\qquad\qquad\qquad\quad get_R :: \mu\ \beta, set_R :: \beta \to \mu\ ()\}
\end{array}
$$

subject to a subset of the *State* monad laws [13], such as:

$$
\begin{array}{lll}
(\mathsf{Get_L Set_L}) & \mathbf{do}\,\{a \leftarrow get_L; set_L\ a\} = return\ () \\
(\mathsf{Set_L Get_L}) & \mathbf{do}\,\{set_L\ a; get_L\} \quad = \mathbf{do}\,\{set_L\ a; return\ a\}
\end{array}
$$

This presentation makes clear that bidirectionality can be viewed as a state effect in which two "views" of some common state are *entangled*. That is, rather than storing a pair of views, each independently variable, they are entangled, in the sense that a change to either may also change the other. Accordingly, the entangled state monad operations do *not* satisfy all of the usual laws of state: for example, the set_L and set_R operations do not commute.

However, one difficulty with the entangled state monad formalism is that, as discussed in Section 2.1, effectful *get* operations cause problems for composition. It turned out to be nontrivial to define a satisfactory notion of composition, even for the well-behaved special case where $\mu = StateT\ \sigma\ \nu$ for some ν (here *StateT* is the *state monad transformer*, i.e. $StateT\ \sigma\ \nu\ \alpha = \sigma \lambda to\ \nu\ (\alpha, \sigma)$). We formulated the definition of *monadic lenses* given earlier in this paper in the process of exploring this design space.

3.3 Spans of monadic lenses

Hofmann et al. [8] showed that a symmetric lens is equivalent to a *span* of two ordinary lenses, and later work by Johnson and Rosebrugh [9] investigated such *spans of lenses* in greater depth. Accordingly, we propose the following definition:

Definition 3.7. A span of monadic lenses ("*M*-lens span") is a pair of *M*-lenses having the same source:

$$\mathbf{type}\ [\alpha \leftsquigarrow \sigma \rightsquigarrow \beta]_\mu = Span\ \{left :: [\sigma \rightsquigarrow \alpha]_\mu, right :: [\sigma \rightsquigarrow \beta]_\mu\}$$

We say that an *M*-lens span is well-behaved if both of its components are. ◇

We first note that we can extend either leg of a span with a monadic lens (preserving well-behavedness if the arguments are well-behaved):

$$
\begin{array}{l}
(\lhd) \quad :: Monad\ \mu \Rightarrow [\alpha_1 \rightsquigarrow \alpha_2]_\mu \rightarrow [\alpha_1 \leftsquigarrow \sigma \rightsquigarrow \beta]_\mu \rightarrow [\alpha_2 \leftsquigarrow \sigma \rightsquigarrow \beta]_\mu \\
ml \lhd sp = Span\ (sp.left\,;ml)\ (sp.right) \\
(\rhd) \quad :: Monad\ \mu \Rightarrow [\alpha \leftsquigarrow \sigma \rightsquigarrow \beta_1]_\mu \rightarrow [\beta_1 \rightsquigarrow \beta_2]_\mu \rightarrow [\alpha \leftsquigarrow \sigma \rightsquigarrow \beta_2]_\mu \\
sp \rhd ml = Span\ sp.left\ (sp.right\,;ml)
\end{array}
$$

To define composition, the basic idea is as follows. Given two spans $[A \leftsquigarrow S_1 \rightsquigarrow B]_M$ and $[B \leftsquigarrow S_2 \rightsquigarrow C]_M$ with a common type B "in the middle", we want to form a single span from A to C. The obvious thing to try is to form a pullback of the two monadic lenses from S_1 and S_2 to the common type B, obtaining a span from some common state type S to the state types S_1 and S_2, and composing with the outer legs. [**JRC:** *TODO: Diagram*] However, the category of monadic lenses doesn't have pullbacks (as Johnson and Rosebrugh note, this is already the case for ordinary lenses). Instead, we construct the appropriate span as follows.

$$
\begin{aligned}
&(\bowtie) :: \mathit{Monad}\ \mu \Rightarrow [\sigma_1 \rightsquigarrow \beta]_\mu \to [\sigma_2 \rightsquigarrow \beta]_\mu \to [\sigma_1 \leftsquigarrow (\sigma_1 \bowtie \sigma_2) \rightsquigarrow \sigma_2]_\mu \\
&l_1 \bowtie l_2 = \mathit{Span}\ (\mathit{MLens}\ \mathit{fst}\ \mathit{put}_L\ \mathit{create}_L)\ (\mathit{MLens}\ \mathit{snd}\ \mathit{put}_R\ \mathit{create}_R)\ \textbf{where} \\
&\quad \mathit{put}_L\ (\_, s_2)\ s_1' = \textbf{do}\ \{ s_2' \leftarrow l_2.\mathit{mput}\ s_2\ (l_1.\mathit{mget}\ s_1'); \mathit{return}\ (s_1', s_2') \} \\
&\quad \mathit{create}_L\ s_1 \qquad = \textbf{do}\ \{ s_2' \leftarrow l_2.\mathit{mcreate}\ (l_1.\mathit{mget}\ s_1); \mathit{return}\ (s_1, s_2') \} \\
&\quad \mathit{put}_R\ (s_1, \_)\ s_2' = \textbf{do}\ \{ s_1' \leftarrow l_1.\mathit{mput}\ s_1\ (l_2.\mathit{mget}\ s_2'); \mathit{return}\ (s_1', s_2') \} \\
&\quad \mathit{create}_R\ s_1 \qquad = \textbf{do}\ \{ s_1' \leftarrow l_1.\mathit{mcreate}\ (l_2.\mathit{mget}\ s_2); \mathit{return}\ (s_1', s_2) \}
\end{aligned}
$$

where we write $S_1 \bowtie S_2$ for the type of *consistent* state pairs $\{ (s_1, s_2) \in S_1 \times S_2 \mid l_1.\mathit{mget}\ (s_1) = l_2.\mathit{mget}\ (s_2) \}$. In the absence of dependent types, we represent this type as (S_1, S_2) in Haskell, and we need to check that the *mput* and *mcreate* operations respect the consistency invariant.

Lemma 3.8. If $ml_1 :: [\sigma_1 \rightsquigarrow \beta]_\mu$ and $ml_2 :: [\sigma_2 \rightsquigarrow \beta]_\mu$ are well-behaved then so is $ml_1 \bowtie ml_2 :: [\sigma_1 \leftsquigarrow (\sigma_1 \bowtie \sigma_2) \rightsquigarrow \sigma_2]_\mu$. ♢

Note that (MPutGet) and (MCreateGet) hold by construction and do not need the corresponding properties for l_1 and l_2, but these properties are needed to show that consistency is established by *mcreate* and preserved by *mput*.

We can now define composition as follows:

$$
\begin{aligned}
&(;) :: \mathit{Monad}\ \mu \Rightarrow [\alpha \leftsquigarrow \sigma_1 \rightsquigarrow \beta]_\mu \to [\beta \leftsquigarrow \sigma_2 \rightsquigarrow \gamma]_\mu \to [\alpha \leftsquigarrow \sigma_1 \bowtie \sigma_2 \rightsquigarrow \gamma]_\mu \\
&\mathit{sp}_1 ; \mathit{sp}_2 = \mathit{sp}_1.\mathit{left} \lhd (\mathit{sp}_1.\mathit{right} \bowtie \mathit{sp}_2.\mathit{left}) \rhd \mathit{sp}_2.\mathit{right}
\end{aligned}
$$

The well-behavedness of the composition of two well-behaved spans is immediate because $\lhd$ and $\rhd$ preserve well-behavedness of their arguments:

Theorem 3.9. If $\mathit{sp}_1 :: [\alpha \leftsquigarrow \sigma_1 \rightsquigarrow \beta]_\mu$ and $\mathit{sp}_2 :: [\beta \leftsquigarrow \sigma_2 \rightsquigarrow \gamma]_\mu$ are well-behaved spans of monadic lenses, then their composition $\mathit{sp}_1 ; \mathit{sp}_2$ is well-behaved. ♢

Given a span of monadic lenses $\mathit{sp} :: [A \leftsquigarrow S \rightsquigarrow B]_M$, we can construct a symmetric lens $\mathit{sl} :: [A \overset{\mathit{Maybe}\ S}{\longleftrightarrow} B]_M$ as follows:

$$
\begin{aligned}
&\mathit{span2smlens}\ (\mathit{left}, \mathit{right}) = \mathit{SMLens}\ \mathit{mput}_R\ \mathit{mput}_L\ \mathit{Nothing}\ \textbf{where} \\
&\quad \mathit{mput}_R\ (a, \mathit{Just}\ s) \quad\ = \textbf{do}\ \{ s' \leftarrow \mathit{left}.\mathit{mput}\ s\ a; \mathit{return}\ (\mathit{right}.\mathit{mget}\ s', \mathit{Just}\ s') \} \\
&\quad \mathit{mput}_R\ (a, \mathit{Nothing}) = \textbf{do}\ \{ s' \leftarrow \mathit{left}.\mathit{mcreate}\ a; \mathit{return}\ (\mathit{right}.\mathit{mget}\ s', \mathit{Just}\ s') \} \\
&\quad \mathit{mput}_L \qquad\qquad\quad = \ldots \quad \text{-- dual}
\end{aligned}
$$

Essentially, these operations use the span's *mput* and *mget* operations to update one side and obtain the new view value for the other side, and use *create* operations to build the initial S state if the complement is *Nothing*.

Well-behavedness is preserved by the conversion from monadic lens spans to *SMLens*, for arbitrary monads M:

Theorem 3.10. If $\mathit{sp} :: [A \leftsquigarrow S \rightsquigarrow B]_M$ is well-behaved, then $\mathit{span2smlens}\ \mathit{sp}$ is also well-behaved. ♢

Given $\mathit{sl} :: [A \overset{C}{\longleftrightarrow} B]_M$, 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 $\mathit{sp} :: [A \leftsquigarrow S \rightsquigarrow B]_M$ by

$$
\begin{array}{l}
\mathit{smlens2span}\ \mathit{sl} = \mathit{Span}\ (\mathit{MLens}\ \mathit{get}_L\ \mathit{put}_L\ \mathit{create}_L)\ (\mathit{MLens}\ \mathit{get}_R\ \mathit{put}_R\ \mathit{create}_R)\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{get}_L\ (a,b,c) & = a \\
\mathit{put}_L\ (a,b,c)\ a' & = \textbf{do}\ \{ (b',c') \leftarrow \mathit{sl.mput}_R\ (a',c); \mathit{return}\ (a',b',c') \} \\
\mathit{create}_L\ a & = \textbf{do}\ \{ (b,c) \leftarrow \mathit{sl.mput}_R\ (a,\mathit{sl.missing}); \mathit{return}\ (a,b,c) \} \\
\mathit{get}_R\ (a,b,c) & = b \\
\mathit{put}_R\ (a,b,c)\ b' & = \textbf{do}\ \{ (a',c') \leftarrow \mathit{sl.mput}_L\ (b',c); \mathit{return}\ (a',b',c') \} \\
\mathit{create}_R\ b & = \textbf{do}\ \{ (a,c) \leftarrow \mathit{sl.mput}_L\ (b,\mathit{sl.missing}); \mathit{return}\ (a,b,c) \}
\end{array}
\end{array}
$$

However, *smlens2span* may not preserve well-behavedness even for commutative monads such as *Maybe*, as the following counterexample illustrates:

Example 3.11. Consider the following monadic symmetric lens construction:

$$
\begin{array}{l}
\mathit{fail} :: [() \overset{()}{\longleftrightarrow} ()]_{\mathit{Maybe}} \\
\mathit{fail} = \mathit{SMLens}\ \mathit{Nothing}\ \mathit{Nothing}\ ()
\end{array}
$$

This is well-behaved but *smlens2span fail* is not; each leg of the induced span is of the following form:

$$
\begin{array}{l}
\mathit{failMLens} :: \mathit{MLens}\ \mathit{Maybe}\ ()\ () \\
\mathit{failMLens} = \mathit{MLens}\ \mathit{id}\ (\lambda ()\ () \rightarrow \mathit{Nothing})\ (\lambda () \rightarrow \mathit{Nothing})
\end{array}
$$

which cannot satisfy (MGetPut). ♢

For pure symmetric lenses, *smlens2span* does preserve well-behavedness.

Theorem 3.12. If $sl :: \mathit{SMLens}\ \mathit{Id}\ C\ A\ B$ is well-behaved, then *smlens2span sl* is also well-behaved, with state space S consisting of the consistent triples of *sl*. ♢

To summarize: spans of monadic lenses are closed under composition, and correspond to well-behaved symmetric monadic lenses. However, there are well-behaved symmetric monadic lenses that do not map to well-behaved spans. It seems to be an interesting open problem to give a direct axiomatization of the symmetric monadic lenses that are essentially spans of monadic lenses (and are therefore closed under composition).

4 Equivalence of spans

As mentioned already, Hofmann et al. [8] introduced a bisimulation-like notion of equivalence for pure symmetric lenses, in order to validate laws such as identity, associativity and congruence of composition. Johnson and Rosebrugh [9] introduced a definition of equivalence of spans and compared it with symmetric lens equivalence. We have considered equivalences based on isomorphism [1] and bisimulation [2]. In this section we consider and relate these approaches in the context of spans of M-lenses.

Definition 4.1 (Isomorphism Equivalence). Two M-lens spans $sp_1 :: [A \leftsquigarrow S_1 \rightsquigarrow B]_M$ and $sp_2 :: [A \leftsquigarrow S_2 \rightsquigarrow B]_M$ are isomorphic ($sp \equiv_{\mathrm{i}} sp'$) if there is an isomorphism $h :: S_1 \rightarrow S_2$ on their state spaces such that $h\,; sp_2.\mathit{left} = sp_1.\mathit{left}$ and $h\,; sp_2.\mathit{right} = sp_1.\mathit{right}$. ♢

Note that any isomorphism $h :: S_1 \to S_2$ can be made into a (monadic) lens; we omit an explicit conversion.

We consider a second definition of equivalence, inspired by Johnson and Rosebrugh [9], which we call *span equivalence*:

Definition 4.2 (Span Equivalence). Two M-lens spans $sp_1 :: [A \leftsquigarrow S_1 \rightsquigarrow B]_M$ and $sp_2 :: [A \leftsquigarrow S_2 \rightsquigarrow B]_M$ are related by $\curvearrowright$ if there is a full lens $h :: S_1 \rightsquigarrow S_2$ such that $h \,;sp_2.left = sp_1.left$ and $h;sp_2.right = sp_1.right$. The equivalence relation $\equiv_{\mathsf{s}}$ is the least equivalence relation containing $\curvearrowright$. $\Diamond$

One important consideration (emphasized by Johnson and Rosebrugh) is the need to avoid making all compatible spans equivalent to the "trivial" span $[A \leftsquigarrow \emptyset \rightsquigarrow B]_M$. To avoid this problem, they imposed conditions on h: its *get* function must be surjective and *split*, meaning that there exists a function c such that $h.get \cdot c = id$. We chose instead to require h to be a full lens. This is actually slightly stronger than Johnson and Rosebrugh's definition (at least from a constructive perspective), because h is equipped with a specific choice of $c = create$ satisfying $h.get \cdot c = id$ (the $(\mathsf{CreateGet})$ law)..

We have defined span equivalence as the reflexive, symmetric, transitive closure of $\curvearrowright$. Interestingly, any span equivalence is witnessed by a pure span of lenses between the respective state spaces.

Theorem 4.3. Given $sp_1 :: [A \leftsquigarrow S_1 \rightsquigarrow B]_M$ and $sp_2 :: [A \leftsquigarrow S_2 \rightsquigarrow B]_M$, if $sp_1 \equiv_{\mathsf{s}} sp_2$ then there exists $sp :: S_1 \leftsquigarrow S \rightsquigarrow S_2$ such that $sp.left \,; sp_1.left = sp.right \,; sp_2.left$ and $sp.left \,; sp_1.right = sp.right \,; sp_2.right$. $\Diamond$

Thus, span equivalence is a doubly appropriate name for $\equiv_{\mathsf{s}}$: it is an equivalence of spans witnessed by a (pure) span.

Finally, we consider a third notion of equivalence, inspired by the natural bisimulation equivalence for coalgebraic bx [2]:

Definition 4.4 (Base map). Given M-lenses $l_1 :: [S_1 \rightsquigarrow V]_M$ and $l_2 :: [S_2 \rightsquigarrow V]_M$, we say that $h : S_1 \to S_2$ is a *base map* from l_1 to l_2 if

$$\begin{array}{ll} l_1.mget\ s & = l_2.mget\ (h\ s) \\ \mathbf{do}\ \{s \leftarrow l_1.mput\ s\ v; return\ (h\ s)\} & = l_2.mput\ (h\ s)\ v \\ \mathbf{do}\ \{s \leftarrow l_1.mcreate\ v; return\ (h\ s)\} & = l_2.mcreate\ v \end{array}$$

Similarly, given two M-lens spans $sp_1 :: [A \leftsquigarrow S_1 \rightsquigarrow B]_M$ and $sp_2 :: [A \leftsquigarrow S_2 \rightsquigarrow B]_M$ we say that $h :: S_1 \to S_2$ is a base map from sp_1 to sp_2 if h is a base map from sp_1 to $sp_2.left$ and from $sp_1.right$ to $sp_2.right$. $\Diamond$

Definition 4.5 (Bisimulation equivalence). A *bisimulation* of M-lens spans $sp_1 :: [A \leftsquigarrow S_1 \rightsquigarrow B]_M$ and $sp_2 :: [A \leftsquigarrow S_2 \rightsquigarrow B]_M$ is a M-lens span $sp :: [A \leftsquigarrow R \rightsquigarrow B]_M$ where $R \subseteq S_1 \times S_2$ and *fst* is a base map from sp to sp_1 and *snd* is a base map sp from to sp_2. We write $sp_1 \equiv_{\mathsf{b}} sp_2$ when there is a bisimulation of spans sp_1 and sp_2. $\Diamond$

Proposition 4.6. Each of the relations $\equiv_{\mathsf{i}}$, $\equiv_{\mathsf{s}}$ and $\equiv_{\mathsf{b}}$ are equivalence relations on compatible spans of M-lenses and satisfy $(\mathsf{Identity})$, (Assoc) and (Cong). $\Diamond$

Theorem 4.7. $sp_1 \equiv_{\mathsf{i}} sp_2$ implies $sp_1 \equiv_{\mathsf{s}} sp_2$, but not the converse. $\Diamond$

Proof. The forward direction is obvious; for the reverse direction, consider

$$
\begin{aligned}
&h :: Bool \rightsquigarrow ()\\
&h = Lens\ (\backslash\_ \to ())\ (\lambda a\ () \to a)\ (\lambda () \to True)\\
&sp_1 :: [() \leftsquigarrow () \rightsquigarrow ()]_\mu\\
&sp_1 = Span\ idMLens\ idMLens\\
&sp_2 = (h\,;sp_1.left, h\,;sp_2.right)
\end{aligned}
$$

Clearly $sp_1 \equiv_{\mathrm{s}} sp_2$ by definition and all three structures are well-behaved, but h is not an isomorphism: any $k :: () \rightsquigarrow Bool$ must satisfy $k.get\ () = True$ or $k.get\ () = False$, so $(h\,;k).get = k.get \cdot h.get$ cannot be the identity function. $\square$

Theorem 4.8. Given $sp_1 :: [A \leftsquigarrow S_1 \rightsquigarrow B]_M, sp_2 :: [A \leftsquigarrow S_2 \rightsquigarrow B]_M$, if $sp_1 \equiv_{\mathrm{s}} sp_2$ then $sp_1 \equiv_{\mathrm{b}} sp_2$. $\diamondsuit$

Proof. For the forward direction, it suffices to show that a single $sp_1 \curvearrowright sp_2$ step implies $sp_1 \equiv_{\mathrm{b}} sp_2$, which is straightforward by taking R to be the set of pairs $\{(s_1, s_2) \mid l_1.get\ s_1 = s_2\}$, and construct an appropriate span $sp : A \leftsquigarrow R \rightsquigarrow B$. Since bisimulation equivalence is transitive, it follows that $sp_1 \equiv_{\mathrm{s}} sp_2$ implies $sp_1 \equiv_{\mathrm{b}} sp_2$ as well. $\square$

In the pure case, we can also show a (surprising) converse:

Theorem 4.9. Given $sp_1 :: A \leftsquigarrow S_1 \rightsquigarrow B, sp_2 :: A \leftsquigarrow S_2 \rightsquigarrow B$, if $sp_1 \equiv_{\mathrm{b}} sp_2$ then $sp_1 \equiv_{\mathrm{s}} sp_2$. $\diamondsuit$

Proof. Given R and a span $sp :: A \leftsquigarrow R \rightsquigarrow B$ constituting a bisimulation $sp_1 \equiv_{\mathrm{b}} sp_2$, it suffices to construct a span $sp' = (l, r) :: S_1 \leftsquigarrow R \rightsquigarrow S_2$ satisfying $l\,;sp_1.left = r\,;sp2.left$ and $l\,;sp_1.right = r\,;sp2.right$. $\square$

We leave it as an open question to determine whether $\equiv_{\mathrm{b}}$ is equivalent to $\equiv_{\mathrm{s}}$ for spans of monadic lenses, or whether an analogous result to Theorem 4.9 carries over to symmetric lenses.

5 Conclusions

Lenses are a popular and powerful abstraction for bidirectional transformations. Although they are most often studied in their conventional, pure form, practical applications of lenses typically grapple with side-effects, including exceptions, state, and user interaction. Some recent proposals for extending lenses with monadic effects have been made and our proposal for (asymmetric) monadic lenses improves on them because it is closed under composition for arbtrary monads. Furthermore, we investigated the symmetric case, and showed that *spans* of monadic lenses are closed under composition, while the obvious generalization of pure symmetric lenses to incorporate monadic effects is not closed under composition. Finally, we presented three notions of equivalence for spans of monadic lenses, related them, and proved a new (and perhaps surprising) result.

Although some of these ideas are present in recent papers [8, 9, 3, 1, 2], this paper reflects our desire to clarify these ideas and expose them in their clearest form — a desire that is strongly influenced by Wadler's work on a wide variety of related topics [16, 11, 17], and by our interactions with him as a colleague. We hope that this presentation will not just be an interesting curiosity but will also help inspire further research and appreciation of bidirectional programming with effects.

References

1. Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Notions of bidirectional computation and entangled state monads. In *The 2015 International Conference on Mathematics of Program Construction (MPC 2015)*, number 9129 in LNCS, pages 187–214. Springer-Verlag, 2015.
2. Faris Abou-Saleh, James McKinna, and Jeremy Gibbons. Coalgebraic aspects of bidirectional computation. In *Proceedings of the 4th International Workshop on Bidirectional Transformations (BX 2015)*, volume 1396 of *CEUR-WS*, pages 15–30, 2015.
3. James Cheney, James McKinna, Perdita Stevens, Jeremy Gibbons, and Faris Abou-Saleh. Entangled state monads. In Terwilliger and Hidaka [15].
4. Péter Diviánszky. LGtk API correction. `http://people.inf.elte.hu/divip/LGtk/CorrectedAPI.html`, April 2013.
5. 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*, pages 233–246. ACM, 2005.
6. 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. *TOPLAS*, 29(3):17, 2007.
7. Nate Foster, Kazutaka Matsuda, and Janis Voigtländer. Three complementary approaches to bidirectional programming. In *Spring School on Generic and Indexed Programming*, volume 7470 of *LNCS*, pages 1–46. Springer, 2012.
8. Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Symmetric lenses. In *POPL*, pages 371–384. ACM, 2011.
9. Michael Johnson and Robert Rosebrugh. Spans of lenses. In Terwilliger and Hidaka [15].
10. Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report RR-1004, DCS, Yale, 1993.
11. David J. King and Philip Wadler. Combining monads. In *Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6-8 July 1992*, pages 134–143, 1992.
12. Hugo Pacheco, Zhenjiang Hu, and Sebastian Fischer. Monadic combinators for "putback" style bidirectional programming. In *PEPM*, pages 39–50. ACM, 2014.
13. Gordon D. Plotkin and John Power. Notions of computation determine monads. In *FOSSACS*, volume 2303 of *LNCS*, pages 342–356. Springer, 2002.
14. Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. *SoSyM*, 9(1):7–20, 2010.
15. James Terwilliger and Soichiro Hidaka, editors. *BX Workshop*. `http://ceur-ws.org/Vol-1133/#bx`, 2014.
16. Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.
17. Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 24–52. Springer, 1995.

A Proofs for Section 2

Theorem 2.5. If $l_1 :: [A \rightsquigarrow B]_M$ and $l_2 :: [B \rightsquigarrow C]_M$ are well-behaved, then so is $l_1 \mathbin{;} l_2$. $\diamondsuit$

Proof. Suppose l_1 and l_2 are well-behaved, and let $l = l_1 \mathbin{;} l_2$. We reason as follows for $(\mathsf{MGetPut})$:

$$
\begin{array}{ll}
& \mathbf{do}\,\{\,l.mput\ a\ (l.mget\ a)\,\} \\
= & [\![\ \text{definition}\ ]\!] \\
& \mathbf{do}\,\{\,b \leftarrow l_2.mput\ (l_1.mget\ a)\ (l_2.mget\ (l_1.mget\ a)); l_1.mput\ a\ b\,\} \\
= & [\![\ (\mathsf{MGetPut})\ ]\!] \\
& \mathbf{do}\,\{\,b \leftarrow return\ (l_1.mget\ a); l_1.mput\ a\ b\,\} \\
= & [\![\ \text{monad unit}\ ]\!] \\
& \mathbf{do}\,\{\,l_1.mput\ a\ (l_1.mget\ a)\,\} \\
= & [\![\ (\mathsf{MGetPut})\ ]\!] \\
& return\ a
\end{array}
$$

For $(\mathsf{MPutGet})$, the proof is as follows:

$$
\begin{array}{ll}
& \mathbf{do}\,\{\,a' \leftarrow l.mput\ a\ c; return\ (a', l.mget\ a')\,\} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{do}\,\{\,b \leftarrow l_2.mput\ (l_1.mget\ a)\ c; \\
& \quad a' \leftarrow l_1.mput\ a\ b; \\
& \quad return\ (a', l_2.mget\ (l_1.mget\ a'))\,\} \\
= & [\![\ (\mathsf{MPutGet})\ ]\!] \\
& \mathbf{do}\,\{\,b \leftarrow l_2.mput\ (l_1.mget\ a)\ c; \\
& \quad a' \leftarrow l_1.mput\ a\ b; \\
& \quad return\ (a', l_2.mget\ b)\,\} \\
= & [\![\ (\mathsf{MPutGet})\ ]\!] \\
& \mathbf{do}\,\{\,b \leftarrow l_2.mput\ (l_1.mget\ a)\ c; \\
& \quad a' \leftarrow l_1.mput\ a\ b; \\
& \quad return\ (a', c)\,\} \\
= & [\![\ \text{definition}\ ]\!] \\
& \mathbf{do}\,\{\,a' \leftarrow l.mput\ a\ c; return\ (a', c)\,\}
\end{array}
$$
$\square$

B Proofs for Section 3

Proposition 3.6. *setBool b* is well-behaved for $b \in \{True, False\}$, but *setBool True* ; *setBool False* is not well-behaved. $\diamondsuit$

For the first part:

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

$$
\begin{array}{ll}
& \mathbf{do}\,\{\,(b, c') \leftarrow (setBool\ x).mput_R\ ((), ()); (setBool\ x).mput_L\ (b, c')\,\} \\
= & [\![\ \text{Definition}\ ]\!]
\end{array}
$$

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

For the second part, taking $sl = \mathit{setBool\,True}\,;\mathit{setBool\,False}$, we proceed as follows:

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

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:

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

$$
\begin{array}{ll}
& \quad (c, s_2') \leftarrow (\mathit{setBool\ False}).\mathit{mput}_R\ (b, s_2); \\
& \quad (c', (s_1'', s_2'')) \leftarrow \mathit{return}\ (c, (s_1', s_2')); \\
& \quad \mathit{return}\ (c', (s_1'', s_2''))\,\} \\
= & \quad [\![\ \text{Monad unit}\ ]\!] \\
& \mathbf{do}\,\{\,(b, s_1') \leftarrow (\mathit{setBool\ True}).\mathit{mput}_R\ (a, s_1); \\
& \quad (c, s_2') \leftarrow (\mathit{setBool\ False}).\mathit{mput}_R\ (b, s_2); \\
& \quad \mathit{return}\ (c, (s_1', s_2'))\,\}
\end{array}
$$

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

Lemma 3.8. If $ml_1 :: [\sigma_1 \rightsquigarrow \beta]_\mu$ and $ml_2 :: [\sigma_2 \rightsquigarrow \beta]_\mu$ are well-behaved then so is $ml_1 \bowtie ml_2 :: [\sigma_1 \leftsquigarrow (\sigma_1 \bowtie \sigma_2) \rightsquigarrow \sigma_2]_\mu$. ♢

Proof. It suffices to consider the two lenses $l_1 = \mathit{MLens\,fst\,put_L\,create_L}$ and $l_2 = \mathit{MLens\,snd\,put_R\,create_R}$ in isolation. Moreover, the two cases are completely symmetric, so we only show the first.

For (MGetPut), we show:

$$
\begin{array}{ll}
& \mathbf{do}\,\{\,l_1.\mathit{mput}\ (s_1, s_2)\ (l_1.\mathit{mget}\ (s_1, s_2))\,\} \\
= & \quad [\![\ \text{definition}\ ]\!] \\
& \mathbf{do}\,\{\,\mathit{put}_L\ (s_1, s_2)\ (\mathit{fst}\ (s_1, s_2))\,\} \\
= & \quad [\![\ \text{definition of } \mathit{put}_L \text{ and } \mathit{fst}\ ]\!] \\
& \mathbf{do}\,\{\,s_2' \leftarrow \mathit{right}.\mathit{mput}\ s_2\ (\mathit{left}.\mathit{mget}\ s_1)\,\} \\
= & \quad [\![\ (s_1, s_2) \text{ consistent}\ ]\!] \\
& \mathbf{do}\,\{\,s_2' \leftarrow \mathit{right}.\mathit{mput}\ s_2\ (\mathit{right}.\mathit{mget}\ s_2)\,\} \\
= & \quad [\![\ (\text{MGetPut})\ ]\!] \\
& \mathit{return}\ s
\end{array}
$$

The proof for (MPutGet) does not: [**JG:** *does not what?*]

$$
\begin{array}{ll}
& \mathbf{do}\,\{\,(s_1', s_2') \leftarrow l_1.\mathit{mput}\ (s_1, s_2)\ v; \mathit{return}\ ((s_1', s_2'), l_1.\mathit{mget}\ (s_1', s_2'))\,\} \\
= & \quad [\![\ \text{definition}\ ]\!] \\
& \mathbf{do}\,\{\,(s_1', s_2') \leftarrow \mathit{put}_R\ (s_1, s_2)\ v; \mathit{return}\ ((s_1', s_2'), \mathit{fst}\ (s_1', s_2'))\,\} \\
= & \quad [\![\ \text{definition}\ ]\!] \\
& \mathbf{do}\,\{\,s_2'' \leftarrow ml_2.\mathit{mput}\ s_2\ (ml_1.\mathit{mget}\ v); (s_1', s_2') \leftarrow \mathit{return}\ (v, s_2''); \mathit{return}\ ((s_1', s_2'), \mathit{fst}\ (s_1', s_2'))\,\} \\
= & \quad [\![\ \text{definition of } \mathit{fst}\ ]\!] \\
& \mathbf{do}\,\{\,s_2'' \leftarrow ml_2.\mathit{mput}\ s_2\ (ml_1.\mathit{mget}\ v); (s_1', s_2') \leftarrow \mathit{return}\ (v, s_2''); \mathit{return}\ ((s_1', s_2'), s_1') \\
= & \quad [\![\ \text{monad laws}\ ]\!] \\
& \mathbf{do}\,\{\,s_2'' \leftarrow ml_2.\mathit{mput}\ s_2\ (ml_1.\mathit{mget}\ v); (s_1', s_2') \leftarrow \mathit{return}\ (v, s_2''); \mathit{return}\ ((s_1', s_2'), v)\,\} \\
= & \quad [\![\ \text{definition}\ ]\!] \\
& \mathbf{do}\,\{\,(s_1', s_2') \leftarrow \mathit{put}_L\ (s_1, s_2)\ v; \mathit{return}\ ((s_1', s_2'), v)\,\} \\
= & \quad [\![\ \text{definition}\ ]\!] \\
& \mathbf{do}\,\{\,(s_1', s_2') \leftarrow l_1.\mathit{mput}\ (s_1, s_2)\ v; \mathit{return}\ ((s_1', s_2'), v)\,\}
\end{array}
$$

The proof for (MCreateGet) is similar.

Finally, we show that $\mathit{put}_L :: (\sigma_1 \bowtie \sigma_2) \to \sigma_1 \to \mu\ (\sigma_1 \bowtie \sigma_2)$, and in particular, that it maintains the consistency invariant on the state space $\sigma_1 \bowtie \sigma_2$. Assume that $(s_1, s_2) ::$

$\sigma_1 \bowtie \sigma_2$ and $s_1' :: \sigma_1$ are given. Thus, $ml_1.mget\ s_1 = ml_2.mget\ s_2$. We must show that any value returned by put_L also satisfies this consistency criterion. By definition,

$$put_L\ (s_1, s_2)\ s_1' = \mathbf{do}\ \{ s_2' \leftarrow ml_2.mput\ s_2\ (ml_1.mget\ s_1'); return\ (s_1', s_2') \}$$

By $(\mathsf{MPutGet})$, any s_2' resulting from $ml_2.mput\ s_2\ (ml_1.mget\ s_1')$ will satisfy $ml_2.mget\ s_2' = ml_1.mget\ s_1'$. The proof that $create_L :: \sigma_1 \to \mu\ (\sigma_1 \bowtie \sigma_2)$ is similar, but simpler. $\square$

Theorem 3.10. If $sp :: [A \leftsquigarrow S \rightsquigarrow B]_M$ is well-behaved, then *span2smlens sp* is also well-behaved. $\diamondsuit$

Proof. Let $sl = span2smlens\ sp$. We need to show that the laws (PutRLM) and (PutLRM) hold. We show (PutRLM), and (PutLRM) is symmetric.

[**JRC:** *TODO: Need to update this*] We need to show that

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

There are two cases, depending on whether the initial state mc is *Nothing* or *Just c* for some c.

If $mc = Nothing$ then we reason as follows:

$$\begin{array}{l} \quad \mathbf{do}\ \{ (b', mc') \leftarrow sl.mput_R\ (a, Nothing); sl.mput_L\ (b', mc') \} \\ = \quad [\![\text{ Definition }]\!] \\ \quad \mathbf{do}\ \{ s' \leftarrow sp.left.mcreate\ a; (b', mc') \leftarrow (sp.right.mget\ s', Just\ s'); sl.mput_L\ (b', mc') \} \\ = \quad [\![\text{ monad unit }]\!] \\ \quad \mathbf{do}\ \{ s' \leftarrow sp.left.mcreate\ a; sl.mput_L\ (sp.right.mget\ s', Just\ s') \} \\ = \quad [\![\text{ definition }]\!] \\ \quad \mathbf{do}\ \{ s' \leftarrow sp.left.mcreate\ a; s'' \leftarrow sp.right.mput\ s'\ (sp.right.mget\ s'); return\ (sp.left.mget\ s'', Just\ s'') \} \\ = \quad [\![(\mathsf{MGetPut})]\!] \\ \quad \mathbf{do}\ \{ s' \leftarrow sp.left.mcreate\ a; s'' \leftarrow return\ s'; return\ (sp.left.mget\ s'', Just\ s'') \} \\ = \quad [\![\text{ monad unit }]\!] \\ \quad \mathbf{do}\ \{ s' \leftarrow sp.left.mcreate\ a; return\ (sp.left.mget\ s', Just\ s') \} \\ = \quad [\![labelMCreateGet]\!] \\ \quad \mathbf{do}\ \{ s' \leftarrow sp.left.mcreate\ a; return\ (a, Just\ s') \} \\ = \quad [\![\text{ monad unit }]\!] \\ \quad \mathbf{do}\ \{ s' \leftarrow sp.left.mcreate\ a; (b', mc') \leftarrow (sp.right.get\ s', Just\ s'); return\ (a, mc') - \} \\ = \quad [\![\text{ Definition }]\!] \\ \quad \mathbf{do}\ \{ (b', mc') \leftarrow sl.mput_R\ (a, Nothing); return\ (a, mc') \} \end{array}$$

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

$$\begin{array}{l} \quad \mathbf{do}\ \{ (b', mc') \leftarrow sl.mput_R\ (a, Just\ s); sl.mput_L\ (b', mc') \} \\ = \quad [\![\text{ Definition }]\!] \\ \quad \mathbf{do}\ \{ s' \leftarrow sp.left.mput\ s\ a; (b', mc') \leftarrow (sp.right.mget\ s', Just\ s'); sl.mput_L\ (b', mc') \} \\ = \quad [\![\text{ monad unit }]\!] \\ \quad \mathbf{do}\ \{ s' \leftarrow sp.left.mput\ s\ a; sl.mput_L\ (sp.right.mget\ s', Just\ s') \} \end{array}$$

$$
\begin{array}{ll}
= & [\![\text{ definition }]\!] \\
& \mathbf{do}\,\{\, s' \leftarrow sp.left.mput\ s\ a; s'' \leftarrow sp.right.mput\ s'\ (sp.right.mget\ s'); return\ (sp.left.mget\ s'', Just\ s'')\,\} \\
= & [\![\ (\mathsf{MGetPut})\ ]\!] \\
& \mathbf{do}\,\{\, s' \leftarrow sp.left.mput\ s\ a; s'' \leftarrow return\ s'; return\ (sp.left.mget\ s'', Just\ s'')\,\} \\
= & [\![\text{ monad unit }]\!] \\
& \mathbf{do}\,\{\, s' \leftarrow sp.left.mput\ s\ a; return\ (sp.left.mget\ s', Just\ s')\,\} \\
= & [\![\ (\mathsf{MPutGet})\ ]\!] \\
& \mathbf{do}\,\{\, s' \leftarrow sp.left.mput\ s\ a; return\ (a, Just\ s')\,\} \\
= & [\![\text{ monad unit }]\!] \\
& \mathbf{do}\,\{\, s' \leftarrow sp.left.mput\ s\ a; (b', mc') \leftarrow (sp.right.get\ s', Just\ s'); return\ (a, mc')\,\} \\
= & [\![\text{ Definition }]\!] \\
& \mathbf{do}\,\{\, (b', mc') \leftarrow sl.mput_R\ (a, Just\ c); return\ (a, mc')\,\}
\end{array}
$$

□
□

Theorem 3.12. If $sl :: SMLens\ Id\ C\ A\ B$ is well-behaved, then *smlens2span 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 $sp = smlens2span\ sl$ preserve consistency of the state. Assume (a,b,c) is consistent. To show that $sp.left.mput\ (a,b,c)\ a'$ is consistent for any a', we have to show that (a',b',c') is consistent, where a' is arbitrary and $(b',c') = sl.mput_R\ (a',c)$. For one half of consistency, we have:

$$
\begin{array}{ll}
& sl.mput_L\ (b',c') \\
= & [\![\ sl.mput_R\ (a',c) = (b',c'), \text{ and } (\mathsf{PutRLM})\ ]\!] \\
& (a',c')
\end{array}
$$

and then for the other half:

$$
\begin{array}{ll}
& sl.mput_R\ (a',c') \\
= & [\![\text{ above, and } (\mathsf{PutLRM})\ ]\!] \\
& (b',c')
\end{array}
$$

as required. The proof that $sp.right.mput\ (a,b,c)\ b'$ is consistent is dual. [**JRC:** *This is the part that's non-obvious how to generalize for T other than Id!*]

We will now show that *smlens2span sl* is a well-behaved span for any symmetric lens sl. For $(\mathsf{MGetPut})$, we proceed as follows:

$$
\begin{array}{ll}
& sl.left.mput\ (a,b,c)\ (sl.left.mget\ (a,b,c)) \\
= & [\![\text{ Definition }]\!] \\
& \mathbf{do}\,\{\, (b',c') \leftarrow sl.mput_R\ (a,c); return\ (a,b',c')\,\} \\
= & [\![\text{ Consistency of } (a,b,c)\ ]\!] \\
& \mathbf{do}\,\{\, (b',c') \leftarrow return\ (b,c); return\ (a,b',c')\,\} \\
= & [\![\text{ monad unit }]\!] \\
& return\ (a,b,c)
\end{array}
$$

For $(\mathsf{MPutGet})$, we have:

$$
\begin{array}{ll}
& \mathbf{do}\,\{\, s' \leftarrow sl.left.put\ (a,b,c)\ a'; return\ (sl.left.mget\ s', s')\,\} \\
= & [\![\text{ Definition }]\!]
\end{array}
$$

$$
\begin{array}{ll}
& \textbf{do}\,\{\,(b',c') \leftarrow \mathit{sl.mput}_R\ (a',c); s' \leftarrow \mathit{return}\ (a',b',c'); \mathit{return}\ (\mathit{sl.left.mget}\ s',s')\,\} \\
= & [\![\ \text{monad unit}\ ]\!] \\
& \textbf{do}\,\{\,(b',c') \leftarrow \mathit{sl.mput}_R\ (a',c); \mathit{return}\ (\mathit{sl.left.mget}\ (a',b',c'),(a',b',c'))\,\} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \textbf{do}\,\{\,(b',c') \leftarrow \mathit{sl.mput}_R\ (a',c); \mathit{return}\ (a',(a',b',c'))\,\} \\
= & [\![\ \text{monad unit}\ ]\!] \\
& \textbf{do}\,\{\,(b',c') \leftarrow \mathit{sl.mput}_R\ (a',c); s' \leftarrow \mathit{return}\ (a',b',c'); \mathit{return}\ (a',s')\,\} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \textbf{do}\,\{\,s' \leftarrow \mathit{sl.left.put}\ (a,b,c)\ a'; \mathit{return}\ (a',s')\,\}
\end{array}
$$

The proof for $(\mathsf{MCreateGet})$ is similar. For $(\mathsf{MPutGet})$, we have:

$$
\begin{array}{ll}
& \textbf{do}\,\{\,s \leftarrow \mathit{sl.left.create}\ a; \mathit{return}\ (\mathit{sl.left.mget}\ s,s)\,\} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \textbf{do}\,\{\,(b',c') \leftarrow \mathit{sl.mput}_R\ (a',c); s' \leftarrow \mathit{return}\ (a',b',c'); \mathit{return}\ (\mathit{sl.left.mget}\ s',s')\,\} \\
= & [\![\ \text{monad unit}\ ]\!] \\
& \textbf{do}\,\{\,(b',c') \leftarrow \mathit{sl.mput}_R\ (a',c); \mathit{return}\ (\mathit{sl.left.mget}\ (a',b',c'),(a',b',c'))\,\} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \textbf{do}\,\{\,(b',c') \leftarrow \mathit{sl.mput}_R\ (a',c); \mathit{return}\ (a',(a',b',c'))\,\} \\
= & [\![\ \text{monad unit}\ ]\!] \\
& \textbf{do}\,\{\,(b',c') \leftarrow \mathit{sl.mput}_R\ (a',c); s' \leftarrow \mathit{return}\ (a',b',c'); \mathit{return}\ (a',s')\,\} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \textbf{do}\,\{\,s \leftarrow \mathit{sl.left.create}\ a; \mathit{return}\ (a,s)\,\}
\end{array}
$$

□

C Proofs for Section 4

Lemma C.1. Suppose $l_1 :: A \rightsquigarrow B$ and $l_2 :: C \rightsquigarrow B$ are pure lenses. Then $(l_1 \bowtie l_2).\mathit{left}\,; l_1 = (l_1 \bowtie l_2).\mathit{right}\,; l_2$. ♢

Proof. We show that each component of $(l_1 \bowtie l_2).\mathit{left}\,; l_1$ equals the corresponding component of $(l_1 \bowtie l_2).\mathit{right}\,; l_2$.

For *get*:

$$
\begin{array}{ll}
& (l_1 \bowtie l_2).\mathit{left}\,; l_1).\mathit{get}\ (a,c) \\
= & [\![\ \text{Definition}\ ]\!] \\
& l_1.\mathit{get}\ (l_1 \bowtie l_2).\mathit{left.get}\ (a,c)) \\
= & [\![\ \text{Definition}\ ]\!] \\
& l_1.\mathit{get}\ a \\
= & [\![\ \text{Consistency}\ ]\!] \\
& l_2.\mathit{get}\ c \\
= & [\![\ \text{Definition}\ ]\!] \\
& l_2.\mathit{get}\ (l_1 \bowtie l_2).\mathit{right.get}\ (a,c)) \\
= & [\![\ \text{Definition}\ ]\!] \\
& (l_1 \bowtie l_2).\mathit{right}\,; l_2).\mathit{get}\ (a,c)
\end{array}
$$

For *put*:

$$
\begin{array}{ll}
& ((l_1 \bowtie l_2).\mathit{left}\,;l_1).\mathit{put}\;(a,c)\;b \\
= & [\![\ \text{Definition}\ ]\!] \\
& (l_1 \bowtie l_2).\mathit{left}.\mathit{put}\;(a,c)\;(l_1.\mathit{put}\;((l_1 \bowtie l_2).\mathit{get}\;(a,c))\;b) \\
= & [\![\ \text{Definition}\ ]\!] \\
& (l_1 \bowtie l_2).\mathit{left}.\mathit{put}\;(a,c)\;(l_1.\mathit{put}\;a\;b) \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{let}\;a' = l_1.\mathit{put}\;a\;b\;\mathbf{in} \\
& \mathbf{let}\;c' = l_2.\mathit{put}\;c\;(l_1.\mathit{get}\;a')\;\mathbf{in}\;(a',c') \\
= & [\![\ \text{inline}\ \mathbf{let}\ ]\!] \\
& (l_1.\mathit{put}\;a\;b, l_2.\mathit{put}\;c\;(l_1.\mathit{get}\;(l_1.\mathit{put}\;a\;b))) \\
= & [\![\ (\mathsf{PutGet})\ ]\!] \\
& (l_1.\mathit{put}\;a\;b, l_2.\mathit{put}\;c\;b) \\
= & [\![\ \text{reverse above steps}\ ]\!] \\
& ((l_1 \bowtie l_2).\mathit{right}\,;l_2).\mathit{put}\;(a,c)\;b
\end{array}
$$

Finally, for *create*:

$$
\begin{array}{ll}
& ((l_1 \bowtie l_2).\mathit{left}\,;l_1).\mathit{create}\;b \\
= & [\![\ \text{Definition}\ ]\!] \\
& (l_1 \bowtie l_2).\mathit{left}.\mathit{create}\;(l_1.\mathit{create}\;b) \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{let}\;c = l_2.\mathit{create}\;(l_1.\mathit{get}\;(l_1.\mathit{create}\;b))\;\mathbf{in}\;(l_1.\mathit{create}\;b, c) \\
= & [\![\ (\mathsf{CreateGet})\ ]\!] \\
& \mathbf{let}\;c = l_2.\mathit{create}\;b\;\mathbf{in}\;(l_1.\mathit{create}\;b, c) \\
= & [\![\ \text{Inline}\ \mathbf{let}\ ]\!] \\
& (l_1.\mathit{create}\;b, l_2.\mathit{create}\;b) \\
= & [\![\ \text{reverse above steps}\ ]\!] \\
& ((l_1 \bowtie l_2).\mathit{right}\,;l_2).\mathit{create}\;b
\end{array}
$$

□

Theorem 4.3. Given $sp_1 :: [A \leftrightsquigarrow S_1 \rightsquigarrow B]_M$ and $sp_2 :: [A \leftrightsquigarrow S_2 \rightsquigarrow B]_M$, if $sp_1 \equiv_s sp_2$ then there exists a pure span $sp :: S_1 \leftrightsquigarrow S \rightsquigarrow S_2$ such that $sp.\mathit{left}\,; sp_1.\mathit{left} = sp.\mathit{right}\,; sp_2.\mathit{left}$ and $sp.\mathit{left}\,; sp_1.\mathit{right} = sp.\mathit{right}\,; sp_2.\mathit{right}$. ♢

Proof. Let sp_1 and sp_2 be given such that $sp_1 \equiv_s sp_2$. The proof is by induction on the length of the sequence of $\curvearrowright$ steps linking sp_1 to sp_2.

If $sp_1 = sp_2$ then the result si immediate. If $sp_1 \curvearrowright sp_2$ then we can complete a span between S_1 and S_2 using identity lenses. For the inductive case, suppose that the result holds for sequences of up to $n \curvearrowright$ steps, and suppose $sp_1 \equiv_s sp_2$‘ holds bn n $\curvearrowright$ or $\curvearrowright$ steps. There are two cases, depending on the direction of the first step. If $sp_1 \curvearrowleft sp\_3 \equiv_s sp_2$ then by induction we must have a pure span sp between S_3 and S_2 and $sp_1 \curvearrowleft sp\_3$ holds by virtue of a lens $h :: S_3 \to S_1$, so we can simply compose h with $sp.\mathit{left}$ to obtain the required span between S_1 and S_2. Otherwise, if $sp_1 \curvearrowright sp\_3 \equiv_s sp_2$ then by induction we must have a pure span sp between S_3 and S_2 and we must have a lens $h :: S_1 \lambda to\, S_3$, so we use Lemma C.1 to form a span $sp' :: S_1 \leftrightsquigarrow S_1 \bowtie S_3 \rightsquigarrow S_3$ and extend $sp'.\mathit{right}$ with $sp.\mathit{right}$ to form the required span between S_1 and S_3. □

Theorem 4.8. Given $sp_1 :: [A \leftsquigarrow S_1 \rightsquigarrow B]_M, sp_2 :: [A \leftsquigarrow S_2 \rightsquigarrow B]_M$, if $sp_1 \equiv_{\mathsf{s}} sp_2$ then $sp_1 \equiv_{\mathsf{b}} sp_2$. ♢

Proof. We give the details for the case $sp_1 \curvearrowright sp_2$. First, write $(l_1, r_1) = sp_1$ and $(l_2, r_2) = sp_2$, and suppose $l :: S_1 \rightsquigarrow S_2$ is a lens satisfying $l_1 = l\,;l_2$ and $r_1 = l\,;r_2$.

We need to define a bisimulation consisting of a set $R \subseteq S_1 \times S_2$ and a span $sp = (l_0, r_0) :: [A \leftsquigarrow R \rightsquigarrow B]_M$ such that *fst* maps R to S_1 and *snd* maps sp to sp_2. We take $R = \{\,(s_1, s_2) \mid s_2 = l.get\ (s_1)\,\}$ and proceed as follows:

$$
\begin{array}{ll}
l_0 & :: [R \rightsquigarrow A]_M \\
l_0.mget\ (s_1,s_2) & = l_1.mget\ s_1 \\
l_0.mput\ (s_1,s_2)\ a & = \mathbf{do}\ \{\, s_1 \leftarrow l_1.mput\ s_1\ a; return\ (s_1, l \cdot get\ s_1)\,\} \\
l_0.mcreate\ a & = \mathbf{do}\ \{\, s_1 \leftarrow l_1.mcreate\ a; return\ (s_1, l \cdot get\ s_1)\,\} \\
r_0 & :: [R \rightsquigarrow B]_M \\
r_0.mget\ (s_1,s_2) & = r_1.mget\ s_1 \\
r_0.mput\ (s_1,s_2)\ b & = \mathbf{do}\ \{\, s_1 \leftarrow r_1.mput\ s_1\ b; return\ (s_1, l \cdot get\ s_1)\,\} \\
r_0.mcreate\ b & = \mathbf{do}\ \{\, s_1 \leftarrow r_1.mcreate\ a; return\ (s_1, l \cdot get\ s_1)\,\}
\end{array}
$$

We must now show that l_0 and r_0 are well-behaved (full) lenses, and that the projections *fst* and *snd* map $sp = (l_0, r_0)$ to sp_1 and sp_2 respectively.

We first show that l_0 is well-behaved; the reasoning for r_0 is symmetric. For (MGetPut) we have:

$$
\begin{array}{ll}
& l_0.mput\ (s_1,s_2)\ (l_0.mget\ (s_1,s_2)) \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{do}\ \{\, s_1' \leftarrow l_1.mput\ s_1\ (l_1.mget\ s_1); return\ (s_1', l \cdot get\ s_1')\,\} \\
= & [\![\ labelMPutGet\ ]\!] \\
& \mathbf{do}\ \{\, s_1' \leftarrow return\ s_1; return\ (s_1', l \cdot get\ s_1')\,\} \\
= & [\![\ \text{Monad unit}\ ]\!] \\
& return\ (s_1, l \cdot get\ s_1) \\
= & [\![\ s_2 = l \cdot get\ s_1\ ]\!] \\
& return\ (s_1, s_2)
\end{array}
$$

For (MPutGet) we have:

$$
\begin{array}{ll}
& \mathbf{do}\ \{\, (s_1'', s_2'') \leftarrow l_0.mput\ (s_1,s_2)\ a; return\ ((s_1'',s_2''), l_0.mget\ (s_1'',s_2''))\,\} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{do}\ \{\, s_1' \leftarrow l_1.mput\ s_1\ a; (s_1'',s_2'') \leftarrow return\ (s_1', l \cdot get\ s_1'); return\ ((s_1'',s_2''), l_1.mget\ s_1')\,\} \\
= & [\![\ \text{Monad unit}\ ]\!] \\
& \mathbf{do}\ \{\, s_1' \leftarrow l_1.mput\ s_1\ a; return\ ((s_1', l \cdot get\ s_1'), l_1.mget\ s_1')\,\} \\
= & [\![\ (\mathsf{MPutGet})\ ]\!] \\
& \mathbf{do}\ \{\, s_1' \leftarrow l_1.mput\ s_1\ a; return\ ((s_1', l \cdot get\ s_1'), a)\,\} \\
= & [\![\ \text{Monad unit}\ ]\!] \\
& \mathbf{do}\ \{\, s_1' \leftarrow l_1.mput\ s_1\ a; (s_1'',s_2'') \leftarrow return\ (s_1', l \cdot get\ s_1'); return\ ((s_1'',s_2''), a)\,\} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{do}\ \{\, (s_1'',s_2'') \leftarrow l_0.mput\ (s_1,s_2)\ a; return\ ((s_1'',s_2''), a)\,\}
\end{array}
$$

Finally, for (MCreateGet) we have:

$$
\begin{array}{l}
\quad \mathbf{do}\,\{(s_1,s_2) \leftarrow l_0.\mathit{mcreate}\ a; \mathit{return}\ ((s_1,s_2), l_0.\mathit{mget}\ (s_1,s_2))\} \\
= \quad [\![\ \text{Definition}\ ]\!] \\
\quad \mathbf{do}\,\{s_1' \leftarrow l_1.\mathit{mcreate}\ a; (s_1,s_2) \leftarrow \mathit{return}\ (s_1', l \cdot \mathit{get}\ s_1'); \mathit{return}\ ((s_1,s_2), l_1.\mathit{mget}\ s_1)\} \\
= \quad [\![\ \text{Monad unit}\ ]\!] \\
\quad \mathbf{do}\,\{s_1' \leftarrow l_1.\mathit{mcreate}\ a; \mathit{return}\ ((s_1', l \cdot \mathit{get}\ s_1'), l_1.\mathit{mget}\ s_1')\} \\
= \quad [\![\ (\mathsf{MPutGet})\ ]\!] \\
\quad \mathbf{do}\,\{s_1' \leftarrow l_1.\mathit{mcreate}\ a; \mathit{return}\ ((s_1', l \cdot \mathit{get}\ s_1'), a)\} \\
= \quad [\![\ \text{Monad unit}\ ]\!] \\
\quad \mathbf{do}\,\{s_1' \leftarrow l_1.\mathit{mcreate}\ a; (s_1,s_2) \leftarrow \mathit{return}\ (s_1', l \cdot \mathit{get}\ s_1'); \mathit{return}\ ((s_1,s_2), a)\} \\
= \quad [\![\ \text{Definition}\ ]\!] \\
\quad \mathbf{do}\,\{(s_1,s_2) \leftarrow l_0.\mathit{mcreate}\ a; \mathit{return}\ ((s_1,s_2), a)\}
\end{array}
$$

Next, we show that *fst* maps (l_0,R) to (l_1,S_1) and *snd* maps (l_0,R) to (l_2,S_2). Moreover, it is easy to show that *fst* maps (l_0,R) to (l_1,S_1) by unfolding definitions and easy applications of monad laws. To show that *snd* maps (l_0,R) to (l_2,S_2), we need to verify the following three equations that show that *snd* commutes with *mget*, *mput* and *mcreate*:

$$
\begin{array}{ll}
l_0.\mathit{mget}\ (s_1,s_2) & = l_2.\mathit{mget}\ s_2 \\
\mathbf{do}\,\{(s_1',s_2') \leftarrow l_0.\mathit{mput}\ (s_1,s_2)\ a; \mathit{return}\ s_2'\} & = l_2.\mathit{mput}\ s_2\ a \\
\mathbf{do}\,\{(s_1,s_2) \leftarrow l_0.\mathit{mcreate}\ a; \mathit{return}\ s_2\} & = l_2.\mathit{mcreate}\ a
\end{array}
$$

For the *mget* equation:

$$
\begin{array}{l}
\mathit{notice\ that\ since}\ |\ (s_1,s_2) \in R\ |, \mathit{we\ have} \\
\quad l_0.\mathit{mget}\ (s_1,s_2) \\
= \quad [\![\ \text{Definition}\ ]\!] \\
\quad l_2.\mathit{mget}\ s_2 \\
= \quad [\![\ (s_1,s_2) \in R \text{ means } s_2 = l \cdot \mathit{get}\ s_1\ ]\!] \\
\quad l_2.\mathit{mget}\ (l \cdot \mathit{get}\ s_1) \\
= \quad [\![\ \text{Definition of lens composition}\ ]\!] \\
\quad (l\,; l_2).\mathit{mget}\ s_1 \\
= \quad [\![\ l\,; l_2 = l_1\ ]\!] \\
\quad l_1.\mathit{mget}\ s_1
\end{array}
$$

For the *mput* equation:

$$
\begin{array}{l}
\quad \mathbf{do}\,\{(s_1',s_2') \leftarrow l_0.\mathit{mput}\ (s_1,s_2)\ a; \mathit{return}\ s_2'\} \\
= \quad [\![\ \text{Definition}\ ]\!] \\
\quad \mathbf{do}\,\{s_1'' \leftarrow l_1.\mathit{mput}\ s_1\ a; (s_1',s_2') \leftarrow \mathit{return}\ (s_1'', l \cdot \mathit{get}\ s_1''); \mathit{return}\ s_2'\} \\
= \quad [\![\ \text{Monad laws}\ ]\!] \\
\quad \mathbf{do}\,\{s_1'' \leftarrow l_1.\mathit{mput}\ s_1\ a; \mathit{return}\ (l \cdot \mathit{get}\ s_1'')\} \\
= \quad [\![\ l\,; l_2 = l_1\ ]\!] \\
\quad \mathbf{do}\,\{s_1'' \leftarrow (l\,; l_2).\mathit{mput}\ s_1\ a; \mathit{return}\ (l \cdot \mathit{get}\ s_1'')\} \\
= \quad [\![\ \text{Definition}\ ]\!] \\
\quad \mathbf{do}\,\{s_2'' \leftarrow l_2.\mathit{mput}\ (l.\mathit{get}\ s_1)\ a; s_1'' \leftarrow \mathit{return}\ (l.\mathit{put}\ s_2''\ s_1); \mathit{return}\ (l \cdot \mathit{get}\ s_1'')\} \\
= \quad [\![\ \text{Monad laws}\ ]\!]
\end{array}
$$

$$
\begin{array}{ll}
& \mathbf{do}\ \{ s_2'' \leftarrow l_2.\mathit{mput}\ (l.\mathit{get}\ s_1)\ a; \mathit{return}\ (l \cdot \mathit{get}\ (l.\mathit{put}\ s_2''\ s_1)) \} \\
= & [\![\ (\mathsf{PutGet})\ ]\!] \\
& \mathbf{do}\ \{ s_2'' \leftarrow l_2.\mathit{mput}\ (l.\mathit{get}\ s_1)\ a; \mathit{return}\ s_2'' \} \\
= & [\![\ (s_1,s_2) \in R \text{ so } l.\mathit{get}\ s_1 = s_2\ ]\!] \\
& \mathbf{do}\ \{ s_2'' \leftarrow l_2.\mathit{mput}\ s_2\ a; \mathit{return}\ s_2'' \} \\
= & [\![\ \text{Monad laws}\ ]\!] \\
& l_2.\mathit{mput}\ s_2\ v
\end{array}
$$

For the *mcreate* equation:

$$
\begin{array}{ll}
& \mathbf{do}\ \{ (s_1,s_2) \leftarrow l_0.\mathit{mcreate}\ a; \mathit{return}\ s_2' \} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{do}\ \{ s_1' \leftarrow l_1.\mathit{mcreate}\ a; (s_1,s_2) \leftarrow \mathit{return}\ (s_1', l.\mathit{get}\ s_1'); \mathit{return}\ s_2 \} \\
= & [\![\ \text{Monad laws}\ ]\!] \\
& \mathbf{do}\ \{ s_1' \leftarrow l_1.\mathit{mcreate}\ a; \mathit{return}\ (l.\mathit{get}\ s_1') \} \\
= & [\![\ l; l_2 = l_1\ ]\!] \\
& \mathbf{do}\ \{ s_1' \leftarrow (l; l_2).\mathit{mcreate}\ a; \mathit{return}\ (l.\mathit{get}\ s_1') \} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{do}\ \{ s_2' \leftarrow l_2.\mathit{mcreate}\ a; s_1' \leftarrow \mathit{return}\ l.\mathit{create}\ s_2'; \mathit{return}\ (l.\mathit{get}\ s_1') \} \\
= & [\![\ \text{Monad laws}\ ]\!] \\
& \mathbf{do}\ \{ s_2' \leftarrow l_2.\mathit{mcreate}\ a; \mathit{return}\ (l.\mathit{get}\ (l.\mathit{create}\ s_2')) \} \\
= & [\![\ (\mathsf{CreateGet})\ ]\!] \\
& \mathbf{do}\ \{ s_2' \leftarrow l_2.\mathit{mcreate}\ a; \mathit{return}\ s_2' \} \\
= & [\![\ \text{Monad laws}\ ]\!] \\
& l_2.\mathit{mcreate}\ a
\end{array}
$$

Similar reasoning suffices to show that *fst* maps (r_0,R) to (r_1,S_1) and *snd* maps (r_0,R) to (r_2,S_2), so we can conclude that R and (l,r) constitute a bisimulation between sp_1 and sp_2, that is, $sp_1 \equiv_{\mathsf{b}} sp_2$. □

Theorem 4.9. Given $sp_1 :: A \leftsquigarrow S_1 \rightsquigarrow B, sp_2 :: A \leftsquigarrow S_2 \rightsquigarrow B$, if $sp_1 \equiv_{\mathsf{b}} sp_2$ then $sp_1 \equiv_{\mathsf{s}} sp_2$. ◇

Proof. For convenience, we again write $sp_1 = (l_1,r_1)$ and $sp_2 = (l_2,r_2)$. We are given R and a span $sp :: A \leftsquigarrow R \rightsquigarrow B$ constituting a bisimulation $sp_1 \equiv_{\mathsf{b}} sp_2$. For later reference, we list the properties that must hold by virtue of this bisimulation for any $(s_1,s_2) \in R$:

$$
\begin{array}{llll}
l_0.\mathit{get}\ (s_1,s_2) & = l_1.\mathit{get}\ s_1 & l_0.\mathit{get}\ (s_1,s_2) & = l_2.\mathit{get}\ s_2 \\
\mathit{fst}\ (l_0.\mathit{put}\ (s_1,s_2)\ a) & = l_1.\mathit{put}\ s_1\ a & \mathit{snd}\ (l_0.\mathit{put}\ (s_1,s_2)\ a) & = l_2.\mathit{put}\ s_2\ a \\
\mathit{fst}\ (l_0.\mathit{create}\ a) & = l_1.\mathit{create}\ s_1 & \mathit{snd}\ (l_0.\mathit{create}\ a) & = l_2.\mathit{create}\ a \\
r_0.\mathit{get}\ (s_1,s_2) & = r_1.\mathit{get}\ s_1 & r_0.\mathit{get}\ (s_1,s_2) & = r_2.\mathit{get}\ s_2 \\
\mathit{fst}\ (r_0.\mathit{put}\ (s_1,s_2)\ b) & = r_1.\mathit{put}\ s_1\ b & \mathit{snd}\ (r_0.\mathit{put}\ (s_1,s_2)\ b) & = r_2.\mathit{put}\ s_2\ b \\
\mathit{fst}\ (r_0.\mathit{create}\ b) & = r_1.\mathit{create}\ s_1 & \mathit{snd}\ (r_0.\mathit{create}\ b) & = r_2.\mathit{create}\ b
\end{array}
$$

In addition, it follows that:

$$
\begin{array}{l}
l_0.\mathit{put}\ (s_1,s_2)\ a = (l_1.\mathit{put}\ s_1\ a, l_2.\mathit{put}\ s_2\ a) \in R \\
r_0.\mathit{put}\ (s_1,s_2)\ b = (r_1.\mathit{put}\ s_1\ b, r_2.\mathit{put}\ s_2\ b) \in R
\end{array}
$$

$$l_0.create\ a = (l_1.create\ a, l_2.create\ a) \in R$$
$$r_0.create\ b = (r_1.create\ b, r_2.create\ b) \in R$$

which also implies the following 'twist' equations:

$$\begin{array}{l}
r_1.get\ (l_1.put\ s_1\ a) = r_0.get\ (l_1.put\ s_1\ a, l_2.put\ s_2\ a) = r_2.get\ (l_2.put\ s_2\ a) \\
l_1.get\ (r_1.put\ s_1\ b) = l_0.get\ (r_1.put\ s_1\ b, r_2.put\ s_2\ b) = l_2.get\ (r_2.put\ s_2\ b) \\
r_1.get\ (l_1.create\ a) = r_0.get\ (l_1.create\ a, l_2.create\ a) = r_2.get\ (l_2.create\ a) \\
l_1.get\ (r_1.create\ a) = l_0.get\ (r_1.create\ a, r_2.create\ a) = l_2.get\ (r_2.create\ a)
\end{array}$$

It suffices to construct a span $(l,r) :: S_1 \leftsquigarrow R \rightsquigarrow S_2$ satisfying $l\,;l_1 = r\,;l_2$ and $l\,;r_1 = r\,;r_2$. Define l and r as follows:

$$\begin{array}{llll}
l.get & = fst & r.get & = snd \\
l.put\ (s_1,s_2)\ s'_1 & = l_0.put\ (s_1,s_2)\ (l_1.get\ s'_1) & r.put\ (s_1,s_2)\ s'_2 & = l_0.put\ (s_1,s_2)\ (l_2.get\ s'_2) \\
l.create\ s_1 & = l_0.create\ (l_1.get\ s_1) & r.create\ s_2 & = l_0.create\ (l_2.get\ s_2)
\end{array}$$

Notice that by construction $l :: R \rightsquigarrow S_1$ and $r :: R \rightsquigarrow S_2$, that is, since we have used l_0 and r_0 to define l and r, we do not need to do any more work to check that the pairs produced by *create* and *put* remain in R. Notice also that l and r only use the lenses l_1 and l_2, not r_1 and r_2; we will show nevertheless that they satisfy the required properties.

First, to show that $l\,;l_1 = r\,;l_2$, we proceed as follows for each operation. For *get*:

$$\begin{array}{ll}
& (l\,;l_1).get\ (s_1,s_2) \\
= & [\![\text{ definition }]\!] \\
& l_1.get\ (l.get\ (s_1,s_2)) \\
= & [\![\text{ definition of } l.get = fst, fst \text{ commutes with } get\ ]\!] \\
& l_0.get\ (s_1,s_2) \\
= & [\![\text{ reverse reasoning }]\!] \\
& (r\,;l_2).get\ (s_1,s_2)
\end{array}$$

For *put*, we have:

$$\begin{array}{ll}
& (l\,;l_1).put\ (s_1,s_2)\ a \\
= & [\![\text{ Definition }]\!] \\
& l.put\ (s_1,s_2)\ (l_1.put\ s_1\ a) \\
= & [\![\text{ Definition }]\!] \\
& l_0.put\ (s_1,s_2)\ (l_1.get\ (l_1.put\ s_1\ a)) \\
= & [\![\text{ (PutGet) for } l_1\ ]\!] \\
& l_0.put\ (s_1,s_2)\ a \\
= & [\![\text{ (PutGet) for } l_2\ ]\!] \\
& l_0.put\ (s_1,s_2)\ (l_2.get\ (l_2.put\ s_2\ a)) \\
= & [\![\text{ Definition }]\!] \\
& r.put\ (s_1,s_2)\ (l_2.put\ s_2\ a) \\
= & [\![\text{ Definition }]\!] \\
& (r\,;l_2).put\ (s_1,s_2)\ a
\end{array}$$

Finally, for *create* we have:

$$
\begin{aligned}
& (l\,;l_1).create\ a \\
= \quad & [\![\ \text{Definition}\ ]\!] \\
& l.create\ (l_1.create\ a) \\
= \quad & [\![\ \text{Definition}\ ]\!] \\
& l_0.create\ (l_1.get\ (l_1.create\ a)) \\
= \quad & [\![\ (\mathsf{CreateGet})\ \text{for}\ l_1\ ]\!] \\
& l_0.create\ a \\
= \quad & [\![\ (\mathsf{CreateGet})\ \text{for}\ l_2\ ]\!] \\
& l_0.create\ (l_2.get\ (l_2.create\ a)) \\
= \quad & [\![\ \text{Definition}\ ]\!] \\
& r.create\ (l_2.create\ a) \\
= \quad & [\![\ \text{Definition}\ ]\!] \\
& (r\,;l_2).create\ a
\end{aligned}
$$

Next, we show that $l\,;r_1 = r\,;r_2$. For *get*:

$$
\begin{aligned}
& (l\,;r_1).get\ (s_1,s_2) \\
= \quad & [\![\ \text{Definition}\ ]\!] \\
& r_1.get\ (l.get\ (s_1,s_2)) \\
= \quad & [\![\ \text{definition of}\ l.get = fst, fst\ \text{commutes with}\ r_1.get\ ]\!] \\
& r_0.get\ (s_1,s_2) \\
= \quad & [\![\ \text{reverse above reasoning}\ ]\!] \\
& (r\,;r_2).get\ (s_1,s_2)
\end{aligned}
$$

For *put*, we have:

$$
\begin{aligned}
& (l\,;r_1).put\ (s_1,s_2)\ b \\
= \quad & [\![\ \text{Definition}\ ]\!] \\
& l.put\ (s_1,s_2)\ (r_1.put\ s_1\ b) \\
= \quad & [\![\ \text{Definition}\ ]\!] \\
& l_0.put\ (s_1,s_2)\ (l_1.get\ (r_1.put\ s_1\ b)) \\
= \quad & [\![\ \text{Do the twist}\ ]\!] \\
& l_0.put\ (s_1,s_2)\ (l_2.get\ (r_2.put\ s_2\ b)) \\
= \quad & [\![\ \text{Definition}\ ]\!] \\
& r.put\ (s_1,s_2)\ (r_2.put\ s_2\ b) \\
= \quad & [\![\ \text{Definition}\ ]\!] \\
& (r\,;r_2).put\ (s_1,s_2)\ b
\end{aligned}
$$

Finally, for *create* we have:

$$
\begin{aligned}
& (l\,;r_1).create\ b \\
= \quad & [\![\ \text{Definition}\ ]\!] \\
& l.create\ (r_1.create\ b) \\
= \quad & [\![\ \text{Definition}\ ]\!] \\
& l_0.create\ (l_1.get\ (r_1.create\ b)) \\
= \quad & [\![\ \text{Do the twist}\ ]\!] \\
& l_0.create\ (l_2.get\ (r_2.create\ b))
\end{aligned}
$$

$$\begin{aligned}
&= \quad [\![\text{ Definition }]\!] \\
&\quad r.create\ (r_2.create\ b) \\
&= \quad [\![\text{ Definition }]\!] \\
&\quad (r\,;r_2).create\ b
\end{aligned}$$

We must also show that l and r are well-behaved full lenses. To show that l is well-behaved, we proceed as follows. For (GetPut):

$$\begin{aligned}
&\quad l.get\ (l.put\ (s_1,s_2)\ s_1') \\
&= \quad [\![\text{ Definition }]\!] \\
&\quad fst\ (l_0.put\ (s_1,s_2)\ (l_1.get\ s_1')) \\
&= \quad [\![\ fst \text{ commutes with } put \ ]\!] \\
&\quad l_1.put\ s_1\ (l_1.get\ s_1') \\
&= \quad [\![\ (\mathsf{GetPut}) \text{ for } l_1 \ ]\!] \\
&\quad s_1'
\end{aligned}$$

For (PutGet):

$$\begin{aligned}
&\quad l.put\ (s_1,s_2)\ (l.get\ (s_1,s_2)) \\
&= \quad [\![\text{ Definition }]\!] \\
&\quad l_0.put\ (s_1,s_2)\ (l_1.get\ s_1) \\
&= \quad [\![\text{ Eta-expansion for pairs }]\!] \\
&\quad (fst\ (l_0.put\ (s_1,s_2)\ (l_1.get\ s_1)), snd\ (l_0.put\ (s_1,s_2)\ (l_1.get\ s_1))) \\
&= \quad [\![\ fst, snd \text{ commutes with } put \ ]\!] \\
&\quad (l_1.put\ s_1\ (l_1.get\ s_1), l_2.put\ s_2\ (l_1.get\ s_1)) \\
&= \quad [\![\ l_1.get\ s_1 = l_2.get\ s_2 \ ]\!] \\
&\quad (l_1.put\ s_1\ (l_1.get\ s_1), l_2.put\ s_2\ (l_2.get\ s_2)) \\
&= \quad [\![\ (\mathsf{PutGet}) \text{ for } l_1, l_2 \ ]\!] \\
&\quad (s_1,s_2)
\end{aligned}$$

For $(\mathsf{CreateGet})$:

$$\begin{aligned}
&\quad l.create\ (l.get\ (s_1,s_2)) \\
&= \quad [\![\text{ Definition }]\!] \\
&\quad l_0.create\ (l_1.get\ s_1) \\
&= \quad [\![\text{ Eta-expansion for pairs }]\!] \\
&\quad (fst\ (l_0.create\ (l_1.get\ s_1)), snd\ (l_0.create\ (l_1.get\ s_1))) \\
&= \quad [\![\ fst, snd \text{ commutes with } put \ ]\!] \\
&\quad (l_1.create\ (l_1.get\ s_1), l_1.create\ (l_1.get\ s_1)) \\
&= \quad [\![\ l_1.get\ s_1 = l_2.get\ s_2 \ ]\!] \\
&\quad (l_1.create\ (l_1.get\ s_1), l_1.create\ (l12.get\ s_2)) \\
&= \quad [\![\ (\mathsf{CreateGet}) \ ]\!] \\
&\quad (s_1,s_2)
\end{aligned}$$

Finally, notice that l and r are defined symmetrically so essentially the same reasoning shows r is well-behaved.

To conclude, $sp = (l,r)$ constitutes a span of lenses witnessing that $sp_1 \equiv_{\mathrm{s}} sp_2$. $\square$