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, Oxford, UK
{faris.abou-saleh,jeremy.gibbons}@cs.ox.ac.uk
[2] University of Edinburgh, Edinburgh, UK
{james.cheney,james.mcKinna,perdita.stevens}@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 to ease programming such *bidirectional transformations* have emerged, including *lenses* (Foster et al. 2007), bx based on consistency relations (Stevens 2010), *symmetric lenses* (Hofmann et al. 2011), and a number of variants and extensions (e.g. (Pacheco et al. 2014; Johnson and Rosebrugh 2014)). Most of these proposals consist of an interface with pure functions and some equational laws that characterise good behaviour; the interaction of bidirectionality with other effects has received comparatively little attention.

S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 1–31, 2016.
DOI: 10.1007/978-3-319-30936-1_1

Some programmers and researchers have already proposed ways to combine lenses and monadic effects (Diviánszky 2013; Pacheco et al. 2014). Recently, we have proposed symmetric notions of bidirectional computation based on *entangled state monads* (Cheney et al. 2014; Abou-Saleh et al. 2015a) and *coalgebras* (Abou-Saleh et al. 2015b). As a result, there are now several alternative proposals for bidirectional transformations with effects. While this diversity is natural and healthy, reflecting an active research area, the different proposals tend to employ somewhat different terminology, and the relationships among them are not well understood. Small differences in definitions can have disproportionate impact.

In this paper we summarise 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 asymmetric 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. 2011) 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 generalisation. We then briefly discuss our recent work on symmetric forms of bx with monadic effects (Cheney et al. 2014; Abou-Saleh et al. 2015a, b). 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 (2014).

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. (2011) and Johnson and Rosebrugh (2014), 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 Abou-Saleh et al. (2015a)), span equivalence (following Johnson and Rosebrugh (2014)), and bisimulation (following Hofmann et al. (2011) and Abou-Saleh et al. (2015b)). 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 (1995)), and with the **do**-notation (following Wadler's monad comprehensions Wadler (1992)). Although some of these ideas are present or implicit in recent papers (Hofmann et al. 2011; Johnson and Rosebrugh 2014; Cheney et al. 2014; Abou-Saleh et al. 2015a, b), 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 (Wadler 1992; King and Wadler 1992; Wadler 1995), and by our interactions with him as a colleague.

2 Asymmetric Monadic Lenses

Recall that a *lens* (Foster et al. 2007, 2012) is a pair of functions, usually called *get* and *put*:

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

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 $A \leadsto B$ maintains a source of type A, providing a view of type B 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 are often equipped with a *create* function

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

satisfying an additional law:

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

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.

Lenses have been investigated extensively; see for example Foster et al. (2012) for a recent tutorial overview. For the purposes of this paper, we just recall the definition of *composition* of lenses:

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

which preserves well-behavedness.

2.1 A Naive Approach

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

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

Such an approach has been considered and discussed in some recent Haskell libraries and online discussions (Diviánszky 2013). A natural question arises immediately: what laws should a lens $l :: [A \rightsquigarrow_0 B]_M$ satisfy? The following generalisations of the laws appear natural:

$$\begin{array}{lll} (\mathsf{MGetPut}_0) & \mathbf{do}\ \{b \leftarrow \mathit{mget}\ a; \mathit{mput}\ a\ b\} & = \mathit{return}\ a \\ (\mathsf{MPutGet}_0) & \mathbf{do}\ \{a' \leftarrow \mathit{mput}\ a\ b; \mathit{mget}\ a'\} & = \mathbf{do}\ \{a' \leftarrow \mathit{mput}\ a\ b; \mathit{return}\ b\} \end{array}$$

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 generalisation of composition from the pure case for these operations is:

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

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 $(\mathsf{MGetPut}_0)$ cannot hold. This suggests that *mget* must either be pure, or have side-effects that are reversible by *mput*, ruling out behaviours 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. (2014) 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:

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

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:

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

Then the laws for MLens_1 (adapted from Pacheco et al. (2014 Proposition 3, p. 49)) are as follows:

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

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 generalise 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 the monad at hand, essentially, a function of type $m\ a \to a$ for any type 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 $m :: \mathit{IO}\ a$ 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 $m :: \mathit{IO}\ a$ to find out what its eventual return value is. But then the precondition of the ($\mathsf{MPutGet}_1$) law is always true, which 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 lenses for any monad M:

Definition 2.1 (Monadic Lens). A *monadic lens* from source type A to view type B in which the put operation may have effects from monad M (or "M-lens from A to B"), is represented by the type $[A \rightsquigarrow B]_M$, 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 Sect. 2.1). We say that M-lens l is *well-behaved* if it satisfies

$$\begin{array}{ll}
(\mathsf{MGetPut}) & \mathbf{do}\ \{\,l.\mathit{mput}\ a\ (l.\mathit{mget}\ a)\,\} = \mathit{return}\ a \\
(\mathsf{MPutGet}) & \mathbf{do}\ \{\,a' \leftarrow l.\mathit{mput}\ a\ b; k\ a'\ (l.\mathit{mget}\ a')\,\} \\
 & \quad = \mathbf{do}\ \{\,a' \leftarrow l.\mathit{mput}\ a\ b; k\ a'\ b\,\}
\end{array}$$ ◇

Note that in (MPutGet), we use a continuation $k :: \alpha \rightarrow \beta \rightarrow \mu\ \gamma$ to quantify over all possible subsequent computations in which a' and $l.\mathit{mget}\ a'$ might appear. In fact, using the laws of monads and simply-typed lambda calculus we can prove this law from just the special case $k = \lambda a\ b \rightarrow \mathit{return}\ (a, b)$, 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. Monadic lenses where $\mu = \mathit{Id}$ are called *pure*, and we may refer to ordinary lenses as pure lenses also.

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

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

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

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

$$\begin{array}{l}
\mathit{id}_{\mathrm{l}} :: \alpha \leadsto \alpha \\
\mathit{id}_{\mathrm{l}} = \mathit{Lens}\ (\lambda a \rightarrow a)\ (\lambda\_\ a \rightarrow a) \\
\mathit{fst}_{\mathrm{l}} :: (\alpha, \beta) \leadsto \alpha \\
\mathit{fst}_{\mathrm{l}} = \mathit{MLens}\ \mathit{fst}\ (\lambda (s_1, s_2)\ s'_1 \rightarrow (s'_1, s_2))
\end{array}$$

Many more examples of pure lenses are to be found in the literature (Foster et al. 2007, 2012), 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. (2015a). Pure lenses are usually defined using total functions, which means that *get* must be surjective whenever A is nonempty, 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 $\mathit{Maybe}\ b$ or give *put* the return type $\mathit{Maybe}\ a$ 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

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

which is well-behaved because 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:

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

In the *mget* direction, this lens maps a source number to its absolute value; in the reverse direction, it fails if the view b is negative, and 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.

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

We presented a number of more involved examples of effectful symmetric bx in (Abou-Saleh et al. 2015a). They show how monadic lenses can employ user interaction, state, or nondeterminism to restore consistency. Most of these examples are equivalently definable as *spans* of monadic lenses, which we will discuss in the next section.

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

We propose the following alternative:

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

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

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

As with (MPutGet), this property follows from the special case $k = \lambda x\ y \to \mathit{return}\ (x, y)$, 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.

Next, we consider composition of monadic lenses.

$$\begin{aligned}
&(;) :: \mathit{Monad}\ \mu \Rightarrow [\alpha \rightsquigarrow \beta]_{\mu} \to [\beta \rightsquigarrow \gamma]_{\mu} \to [\alpha \rightsquigarrow \gamma]_{\mu}\\
&l_1\ ;\ l_2 = \mathit{MLens}\ (l_2.\mathit{mget} \cdot l_1.\mathit{mget})\ \mathit{mput}\ \mathit{mcreate}\ \mathbf{where}
\end{aligned}$$

$$\begin{array}{l} \mathit{mput}\ a\ c = \mathbf{do}\ \{\, b \leftarrow l_2.\mathit{mput}\ (l_1.\mathit{mget}\ a)\ c; l_1.\mathit{mput}\ a\ b \,\} \\ \mathit{mcreate}\ c = \mathbf{do}\ \{\, b \leftarrow l_2.\mathit{mcreate}; l_1.\mathit{mcreate} \,\} \end{array}$$

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 themselves, which is nontrivial (King and Wadler 1992; Jones and Duponcheel 1993).

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

3 Symmetric Monadic Lenses and Spans

Hofmann et al. (2011) proposed *symmetric lenses* that use a *complement* to store (at least) the information that is not present in both views.

$$\begin{array}{llll} \mathbf{data}\ \alpha \overset{\gamma}{\longleftrightarrow} \beta = \mathit{SLens}\ \{ & \mathit{put}_{\mathrm{R}} & :: (\alpha, \gamma) \to (\beta, \gamma), \\ & \mathit{put}_{\mathrm{L}} & :: (\beta, \gamma) \to (\alpha, \gamma), \\ & \mathit{missing} & :: \gamma \} \end{array}$$

Informally, $\mathit{put}_{\mathrm{R}}$ turns an α into a β, modifying a complement γ as it goes, and symmetrically for $\mathit{put}_{\mathrm{L}}$; and *missing* is an initial complement, to get the ball rolling. Well-behavedness for symmetric lenses amounts to the following equational laws:

$$\begin{array}{ll} (\mathsf{PutRL}) & \mathbf{let}\ (b, c') = \mathit{sl}.\mathit{put}_{\mathrm{R}}\ (a, c)\ \mathbf{in}\ \mathit{sl}.\mathit{put}_{\mathrm{L}}\ (b, c') \\ & \quad = \mathbf{let}\ (b, c') = \mathit{sl}.\mathit{put}_{\mathrm{R}}\ (a, c)\ \mathbf{in}\ (a, c') \\ (\mathsf{PutLR}) & \mathbf{let}\ (a, c') = \mathit{sl}.\mathit{put}_{\mathrm{L}}\ (b, c)\ \mathbf{in}\ \mathit{sl}.\mathit{put}_{\mathrm{R}}\ (a, c') \\ & \quad = \mathbf{let}\ (a, c') = \mathit{sl}.\mathit{put}_{\mathrm{L}}\ (b, c)\ \mathbf{in}\ (b, c') \end{array}$$

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

$$\begin{array}{l} (;) :: (\alpha \overset{\sigma_1}{\longleftrightarrow} \beta) \to (\beta \overset{\sigma_2}{\longleftrightarrow} \gamma) \to (\alpha \overset{(\sigma_1, \sigma_2)}{\longleftrightarrow} \gamma) \\ l_1\ ;\ l_2 = \mathit{SLens}\ \mathit{put}_{\mathrm{R}}\ \mathit{put}_{\mathrm{L}}\ (l_1.\mathit{missing}, l_2.\mathit{missing})\ \mathbf{where} \\ \quad \mathit{put}_{\mathrm{R}}\ (a, (s_1, s_2)) = \mathbf{let}\ (b, s_1') = \mathit{put}_{\mathrm{R}}\ (a, s_1) \\ \qquad\qquad\qquad\qquad\qquad\quad (c, s_2') = \mathit{put}_{\mathrm{R}}\ (b, s_2) \\ \qquad\qquad\qquad\qquad\quad \mathbf{in}\ (c, (s_1', s_2')) \\ \quad \mathit{put}_{\mathrm{L}}\ (c, (s_1, s_2)) = \mathbf{let}\ (b, s_2') = \mathit{put}_{\mathrm{L}}\ (c, s_2) \\ \qquad\qquad\qquad\qquad\qquad\quad (a, s_1') = \mathit{put}_{\mathrm{L}}\ (b, s_1) \\ \qquad\qquad\qquad\qquad\quad \mathbf{in}\ (a, (s_1', s_2')) \end{array}$$

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

$$\begin{array}{l} \mathit{id}_{\mathrm{sl}} :: \alpha \overset{()}{\longleftrightarrow} \alpha \\ \mathit{id}_{\mathrm{sl}} = \mathit{SLens}\ \mathit{id}\ \mathit{id}\ () \end{array}$$

It is natural to wonder whether 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 $id_{\mathrm{sl}}; sl$ and of sl differ, so it is not even type-correct to ask whether $id_{\mathrm{sl}}; sl$ and sl are equal. To make it possible to relate the behaviour 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 $sl_1 :: X \xleftrightarrow{C_1} Y$ and $sl_2 :: X \xleftrightarrow{C_2} Y$ 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. $(sl_1.\mathit{missing}, sl_2.\mathit{missing}) \in R$,
2. $sl_1.\mathit{put}_{\mathrm{R}} \sim_R sl_2.\mathit{put}_{\mathrm{R}}$, and
3. $sl_1.\mathit{put}_{\mathrm{L}} \sim_R sl_2.\mathit{put}_{\mathrm{L}}$. ♢

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 (Hofmann et al. 2011). If $sl_1 :: X \xleftrightarrow{C_1} Y$ and $sl_2 :: Y \xleftrightarrow{C_2} Z$ are well-behaved, then so is $sl_1 \mathbin{;} sl_2$. In addition, composition satisfies the laws:

$$\begin{array}{ll}
\textsf{(Identity)} & sl \mathbin{;} id_{\mathrm{sl}} \equiv_{\mathrm{sl}} sl \equiv_{\mathrm{sl}} id_{\mathrm{sl}} \mathbin{;} sl \\
\textsf{(Assoc)} & sl_1 \mathbin{;} (sl_2 \mathbin{;} sl_3) \equiv_{\mathrm{sl}} (sl_1 \mathbin{;} sl_2) \mathbin{;} sl_3 \\
\textsf{(Cong)} & sl_1 \equiv_{\mathrm{sl}} sl_1' \wedge sl_2 \equiv_{\mathrm{sl}} sl_2' \Longrightarrow sl_1 \mathbin{;} sl_2 \equiv_{\mathrm{sl}} sl_1' \mathbin{;} sl_2'
\end{array}$$

♢

3.1 Naive Monadic Symmetric Lenses

We now consider an obvious monadic generalisation of symmetric lenses, in which the $\mathit{put}_{\mathrm{L}}$ and $\mathit{put}_{\mathrm{R}}$ 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:

$$\begin{array}{l}
\mathbf{data}\ [\alpha \xleftrightarrow{\gamma} \beta]_{\mu} = \mathit{SMLens}\ \{\mathit{mput}_{\mathrm{R}} :: (\alpha, \gamma) \to \mu\ (\beta, \gamma), \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{mput}_{\mathrm{L}} :: (\beta, \gamma) \to \mu\ (\alpha, \gamma), \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{missing} :: \gamma\}
\end{array}$$

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

$$\begin{array}{ll}
\textsf{(PutRLM)} & \mathbf{do}\ \{(b, c') \leftarrow sl.\mathit{mput}_{\mathrm{R}}\ (a, c); sl.\mathit{mput}_{\mathrm{L}}\ (b, c')\} \\
& \quad = \mathbf{do}\ \{(b, c') \leftarrow sl.\mathit{mput}_{\mathrm{R}}\ (a, c); \mathit{return}\ (a, c')\} \\
\textsf{(PutLRM)} & \mathbf{do}\ \{(a, c') \leftarrow sl.\mathit{mput}_{\mathrm{L}}\ (b, c); sl.\mathit{mput}_{\mathrm{R}}\ (a, c')\} \\
& \quad = \mathbf{do}\ \{(a, c') \leftarrow sl.\mathit{mput}_{\mathrm{L}}\ (b, c); \mathit{return}\ (b, c')\}
\end{array}$$

♢

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 Sect. 2.1.

Composition and Well-Behavedness. Consider the following candidate definition of composition for monadic symmetric lenses:

$$\begin{array}{l}
(;) :: \mathit{Monad}\ \mu \Rightarrow [\alpha \overset{\sigma_1}{\longleftrightarrow} \beta]_\mu \to [\beta \overset{\sigma_2}{\longleftrightarrow} \gamma]_\mu \to [\alpha \overset{(\sigma_1,\sigma_2)}{\longleftrightarrow} \gamma]_\mu \\
\mathit{sl}_1\ ;\ \mathit{sl}_2 = \mathit{SMLens}\ \mathit{put}_{\mathrm{R}}\ \mathit{put}_{\mathrm{L}}\ \mathit{missing}\ \mathbf{where} \\
\quad \begin{array}{lcl}
\mathit{put}_{\mathrm{R}}\ (a,(s_1,s_2)) & = & \mathbf{do}\ \{ (b, s_1') \leftarrow \mathit{sl}_1.\mathit{mput}_{\mathrm{R}}\ (a, s_1); \\
 & & \phantom{\mathbf{do}\ \{} (c, s_2') \leftarrow \mathit{sl}_2.\mathit{mput}_{\mathrm{R}}\ (b, s_2); \\
 & & \phantom{\mathbf{do}\ \{} \mathit{return}\ (c, (s_1', s_2'))\} \\
\mathit{put}_{\mathrm{L}}\ (c,(s_1,s_2)) & = & \mathbf{do}\ \{ (b, s_2') \leftarrow \mathit{sl}_2.\mathit{mput}_{\mathrm{L}}\ (c, s_2); \\
 & & \phantom{\mathbf{do}\ \{} (a, s_1') \leftarrow \mathit{sl}_1.\mathit{mput}_{\mathrm{L}}\ (b, s_1); \\
 & & \phantom{\mathbf{do}\ \{} \mathit{return}\ (a, (s_1', s_2'))\} \\
\mathit{missing} & = & (\mathit{sl}_1.\mathit{missing}, \mathit{sl}_2.\mathit{missing})
\end{array}
\end{array}$$

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:

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

The lens *setBool True* has no effect on the complement or values, 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 $b \in \{\mathit{True}, \mathit{False}\}$, but *setBool True* ; *setBool False* is not well-behaved. ♢

Composition does preserve well-behavedness for commutative monads, i.e. those for which

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

but this rules out many interesting monads, such as *State* and *IO*.

3.2 Entangled State Monads

The types of the $\mathit{mput}_{\mathrm{R}}$ and $\mathit{mput}_{\mathrm{L}}$ operations of symmetric lenses can be seen (modulo mild reordering) as stateful operations in the *state monad* $\mathit{State}\ \gamma\ \alpha = \gamma \to (\alpha, \gamma)$, where the state $\gamma = C$. This observation was also anticipated by Hofmann et al. In a sequence of papers, we considered generalising these operations and their laws to an arbitrary monad (Cheney et al. 2014; Abou-Saleh et al. 2015a, b). In our initial workshop paper, we proposed the following definition:

$$\begin{aligned}&\textbf{data}\ [\alpha \nleftrightarrow \beta]_{\mu} = \mathit{SetBX}\ \{\mathit{get}_L :: \mu\ \alpha, \mathit{set}_L :: \alpha \rightarrow \mu\ (),\\ &\qquad\qquad\qquad\qquad\qquad \mathit{get}_R :: \mu\ \beta, \mathit{set}_R :: \beta \rightarrow \mu\ ()\}\end{aligned}$$

subject to a subset of the *State* monad laws (Plotkin and Power 2002), such as:

$$\begin{array}{lll}(\mathsf{Get_L Set_L}) & \textbf{do}\ \{a \leftarrow \mathit{get}_L; \mathit{set}_L\ a\} & = \mathit{return}\ ()\\ (\mathsf{Set_L Get_L}) & \textbf{do}\ \{\mathit{set}_L\ a; \mathit{get}_L\} & = \textbf{do}\ \{\mathit{set}_L\ a; \mathit{return}\ a\}\end{array}$$

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 Sect. 2.1, effectful *mget* 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 $\mu = \mathit{StateT}\ \sigma\ \nu$ for some ν, where *StateT* is the *state monad transformer* (Liang et al. 1995), i.e. $\mathit{StateT}\ \sigma\ \nu\ \alpha = \sigma \rightarrow \nu\ (\alpha, \sigma)$. 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. (2011) showed that a symmetric lens is equivalent to a *span* of two ordinary lenses, and later work by Johnson and Rosebrugh (2014) investigated such *spans of lenses* in greater depth. Accordingly, we propose the following definition:

Definition 3.7 (Monadic Lens Spans). A span of monadic lenses (“M-lens span”) is a pair of M-lenses having the same source:

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

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):

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

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”,

$$\begin{tikzcd}
 & & S_1 \bowtie S_2 \arrow[dl, dotted] \arrow[dr, dotted] & & \\
 & S_1 \arrow[dl] \arrow[dr] & & S_2 \arrow[dl] \arrow[dr] & \\
A & & B & & C
\end{tikzcd}$$

Fig. 1. Composing spans of lenses

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. (See Fig. 1.) 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.

$$\begin{array}{l}
(\bowtie) :: \mathit{Monad}\ \mu \Rightarrow [\sigma_1 \rightsquigarrow \beta]_\mu \rightarrow [\sigma_2 \rightsquigarrow \beta]_\mu \rightarrow [\sigma_1 \leftsquigarrow (\sigma_1 \bowtie \sigma_2) \rightsquigarrow \sigma_2]_\mu \\
l_1 \bowtie l_2 = \mathit{Span}\ (\mathit{MLens}\ \mathit{fst}\ \mathit{put}_{\mathrm{L}}\ \mathit{create}_{\mathrm{L}})\ (\mathit{MLens}\ \mathit{snd}\ \mathit{put}_{\mathrm{R}}\ \mathit{create}_{\mathrm{R}})\ \mathbf{where} \\
\quad \mathit{put}_{\mathrm{L}}\ (\_, s_2)\ s_1' = \mathbf{do}\ \{ s_2' \leftarrow l_2.\mathit{mput}\ s_2\ (l_1.\mathit{mget}\ s_1'); \mathit{return}\ (s_1', s_2') \} \\
\quad \mathit{create}_{\mathrm{L}}\ s_1 \qquad = \mathbf{do}\ \{ s_2' \leftarrow l_2.\mathit{mcreate}\ (l_1.\mathit{mget}\ s_1); \mathit{return}\ (s_1, s_2') \} \\
\quad \mathit{put}_{\mathrm{R}}\ (s_1, \_)\ s_2' = \mathbf{do}\ \{ s_1' \leftarrow l_1.\mathit{mput}\ s_1\ (l_2.\mathit{mget}\ s_2'); \mathit{return}\ (s_1', s_2') \} \\
\quad \mathit{create}_{\mathrm{R}}\ s_1 \qquad = \mathbf{do}\ \{ s_1' \leftarrow l_1.\mathit{mcreate}\ (l_2.\mathit{mget}\ s_2); \mathit{return}\ (s_1', s_2) \}
\end{array}$$

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 $ml_1 :: [S_1 \rightsquigarrow B]_M$ and $ml_2 :: [S_2 \rightsquigarrow B]_M$ are well-behaved then so is $ml_1 \bowtie ml_2 :: [S_1 \leftsquigarrow (S_1 \bowtie S_2) \rightsquigarrow S_2]_\mu$. ♢

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:

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

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 $\mathit{sp}_1 :: [A \leftsquigarrow S_1 \rightsquigarrow B]_M$ and $\mathit{sp}_2 :: [B \leftsquigarrow S_2 \rightsquigarrow C]_M$ 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 $sp :: [A \leftsquigarrow S \rightsquigarrow B]_M$, we can construct a monadic symmetric lens $sl :: [A \overset{Maybe\ S}{\longleftrightarrow} B]_M$ as follows:

```
span2smlens (left, right) = SMLens mput_R mput_L Nothing where
  mput_R (a, Just s)    = do { s' ← left.mput s a; return (right.mget s', Just s') }
  mput_R (a, Nothing)   = do { s' ← left.mcreate a; return (right.mget s', Just s') }
  mput_L (b, Just s)    = do { s' ← right.mput s b; return (left.mget s', Just s') }
  mput_L (b, Nothing)   = do { s' ← right.mcreate b; return (left.mget s', Just s') }
```

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 the *mcreate* 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 $sp :: [A \leftsquigarrow S \rightsquigarrow B]_M$ is well-behaved, then *span2smlens sp* is also well-behaved. ◇

Given $sl :: [A \overset{C}{\longleftrightarrow} B]_M$, let $S \subseteq A \times B \times C$ be the set of *consistent triples* (a, b, c), that is, those for which $sl.mput_{\mathrm{R}}\ (a, c) = return\ (b, c)$ and $sl.mput_{\mathrm{L}}\ (b, c) = return\ (a, c)$. We construct $sp :: [A \leftsquigarrow S \rightsquigarrow B]_M$ by

```
smlens2span sl = Span (MLens get_L put_L create_L) (MLens get_R put_R create_R)
where
  get_L (a, b, c)       = a
  put_L (a, b, c) a'    = do { (b', c') ← sl.mput_R (a', c); return (a', b', c') }
  create_L a            = do { (b, c) ← sl.mput_R (a, sl.missing); return (a, b, c) }
  get_R (a, b, c)       = b
  put_R (a, b, c) b'    = do { (a', c') ← sl.mput_L (b', c); return (a', b', c') }
  create_R b            = do { (a, c) ← sl.mput_L (b, sl.missing); return (a, b, c) }
```

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

Example 3.11. Consider the following monadic symmetric lens construction:

```
fail :: [() ⟵()⟶ ()]_Maybe
fail = SMLens Nothing Nothing ()
```

This is well-behaved but *smlens2span fail* is not. In fact, the set of consistent states of *fail* is empty, and each leg of the induced span is of the following form:

```
failMLens :: MLens Maybe ∅ ()
failMLens = MLens (λ_ → ()) (λ_ () → Nothing) (λ_ → Nothing)
```

which fails to satisfy (MGetPut). ◇

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

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*. ◇

To summarise: 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 axiomatisation of the symmetric monadic lenses that are essentially spans of monadic lenses (and are therefore closed under composition).

4 Equivalence of Spans

Hofmann et al. (2011) 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 (2014) introduced a definition of equivalence of spans and compared it with symmetric lens equivalence. We have considered equivalences based on isomorphism (Abou-Saleh et al. 2015a) and bisimulation (Abou-Saleh et al. 2015b). 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 $h :: S_1 \to S_2$ 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 the explicit conversion.

We consider a second definition of equivalence, inspired by Johnson and Rosebrugh (2014), 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 $h \,;\, sp_2.\mathit{left} = sp_1.\mathit{left}$ and $h \,;\, sp_2.\mathit{right} = sp_1.\mathit{right}$. The equivalence relation $\equiv_{\mathrm{s}}$ is the least equivalence relation containing $\curvearrowright$. ♢

One important consideration emphasised 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 $h.\mathit{get} \cdot c = \mathit{id}$. 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 $c = \mathit{create}$ satisfying $h.\mathit{get} \cdot c = \mathit{id}$, that is, the (CreateGet) law.

We have defined span equivalence as the reflexive, symmetric, transitive closure of $\curvearrowright$. Interestingly, even though span equivalence allows for an arbitrary sequence of (pure) lenses between the respective state spaces, it suffices to consider only spans of lenses. To prove this, we first state a lemma about the $(\bowtie)$ operation used in composition. Its proof is straightforward equational reasoning.

Lemma 4.3. Suppose $l_1 :: A \rightsquigarrow B$ and $l_2 :: C \rightsquigarrow B$ are pure lenses. Then $(l_1 \bowtie l_2).\mathit{left} \mathbin{;} l_1 = (l_1 \bowtie l_2).\mathit{right} \mathbin{;} l_2$. $\diamondsuit$

Theorem 4.4. Given $sp_1 :: [A \leftsquigarrow S_1 \rightsquigarrow B]_M$ and $sp_2 :: [A \leftsquigarrow S_2 \rightsquigarrow B]_M$, if $sp_1 \equiv_{\mathrm{s}} sp_2$ then there exists $sp :: S_1 \leftsquigarrow S \rightsquigarrow S_2$ such that $sp.\mathit{left} \mathbin{;} sp_1.\mathit{left} = sp.\mathit{right} \mathbin{;} sp_2.\mathit{left}$ and $sp.\mathit{left} \mathbin{;} sp_1.\mathit{right} = sp.\mathit{right} \mathbin{;} sp_2.\mathit{right}$. $\diamondsuit$

Proof. Let sp_1 and sp_2 be given such that $sp_1 \equiv_{\mathrm{s}} sp_2$. The proof is by induction on the length of the sequence of $\curvearrowright$ or $\curvearrowleft$ steps linking sp_1 to sp_2.

If $sp_1 = sp_2$ then the result is immediate. If $sp_1 \curvearrowright sp_2$ then we can complete a span between S_1 and S_2 using the identity lens. For the inductive case, suppose that the result holds for sequences of up to n $\curvearrowright$ or $\curvearrowleft$ steps, and suppose $sp_1 \equiv_{\mathrm{s}} sp_2$ holds in n $\curvearrowright$ or $\curvearrowleft$ steps. There are two cases, depending on the direction of the first step. If $sp_1 \curvearrowleft sp_3 \equiv_{\mathrm{s}} sp_2$ 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 $sp_1 \curvearrowright sp_3 \equiv_{\mathrm{s}} sp_2$ then by induction we must have a pure span sp between S_3 and S_2 and we must have a lens $h :: S_1 \to S_3$, so we use Lemma 4.3 to form a span $sp_0 :: S_1 \leftsquigarrow (S_1 \bowtie S_3) \rightsquigarrow S_3$ and extend $sp_0.\mathit{right}$ with $sp.\mathit{right}$ to form the required span between S_1 and S_3. $\square$

Thus, span equivalence is a doubly appropriate name for $\equiv_{\mathrm{s}}$: 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 (Abou-Saleh et al. 2015b):

Definition 4.5 (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

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

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 $sp_1.\mathit{left}$ to $sp_2.\mathit{left}$ and from $sp_1.\mathit{right}$ to $sp_2.\mathit{right}$. $\diamondsuit$

Definition 4.6 (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 from sp to sp_2. We write $sp_1 \equiv_{\mathrm{b}} sp_2$ when there is a bisimulation of spans sp_1 and sp_2. $\diamondsuit$

Figure 2 illustrates the three equivalences diagrammatically.

Proposition 4.7. Each of the relations $\equiv_{\mathrm{i}}$, $\equiv_{\mathrm{s}}$ and $\equiv_{\mathrm{b}}$ are equivalence relations on compatible spans of M-lenses and satisfy (Identity), (Assoc) and (Cong). $\diamondsuit$

Theorem 4.8. $sp_1 \equiv_{\mathrm{i}} sp_2$ implies $sp_1 \equiv_{\mathrm{s}} sp_2$, but not the converse. $\diamondsuit$

$$\begin{tikzcd}
 & S_1 \arrow[dl] \arrow[dr] \arrow[dd, leftrightarrow, "\simeq"] & \\
A & & B \\
 & S_2 \arrow[ul] \arrow[ur] &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & S_1 \arrow[dl] \arrow[dr] & \\
A & S \arrow[u] \arrow[d] & B \\
 & S_2 \arrow[ul] \arrow[ur] &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & S_1 \arrow[dl] \arrow[dr] & \\
A & R \subseteq S_1 \times S_2 \arrow[l] \arrow[r] \arrow[u, dotted, "fst"] \arrow[d, dotted, "snd"'] & B \\
 & S_2 \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

(a) (b) (c)

Fig. 2. (a) Isomorphism equivalence ($\equiv_\mathrm{i}$), (b) span equivalence ($\equiv_\mathrm{s}$), and (c) bisimulation ($\equiv_\mathrm{b}$) equivalence. In (c), the dotted arrows are base maps; all other arrows are (monadic) lenses.

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

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

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

Theorem 4.9. Given $sp_1 :: [A \leftsquigarrow S_1 \leadsto B]_M, sp_2 :: [A \leftsquigarrow S_2 \leadsto B]_M$, if $sp_1 \equiv_\mathrm{s} sp_2$ then $sp_1 \equiv_\mathrm{b} sp_2$. ♢

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 $\{(s_1, s_2) \mid l_1.\mathit{get}\ s_1 = s_2\}$, and constructing an appropriate span $sp : A \leftsquigarrow R \leadsto B$. 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. □

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

Theorem 4.10. Given $sp_1 :: A \leftsquigarrow S_1 \leadsto B, sp_2 :: A \leftsquigarrow S_2 \leadsto B$, if $sp_1 \equiv_\mathrm{b} sp_2$ then $sp_1 \equiv_\mathrm{s} sp_2$. ♢

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

This result is surprising because the two equivalences come from rather different perspectives. Johnson and Rosebrugh introduced a form of span equivalence, and showed that it implies bisimulation equivalence. They did not explicitly address the question of whether this implication is strict. However, there are some differences between their presentation and ours; the most important difference is the fact that we assume lenses to be equipped with a create function, while they consider lenses without create functions but sometimes consider spans of lenses to be "pointed", or equipped with designated initial state values. Likewise, Abou-Saleh et al. (2015b) considered bisimulation equivalence for coalgebraic bx over pointed sets (i.e. sets equipped with designated initial values). It remains to be determined whether Theorem 4.10 transfers to these settings.

We leave it as an open question to determine whether $\equiv_{\mathrm{b}}$ is equivalent to $\equiv_{\mathrm{s}}$ for spans of monadic lenses (we conjecture that they are not), or whether an analogous result to Theorem 4.10 carries over to symmetric lenses (we conjecture that it does).

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; our proposal for (asymmetric) monadic lenses improves on them because M-lenses are closed under composition for any fixed monad M. Furthermore, we investigated the symmetric case, and showed that *spans* of monadic lenses are also closed under composition, while the obvious generalisation 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 result: bisimulation and span equivalence coincide for pure spans of lenses. This last result is somewhat surprising, given that Johnson and Rosebrugh introduced (what we call) span equivalence to overcome perceived shortcomings in Hofmann et al.'s bisimulation-based notion of symmetric lens equivalence. Further investigation is necessary to determine whether this result generalises.

These results illustrate the benefits of our formulation of monadic lenses and we hope they will inspire further research and appreciation of bidirectional programming with effects.

Acknowledgements. The work was supported by the UK EPSRC-funded project *A Theory of Least Change for Bidirectional Transformations* (TLCBX Project 2013–2016) (EP/K020218/1, EP/K020919/1).

A Proofs for Sect. 2

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

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 (MPutGet), the proof is as follows:

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

□

B Proofs for Sect. 3

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

For the first part:

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

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

$$
\begin{array}{ll}
= & \llbracket \text{ monad associativity } \rrbracket \\
 & \textbf{do}\,\{(b, c') \leftarrow \textbf{do}\,\{\mathit{set}\;x; \mathit{return}\;((), ())\}; \mathit{return}\;((), c')\} \\
= & \llbracket \text{ Definition } \rrbracket \\
 & \textbf{do}\,\{(b, c') \leftarrow (\mathit{setBool}\;x).\mathit{mput}_{\mathrm{R}}\;((), ()); \mathit{return}\;((), c')\}
\end{array}
$$

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

$$
\begin{array}{ll}
 & \textbf{do}\,\{(c, s') \leftarrow \mathit{sl}.\mathit{mput}_{\mathrm{R}}\;(a, s); \mathit{sl}.\mathit{mput}_{\mathrm{L}}\;(c, s')\} \\
= & \llbracket \text{ let } s = (s_1, s_2) \text{ and } s' = (s_1''', s_2'''); \text{ definition } \rrbracket \\
 & \textbf{do}\,\{(b, s_1') \leftarrow (\mathit{setBool}\;\mathit{True}).\mathit{mput}_{\mathrm{R}}\;(a, s_1); \\
 & \qquad (c, s_2') \leftarrow (\mathit{setBool}\;\mathit{False}).\mathit{mput}_{\mathrm{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}_{\mathrm{L}}\;(c', s_2''); \\
 & \qquad (a', s_2''') \leftarrow (\mathit{setBool}\;\mathit{True}).\mathit{mput}_{\mathrm{L}}\;(b', s_1''); \\
 & \qquad \mathit{return}\;(c, (s_1''', s_2'''))\} \\
= & \llbracket \text{ monad unit } \rrbracket \\
 & \textbf{do}\,\{(b, s_1') \leftarrow (\mathit{setBool}\;\mathit{True}).\mathit{mput}_{\mathrm{R}}\;(a, s_1); \\
 & \qquad (c, s_2') \leftarrow (\mathit{setBool}\;\mathit{False}).\mathit{mput}_{\mathrm{R}}\;(b, s_2); \\
 & \qquad (b', s_2''') \leftarrow (\mathit{setBool}\;\mathit{False}).\mathit{mput}_{\mathrm{L}}\;(c', s_2'); \\
 & \qquad (a', s_2''') \leftarrow (\mathit{setBool}\;\mathit{True}).\mathit{mput}_{\mathrm{L}}\;(b', s_1'); \\
 & \qquad \mathit{return}\;(c, (s_1''', s_2'''))\} \\
= & \llbracket \text{ } \mathsf{(PutRLM)} \text{ for } \mathit{setBool}\;\mathit{False} \text{ } \rrbracket \\
 & \textbf{do}\,\{(b, s_1') \leftarrow (\mathit{setBool}\;\mathit{True}).\mathit{mput}_{\mathrm{R}}\;(a, s_1); \\
 & \qquad (c, s_2') \leftarrow (\mathit{setBool}\;\mathit{False}).\mathit{mput}_{\mathrm{R}}\;(b, s_2); \\
 & \qquad (b', s_2''') \leftarrow \mathit{return}\;(b, s_2'); \\
 & \qquad (a', s_2''') \leftarrow (\mathit{setBool}\;\mathit{False}).\mathit{mput}_{\mathrm{L}}\;(b', s_1'); \\
 & \qquad \mathit{return}\;(c, (s_1''', s_2'''))\} \\
= & \llbracket \text{ monad unit } \rrbracket \\
 & \textbf{do}\,\{(b, s_1') \leftarrow (\mathit{setBool}\;\mathit{True}).\mathit{mput}_{\mathrm{R}}\;(a, s_1); \\
 & \qquad (c, s_2') \leftarrow (\mathit{setBool}\;\mathit{False}).\mathit{mput}_{\mathrm{R}}\;(b, s_2); \\
 & \qquad (a', s_2''') \leftarrow (\mathit{setBool}\;\mathit{True}).\mathit{mput}_{\mathrm{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:

$$
\begin{array}{ll}
 & \textbf{do}\,\{(c, s') \leftarrow \mathit{sl}.\mathit{mput}_{\mathrm{R}}\;(a, s); \mathit{sl}.\mathit{mput}_{\mathrm{L}}\;(c, s'')\} \\
= & \llbracket \text{ let } s = (s_1, s_2) \text{ and } s' = (s_1''', s_2'''); \text{ Definition } \rrbracket \\
 & \textbf{do}\,\{(b, s_1') \leftarrow (\mathit{setBool}\;\mathit{True}).\mathit{mput}_{\mathrm{R}}\;(a, s_1); \\
 & \qquad (c, s_2') \leftarrow (\mathit{setBool}\;\mathit{False}).\mathit{mput}_{\mathrm{R}}\;(b, s_2); \\
 & \qquad (c', (s_1'', s_2'')) \leftarrow \mathit{return}\;(c, (s_1', s_2')); \\
 & \qquad \mathit{return}\;(c', (s_1'', s_2''))\} \\
= & \llbracket \text{ Monad unit } \rrbracket \\
 & \textbf{do}\,\{(b, s_1') \leftarrow (\mathit{setBool}\;\mathit{True}).\mathit{mput}_{\mathrm{R}}\;(a, s_1); \\
 & \qquad (c, s_2') \leftarrow (\mathit{setBool}\;\mathit{False}).\mathit{mput}_{\mathrm{R}}\;(b, s_2); \\
 & \qquad \mathit{return}\;(c, (s_1', s_2'))\}
\end{array}
$$

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 $ml_1 :: [\sigma_1 \leadsto \beta]_\mu$ and $ml_2 :: [\sigma_2 \leadsto \beta]_\mu$ are well-behaved then so is $ml_1 \bowtie ml_2 :: [\sigma_1 \leftrightsquigarrow (\sigma_1 \bowtie \sigma_2) \leadsto \sigma_2]_\mu$. ♢

Proof. It suffices to consider the two lenses $l_1 = \mathit{MLens}\ \mathit{fst}\ \mathit{put}_{\mathrm{L}}\ \mathit{create}_{\mathrm{L}}$ and $l_2 = \mathit{MLens}\ \mathit{snd}\ \mathit{put}_{\mathrm{R}}\ \mathit{create}_{\mathrm{R}}$ in isolation. Moreover, the two cases are completely symmetric, so we only show the first.

For (MGetPut), we show:

$$
\begin{array}{ll}
& \mathbf{do}\ \{l_1.\mathit{mput}\ (s_1, s_2)\ (l_1.\mathit{mget}\ (s_1, s_2))\} \\
= & \llbracket\ \text{definition}\ \rrbracket \\
& \mathbf{do}\ \{\mathit{put}_{\mathrm{L}}\ (s_1, s_2)\ (\mathit{fst}\ (s_1, s_2))\} \\
= & \llbracket\ \text{definition of } \mathit{put}_{\mathrm{L}} \text{ and } \mathit{fst}\ \rrbracket \\
& \mathbf{do}\ \{s_2' \leftarrow ml_2.\mathit{mput}\ s_2\ (ml_1.\mathit{mget}\ s_1)\} \\
= & \llbracket\ (s_1, s_2) \text{ consistent}\ \rrbracket \\
& \mathbf{do}\ \{s_2' \leftarrow ml_2.\mathit{mput}\ s_2\ (ml_2.\mathit{mget}\ s_2)\} \\
= & \llbracket\ (\mathsf{MGetPut})\ \rrbracket \\
& \mathit{return}\ s
\end{array}
$$

The proof for (MPutGet) goes as follows. Note that it holds by construction, without appealing to well-behavedness of ml_1 or ml_2.

$$
\begin{array}{ll}
& \mathbf{do}\ \{(s_1', s_2') \leftarrow l_1.\mathit{mput}\ (s_1, s_2)\ a; \mathit{return}\ ((s_1', s_2'), l_1.\mathit{mget}\ (s_1', s_2'))\} \\
= & \llbracket\ \text{definition}\ \rrbracket \\
& \mathbf{do}\ \{(s_1', s_2') \leftarrow \mathit{put}_{\mathrm{R}}\ (s_1, s_2)\ a; \mathit{return}\ ((s_1', s_2'), \mathit{fst}\ (s_1', s_2'))\} \\
= & \llbracket\ \text{definition}\ \rrbracket \\
& \mathbf{do}\ \{s_2'' \leftarrow ml_2.\mathit{mput}\ s_2\ (ml_1.\mathit{mget}\ a); (s_1', s_2') \leftarrow \mathit{return}\ (a, s_2''); \\
& \qquad \mathit{return}\ ((s_1', s_2'), \mathit{fst}\ (s_1', s_2'))\} \\
= & \llbracket\ \text{definition of } \mathit{fst}\ \rrbracket \\
& \mathbf{do}\ \{s_2'' \leftarrow ml_2.\mathit{mput}\ s_2\ (ml_1.\mathit{mget}\ a); (s_1', s_2') \leftarrow \mathit{return}\ (a, s_2''); \\
& \qquad \mathit{return}\ ((s_1', s_2'), s_1') \\
= & \llbracket\ \text{monad laws}\ \rrbracket \\
& \mathbf{do}\ \{s_2'' \leftarrow ml_2.\mathit{mput}\ s_2\ (ml_1.\mathit{mget}\ a); (s_1', s_2') \leftarrow \mathit{return}\ (a, s_2''); \\
& \qquad \mathit{return}\ ((s_1', s_2'), a)\} \\
= & \llbracket\ \text{definition}\ \rrbracket \\
& \mathbf{do}\ \{(s_1', s_2') \leftarrow \mathit{put}_{\mathrm{L}}\ (s_1, s_2)\ a; \mathit{return}\ ((s_1', s_2'), a)\} \\
= & \llbracket\ \text{definition}\ \rrbracket \\
& \mathbf{do}\ \{(s_1', s_2') \leftarrow l_1.\mathit{mput}\ (s_1, s_2)\ a; \mathit{return}\ ((s_1', s_2'), a)\}
\end{array}
$$

The proof for (MCreateGet) is similar.

Finally, we show that $\mathit{put}_{\mathrm{L}} :: (\sigma_1 \bowtie \sigma_2) \to \sigma_1 \to \mu\ (\sigma_1 \bowtie \sigma_2)$, 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, $ml_1.\mathit{mget}\ s_1 = ml_2.\mathit{mget}\ s_2$. We must show that any value returned by $\mathit{put}_{\mathrm{L}}$ also satisfies this consistency criterion. By definition,

$$\mathit{put}_{\mathrm{L}}\ (s_1, s_2)\ s'_1 = \mathbf{do}\ \{\, s'_2 \leftarrow \mathit{ml}_2.\mathit{mput}\ s_2\ (\mathit{ml}_1.\mathit{mget}\ s'_1); \mathit{return}\ (s'_1, s'_2)\,\}$$

By (MPutGet), any s'_2 resulting from $\mathit{ml}_2.\mathit{mput}\ s_2\ (\mathit{ml}_1.\mathit{mget}\ s'_1)$ will satisfy $\mathit{ml}_2.\mathit{mget}\ s'_2 = \mathit{ml}_1.\mathit{mget}\ s'_1$. The proof that $\mathit{create}_{\mathrm{L}} :: \sigma_1 \to \mu\ (\sigma_1 \bowtie \sigma_2)$ is similar, but simpler. □

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

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

We need to show that

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

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

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

$$\begin{array}{l}
\quad \mathbf{do}\ \{\, (b', \mathit{mc}') \leftarrow \mathit{sl}.\mathit{mput}_{\mathrm{R}}\ (a, \mathit{Nothing}); \mathit{sl}.\mathit{mput}_{\mathrm{L}}\ (b', \mathit{mc}')\,\} \\
= \quad [\![\ \text{Definition}\ ]\!] \\
\quad \mathbf{do}\ \{\, s' \leftarrow \mathit{sp}.\mathit{left}.\mathit{mcreate}\ a; (b', \mathit{mc}') \leftarrow \mathit{return}\ (\mathit{sp}.\mathit{right}.\mathit{mget}\ s', \mathit{Just}\ s'); \\
\qquad \mathit{sl}.\mathit{mput}_{\mathrm{L}}\ (b', \mathit{mc}')\,\} \\
= \quad [\![\ \text{monad unit}\ ]\!] \\
\quad \mathbf{do}\ \{\, s' \leftarrow \mathit{sp}.\mathit{left}.\mathit{mcreate}\ a; \mathit{sl}.\mathit{mput}_{\mathrm{L}}\ (\mathit{sp}.\mathit{right}.\mathit{mget}\ s', \mathit{Just}\ s')\,\} \\
= \quad [\![\ \text{definition}\ ]\!] \\
\quad \mathbf{do}\ \{\, s' \leftarrow \mathit{sp}.\mathit{left}.\mathit{mcreate}\ a; s'' \leftarrow \mathit{sp}.\mathit{right}.\mathit{mput}\ s'\ (\mathit{sp}.\mathit{right}.\mathit{mget}\ s'); \\
\qquad \mathit{return}\ (\mathit{sp}.\mathit{left}.\mathit{mget}\ s'', \mathit{Just}\ s'')\,\} \\
= \quad [\![\ (\textsf{MGetPut})\ ]\!] \\
\quad \mathbf{do}\ \{\, s' \leftarrow \mathit{sp}.\mathit{left}.\mathit{mcreate}\ a; s'' \leftarrow \mathit{return}\ s'; \\
\qquad \mathit{return}\ (\mathit{sp}.\mathit{left}.\mathit{mget}\ s'', \mathit{Just}\ s'')\,\} \\
= \quad [\![\ \text{monad unit}\ ]\!] \\
\quad \mathbf{do}\ \{\, s' \leftarrow \mathit{sp}.\mathit{left}.\mathit{mcreate}\ a; \mathit{return}\ (\mathit{sp}.\mathit{left}.\mathit{mget}\ s', \mathit{Just}\ s')\,\} \\
= \quad [\![\ (\textsf{MCreateGet})\ ]\!] \\
\quad \mathbf{do}\ \{\, s' \leftarrow \mathit{sp}.\mathit{left}.\mathit{mcreate}\ a; \mathit{return}\ (a, \mathit{Just}\ s')\,\} \\
= \quad [\![\ \text{monad unit}\ ]\!] \\
\quad \mathbf{do}\ \{\, s' \leftarrow \mathit{sp}.\mathit{left}.\mathit{mcreate}\ a; (b', \mathit{mc}') \leftarrow (\mathit{sp}.\mathit{right}.\mathit{get}\ s', \mathit{Just}\ s'); \\
\qquad \mathit{return}\ (a, \mathit{mc}')\,\} \\
= \quad [\![\ \text{Definition}\ ]\!] \\
\quad \mathbf{do}\ \{\, (b', \mathit{mc}') \leftarrow \mathit{sl}.\mathit{mput}_{\mathrm{R}}\ (a, \mathit{Nothing}); \mathit{return}\ (a, \mathit{mc}')\,\}
\end{array}$$

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

$$\begin{array}{l}
\quad \mathbf{do}\ \{\, (b', \mathit{mc}') \leftarrow \mathit{sl}.\mathit{mput}_{\mathrm{R}}\ (a, \mathit{Just}\ s); \mathit{sl}.\mathit{mput}_{\mathrm{L}}\ (b', \mathit{mc}')\,\} \\
= \quad [\![\ \text{Definition}\ ]\!] \\
\quad \mathbf{do}\ \{\, s' \leftarrow \mathit{sp}.\mathit{left}.\mathit{mput}\ s\ a; (b', \mathit{mc}') \leftarrow (\mathit{sp}.\mathit{right}.\mathit{mget}\ s', \mathit{Just}\ s');
\end{array}$$

$$\begin{array}{ll}
& \quad \mathit{sl.mput}_{\mathrm{L}}\ (b', \mathit{mc}')\,\} \\
= & \llbracket\ \text{monad unit}\ \rrbracket \\
& \mathbf{do}\ \{\,s' \leftarrow \mathit{sp.left.mput}\ s\ a; \mathit{sl.mput}_{\mathrm{L}}\ (\mathit{sp.right.mget}\ s', \mathit{Just}\ s')\,\} \\
= & \llbracket\ \text{definition}\ \rrbracket \\
& \mathbf{do}\ \{\,s' \leftarrow \mathit{sp.left.mput}\ s\ a; s'' \leftarrow \mathit{sp.right.mput}\ s'\ (\mathit{sp.right.mget}\ s'); \\
& \quad \mathit{return}\ (\mathit{sp.left.mget}\ s'', \mathit{Just}\ s'')\,\} \\
= & \llbracket\ (\mathsf{MGetPut})\ \rrbracket \\
& \mathbf{do}\ \{\,s' \leftarrow \mathit{sp.left.mput}\ s\ a; s'' \leftarrow \mathit{return}\ s'; \\
& \quad \mathit{return}\ (\mathit{sp.left.mget}\ s'', \mathit{Just}\ s'')\,\} \\
= & \llbracket\ \text{monad unit}\ \rrbracket \\
& \mathbf{do}\ \{\,s' \leftarrow \mathit{sp.left.mput}\ s\ a; \mathit{return}\ (\mathit{sp.left.mget}\ s', \mathit{Just}\ s')\,\} \\
= & \llbracket\ (\mathsf{MPutGet})\ \rrbracket \\
& \mathbf{do}\ \{\,s' \leftarrow \mathit{sp.left.mput}\ s\ a; \mathit{return}\ (a, \mathit{Just}\ s')\,\} \\
= & \llbracket\ \text{monad unit}\ \rrbracket \\
& \mathbf{do}\ \{\,s' \leftarrow \mathit{sp.left.mput}\ s\ a; (b', \mathit{mc}') \leftarrow (\mathit{sp.right.get}\ s', \mathit{Just}\ s'); \\
& \quad \mathit{return}\ (a, \mathit{mc}')\,\} \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
& \mathbf{do}\ \{\,(b', \mathit{mc}') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\ (a, \mathit{Just}\ c); \mathit{return}\ (a, \mathit{mc}')\,\}
\end{array}$$

□

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*. ♢

Proof. First we show that, given a symmetric lens *sl*, the operations of $\mathit{sp} = \mathit{smlens2span}\ \mathit{sl}$ preserve consistency of the state. Assume (a, b, c) is consistent. To show that $\mathit{sp.left.mput}\ (a, b, c)\ a'$ is consistent for any a', we have to show that (a', b', c') is consistent, where a' is arbitrary and $\mathit{return}\ (b', c') = \mathit{sl.mput}_{\mathrm{R}}\ (a', c)$. For one half of consistency, we have:

$$\begin{array}{ll}
& \mathit{sl.mput}_{\mathrm{L}}\ (b', c') \\
= & \llbracket\ \mathit{sl.mput}_{\mathrm{R}}\ (a', c) = \mathit{return}\ (b', c'), \text{ and } (\mathsf{PutRLM})\ \rrbracket \\
& \mathit{return}\ (a', c')
\end{array}$$

The proof that $\mathit{sl.mput}_{\mathrm{R}}\ (a', c') = \mathit{return}\ (b', c')$ is symmetric.

$$\begin{array}{ll}
& \mathit{sl.mput}_{\mathrm{R}}\ (a', c') \\
= & \llbracket\ \text{above, and } (\mathsf{PutLRM})\ \rrbracket \\
& \mathit{return}\ (b', c')
\end{array}$$

as required. The proof that $\mathit{sp.right.mput}\ (a, b, c)\ b'$ is consistent is dual.

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

$$\begin{array}{ll}
& \mathit{sp.left.mput}\ (a, b, c)\ (\mathit{sp.left.mget}\ (a, b, c)) \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
& \mathbf{do}\ \{\,(b', c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\ (a, c); \mathit{return}\ (a, b', c')\,\} \\
= & \llbracket\ \text{Consistency of } (a, b, c)\ \rrbracket
\end{array}$$

$$\begin{array}{ll}
 & \mathbf{do}\,\{(b',c') \leftarrow \mathit{return}\;(b,c); \mathit{return}\;(a,b',c')\} \\
= & \llbracket\ \text{monad unit}\ \rrbracket \\
 & \mathit{return}\;(a,b,c)
\end{array}$$

For (MPutGet), we have:

$$\begin{array}{ll}
 & \mathbf{do}\,\{s' \leftarrow \mathit{sp.left.put}\;(a,b,c)\;a'; \mathit{return}\;(s', \mathit{sp.left.mget}\;s')\} \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
 & \mathbf{do}\,\{(b',c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\;(a',c); s' \leftarrow \mathit{return}\;(a',b',c'); \\
 & \qquad \mathit{return}\;(s', \mathit{sp.left.mget}\;s')\} \\
= & \llbracket\ \text{monad unit}\ \rrbracket \\
 & \mathbf{do}\,\{(b',c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\;(a',c); \\
 & \qquad \mathit{return}\;((a',b',c'), \mathit{sp.left.mget}\;(a',b',c'))\} \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
 & \mathbf{do}\,\{(b',c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\;(a',c); \mathit{return}\;((a',b',c'),a')\} \\
= & \llbracket\ \text{monad unit}\ \rrbracket \\
 & \mathbf{do}\,\{(b',c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\;(a',c); s' \leftarrow \mathit{return}\;(a',b',c'); \mathit{return}\;(s',a')\} \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
 & \mathbf{do}\,\{s' \leftarrow \mathit{sp.left.put}\;(a,b,c)\;a'; \mathit{return}\;(s',a')\}
\end{array}$$

The proof for (MCreateGet) is similar.

$$\begin{array}{ll}
 & \mathbf{do}\,\{s \leftarrow \mathit{sp.left.create}\;a; \mathit{return}\;(s, \mathit{sp.left.mget}\;s)\} \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
 & \mathbf{do}\,\{(b',c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\;(a,c); s \leftarrow \mathit{return}\;(a,b',c'); \\
 & \qquad \mathit{return}\;(s, \mathit{sp.left.mget}\;s)\} \\
= & \llbracket\ \text{monad unit}\ \rrbracket \\
 & \mathbf{do}\,\{(b',c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\;(a,c); \\
 & \qquad \mathit{return}\;((a,b',c'), \mathit{sp.left.mget}\;(a,b',c'))\} \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
 & \mathbf{do}\,\{(b',c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\;(a,c); \mathit{return}\;((a,b',c'),a)\} \\
= & \llbracket\ \text{monad unit}\ \rrbracket \\
 & \mathbf{do}\,\{(b',c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\;(a,c); s \leftarrow \mathit{return}\;(a,b',c'); \mathit{return}\;(s,a)\} \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
 & \mathbf{do}\,\{s \leftarrow \mathit{sp.left.create}\;a; \mathit{return}\;(a,s)\}
\end{array}$$

□

C Proofs for Sect. 4

Lemma 4.3. 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. Let $(l, r) = l_1 \bowtie l_2$. We show that each component of $l\,;\,l_1$ equals the corresponding component of $r\,;\,l_2$.

For *get*:

$$\begin{array}{ll}
 & (l\,;\,l_1).\mathit{get}\;(a,c) \\
= & \llbracket\ \text{Definition}\ \rrbracket
\end{array}$$

$$\begin{array}{ll}
& l_1.\mathit{get}\ (l.\mathit{get}\ (a,c)) \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
& l_1.\mathit{get}\ a \\
= & \llbracket\ \text{Consistency}\ \rrbracket \\
& l_2.\mathit{get}\ c \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
& l_2.\mathit{get}\ (r.\mathit{get}\ (a,c)) \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
& (r\,;l_2).\mathit{get}\ (a,c)
\end{array}$$

For *put*:

$$\begin{array}{ll}
& (l\,;l_1).\mathit{put}\ (a,c)\ b \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
& l.\mathit{put}\ (a,c)\ (l_1.\mathit{put}\ (l.\mathit{get}\ (a,c))\ b) \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
& l.\mathit{put}\ (a,c)\ (l_1.\mathit{put}\ a\ b) \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
& \mathbf{let}\ a' = l_1.\mathit{put}\ a\ b\ \mathbf{in} \\
& \mathbf{let}\ c' = l_2.\mathit{put}\ c\ (l_1.\mathit{get}\ a')\ \mathbf{in}\ (a',c') \\
= & \llbracket\ \text{inline } \mathbf{let}\ \rrbracket \\
& (l_1.\mathit{put}\ a\ b, l_2.\mathit{put}\ c\ (l_1.\mathit{get}\ (l_1.\mathit{put}\ a\ b))) \\
= & \llbracket\ (\mathsf{PutGet})\ \rrbracket \\
& (l_1.\mathit{put}\ a\ b, l_2.\mathit{put}\ c\ b) \\
= & \llbracket\ \text{reverse above steps}\ \rrbracket \\
& (r\,;l_2).\mathit{put}\ (a,c)\ b
\end{array}$$

Finally, for *create*:

$$\begin{array}{ll}
& (l\,;l_1).\mathit{create}\ b \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
& l.\mathit{create}\ (l_1.\mathit{create}\ b) \\
= & \llbracket\ \text{Definition}\ \rrbracket \\
& \mathbf{let}\ c = l_2.\mathit{create}\ (l_1.\mathit{get}\ (l_1.\mathit{create}\ b))\ \mathbf{in}\ (l_1.\mathit{create}\ b, c) \\
= & \llbracket\ (\mathsf{CreateGet})\ \rrbracket \\
& \mathbf{let}\ c = l_2.\mathit{create}\ b\ \mathbf{in}\ (l_1.\mathit{create}\ b, c) \\
= & \llbracket\ \text{Inline } \mathbf{let}\ \rrbracket \\
& (l_1.\mathit{create}\ b, l_2.\mathit{create}\ b) \\
= & \llbracket\ \text{reverse above steps}\ \rrbracket \\
& (r\,;l_2).\mathit{create}\ b
\end{array}$$

□

Theorem 4.9. 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 $sp_1 \equiv_{\mathrm{b}} sp_2$. ♢

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* is a base map from *sp* to sp_1 and

snd is a base map from sp to sp_2. We take $R = \{(s_1, s_2) \mid s_2 = l.get\ (s_1)\}$ and proceed as follows:

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

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:

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

For (MPutGet) we have:

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

Finally, for (MCreateGet) we have:

$$\begin{array}{ll}
& \mathbf{do}\ \{(s_1, s_2) \leftarrow l_0.mcreate\ a; return\ ((s_1, s_2), l_0.mget\ (s_1, s_2))\} \\
= & \llbracket \text{ Definition } \rrbracket
\end{array}$$

$$\begin{array}{ll}
& \mathbf{do}\,\{\,s'_1 \leftarrow l_1.\mathit{mcreate}\ a; (s_1, s_2) \leftarrow \mathit{return}\ (s'_1, l.\mathit{get}\ s'_1);\\
& \qquad \mathit{return}\ ((s_1, s_2), l_1.\mathit{mget}\ s_1)\,\}\\
= & \llbracket\ \text{Monad unit}\ \rrbracket\\
& \mathbf{do}\,\{\,s'_1 \leftarrow l_1.\mathit{mcreate}\ a; \mathit{return}\ ((s'_1, l.\mathit{get}\ s'_1), l_1.\mathit{mget}\ s'_1)\,\}\\
= & \llbracket\ (\mathsf{MCreateGet})\ \rrbracket\\
& \mathbf{do}\,\{\,s'_1 \leftarrow l_1.\mathit{mcreate}\ a; \mathit{return}\ ((s'_1, l.\mathit{get}\ s'_1), a)\,\}\\
= & \llbracket\ \text{Monad unit}\ \rrbracket\\
& \mathbf{do}\,\{\,s'_1 \leftarrow l_1.\mathit{mcreate}\ a; (s_1, s_2) \leftarrow \mathit{return}\ (s'_1, l.\mathit{get}\ s'_1);\\
& \qquad \mathit{return}\ ((s_1, s_2), a)\,\}\\
= & \llbracket\ \text{Definition}\ \rrbracket\\
& \mathbf{do}\,\{\,(s_1, s_2) \leftarrow l_0.\mathit{mcreate}\ a; \mathit{return}\ ((s_1, s_2), a)\,\}
\end{array}$$

Next, we show that *fst* is a base map from l_0 to l_1 and *snd* is a base map from l_0 to l_2. It is easy to show that *fst* is a base map from l_0 to l_1 by unfolding definitions and applying of monad laws. To show that *snd* is a base map from l_0 to l_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:

$$\begin{array}{ll}
& l_0.\mathit{mget}\ (s_1, s_2)\\
= & \llbracket\ \text{Definition}\ \rrbracket\\
& l_1.\mathit{mget}\ s_1\\
= & \llbracket\ \text{Assumption}\ l; l_2 = l_1\ \rrbracket\\
& (l\,;\,l_2).\mathit{mget}\ s_1\\
= & \llbracket\ \text{Definition}\ \rrbracket\\
& l_2.\mathit{mget}\ (l.\mathit{get}\ s_1)\\
= & \llbracket\ (s_1, s_2) \in R\ \rrbracket\\
& l_2.\mathit{mget}\ s_2
\end{array}$$

For the *mput* equation:

$$\begin{array}{ll}
& \mathbf{do}\,\{\,(s'_1, s'_2) \leftarrow l_0.\mathit{mput}\ (s_1, s_2)\ a; \mathit{return}\ s'_2\,\}\\
= & \llbracket\ \text{Definition}\ \rrbracket\\
& \mathbf{do}\,\{\,s''_1 \leftarrow l_1.\mathit{mput}\ s_1\ a; (s'_1, s'_2) \leftarrow \mathit{return}\ (s''_1, l.\mathit{get}\ s''_1); \mathit{return}\ s'_2\,\}\\
= & \llbracket\ \text{Monad laws}\ \rrbracket\\
& \mathbf{do}\,\{\,s''_1 \leftarrow l_1.\mathit{mput}\ s_1\ a; \mathit{return}\ (l.\mathit{get}\ s''_1)\,\}\\
= & \llbracket\ l\,;\,l_2 = l_1\ \rrbracket\\
& \mathbf{do}\,\{\,s''_1 \leftarrow (l\,;\,l_2).\mathit{mput}\ s_1\ a; \mathit{return}\ (l.\mathit{get}\ s''_1)\,\}\\
= & \llbracket\ \text{Definition}\ \rrbracket\\
& \mathbf{do}\,\{\,s''_2 \leftarrow l_2.\mathit{mput}\ (l.\mathit{get}\ s_1)\ a; s''_1 \leftarrow \mathit{return}\ (l.\mathit{put}\ s_1\ s''_2); \mathit{return}\ (l.\mathit{get}\ s''_1)\,\}\\
= & \llbracket\ \text{Monad laws}\ \rrbracket\\
& \mathbf{do}\,\{\,s''_2 \leftarrow l_2.\mathit{mput}\ (l.\mathit{get}\ s_1)\ a; \mathit{return}\ (l.\mathit{get}\ (l.\mathit{put}\ s_1\ s''_2))\,\}\\
= & \llbracket\ (\mathsf{PutGet})\ \rrbracket
\end{array}$$

$$
\begin{array}{ll}
& \mathbf{do}\ \{ s_2'' \leftarrow l_2.\mathit{mput}\ (l.\mathit{get}\ s_1)\ a; \mathit{return}\ s_2'' \} \\
= & \llbracket\ (s_1, s_2) \in R \text{ so } l.\mathit{get}\ s_1 = s_2\ \rrbracket \\
& \mathbf{do}\ \{ s_2'' \leftarrow l_2.\mathit{mput}\ s_2\ a; \mathit{return}\ s_2'' \} \\
= & \llbracket\ \text{Monad laws}\ \rrbracket \\
& l_2.\mathit{mput}\ s_2\ a
\end{array}
$$

For the *mcreate* equation:

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

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

Theorem 4.10. 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 $sp_1 \equiv_{\mathrm{s}} sp_2$. ♢

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 $sp_0 :: A \leftsquigarrow R \rightsquigarrow B$ constituting a bisimulation $sp_1 \equiv_{\mathrm{b}} sp_2$. Let $sp_0 = (l_0, r_0)$. 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:

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

$$\begin{aligned}
&l_0.\mathit{create}\ a = (l_1.\mathit{create}\ a, l_2.\mathit{create}\ a) \in R \\
&r_0.\mathit{create}\ b = (r_1.\mathit{create}\ b, r_2.\mathit{create}\ b) \in R
\end{aligned}$$

which also implies the following identities, which we call *twists*:

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

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

$$\begin{aligned}
&l.\mathit{get} && = \mathit{fst} \\
&l.\mathit{put}\ (s_1, s_2)\ s_1' && = l_0.\mathit{put}\ (s_1, s_2)\ (l_1.\mathit{get}\ s_1') \\
&l.\mathit{create}\ s_1 && = l_0.\mathit{create}\ (l_1.\mathit{get}\ s_1) \\
&r.\mathit{get} && = \mathit{snd} \\
&r.\mathit{put}\ (s_1, s_2)\ s_2' && = l_0.\mathit{put}\ (s_1, s_2)\ (l_2.\mathit{get}\ s_2') \\
&r.\mathit{create}\ s_2 && = l_0.\mathit{create}\ (l_2.\mathit{get}\ s_2)
\end{aligned}$$

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*:

$$\begin{aligned}
& (l\ ;\ l_1).\mathit{get}\ (s_1, s_2) \\
= {} & \quad \llbracket \text{ definition } \rrbracket \\
& l_1.\mathit{get}\ (l.\mathit{get}\ (s_1, s_2)) \\
= {} & \quad \llbracket \text{ definition of } l.\mathit{get} = \mathit{fst}\text{, } \mathit{fst} \text{ commutes with } \mathit{get} \ \rrbracket \\
& l_0.\mathit{get}\ (s_1, s_2) \\
= {} & \quad \llbracket \text{ reverse reasoning } \rrbracket \\
& (r\ ;\ l_2).\mathit{get}\ (s_1, s_2)
\end{aligned}$$

For *put*, we have:

$$\begin{aligned}
& (l\ ;\ l_1).\mathit{put}\ (s_1, s_2)\ a \\
= {} & \quad \llbracket \text{ Definition } \rrbracket \\
& l.\mathit{put}\ (s_1, s_2)\ (l_1.\mathit{put}\ s_1\ a) \\
= {} & \quad \llbracket \text{ Definition } \rrbracket \\
& l_0.\mathit{put}\ (s_1, s_2)\ (l_1.\mathit{get}\ (l_1.\mathit{put}\ s_1\ a)) \\
= {} & \quad \llbracket \text{ (PutGet) for } l_1 \ \rrbracket \\
& l_0.\mathit{put}\ (s_1, s_2)\ a \\
= {} & \quad \llbracket \text{ (PutGet) for } l_2 \ \rrbracket \\
& l_0.\mathit{put}\ (s_1, s_2)\ (l_2.\mathit{get}\ (l_2.\mathit{put}\ s_2\ a))
\end{aligned}$$

$$
\begin{array}{ll}
= & \llbracket \text{ Definition } \rrbracket \\
 & r.\mathit{put}\ (s_1, s_2)\ (l_2.\mathit{put}\ s_2\ a) \\
= & \llbracket \text{ Definition } \rrbracket \\
 & (r \mathbin{;} l_2).\mathit{put}\ (s_1, s_2)\ a
\end{array}
$$

Finally, for *create* we have:

$$
\begin{array}{ll}
 & (l \mathbin{;} l_1).\mathit{create}\ a \\
= & \llbracket \text{ Definition } \rrbracket \\
 & l.\mathit{create}\ (l_1.\mathit{create}\ a) \\
= & \llbracket \text{ Definition } \rrbracket \\
 & l_0.\mathit{create}\ (l_1.\mathit{get}\ (l_1.\mathit{create}\ a)) \\
= & \llbracket \text{ (\textsf{CreateGet}) for } l_1 \rrbracket \\
 & l_0.\mathit{create}\ a \\
= & \llbracket \text{ (\textsf{CreateGet}) for } l_2 \rrbracket \\
 & l_0.\mathit{create}\ (l_2.\mathit{get}\ (l_2.\mathit{create}\ a)) \\
= & \llbracket \text{ Definition } \rrbracket \\
 & r.\mathit{create}\ (l_2.\mathit{create}\ a) \\
= & \llbracket \text{ Definition } \rrbracket \\
 & (r \mathbin{;} l_2).\mathit{create}\ a
\end{array}
$$

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

$$
\begin{array}{ll}
 & (l \mathbin{;} r_1).\mathit{get}\ (s_1, s_2) \\
= & \llbracket \text{ Definition } \rrbracket \\
 & r_1.\mathit{get}\ (l.\mathit{get}\ (s_1, s_2)) \\
= & \llbracket \text{ definition of } l.\mathit{get} = \mathit{fst}, \mathit{fst} \text{ commutes with } r_1.\mathit{get} \rrbracket \\
 & r_0.\mathit{get}\ (s_1, s_2) \\
= & \llbracket \text{ reverse above reasoning } \rrbracket \\
 & (r \mathbin{;} r_2).\mathit{get}\ (s_1, s_2)
\end{array}
$$

For *put*, we have:

$$
\begin{array}{ll}
 & (l \mathbin{;} r_1).\mathit{put}\ (s_1, s_2)\ b \\
= & \llbracket \text{ Definition } \rrbracket \\
 & l.\mathit{put}\ (s_1, s_2)\ (r_1.\mathit{put}\ s_1\ b) \\
= & \llbracket \text{ Definition } \rrbracket \\
 & l_0.\mathit{put}\ (s_1, s_2)\ (l_1.\mathit{get}\ (r_1.\mathit{put}\ s_1\ b)) \\
= & \llbracket \text{ Twist equation } \rrbracket \\
 & l_0.\mathit{put}\ (s_1, s_2)\ (l_2.\mathit{get}\ (r_2.\mathit{put}\ s_2\ b)) \\
= & \llbracket \text{ Definition } \rrbracket \\
 & r.\mathit{put}\ (s_1, s_2)\ (r_2.\mathit{put}\ s_2\ b) \\
= & \llbracket \text{ Definition } \rrbracket \\
 & (r \mathbin{;} r_2).\mathit{put}\ (s_1, s_2)\ b
\end{array}
$$

Finally, for *create* we have:

$$
\begin{array}{ll}
 & (l \mathbin{;} r_1).\mathit{create}\ b \\
= & \llbracket \text{ Definition } \rrbracket
\end{array}
$$

$$\begin{array}{ll}
& l.create\ (r_1.create\ b) \\
= & \llbracket \text{ Definition } \rrbracket \\
& l_0.create\ (l_1.get\ (r_1.create\ b)) \\
= & \llbracket \text{ Twist equation } \rrbracket \\
& l_0.create\ (l_2.get\ (r_2.create\ b)) \\
= & \llbracket \text{ Definition } \rrbracket \\
& r.create\ (r_2.create\ b) \\
= & \llbracket \text{ Definition } \rrbracket \\
& (r \mathbin{;} r_2).create\ b
\end{array}$$

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):

$$\begin{array}{ll}
& l.get\ (l.put\ (s_1, s_2)\ s_1') \\
= & \llbracket \text{ Definition } \rrbracket \\
& fst\ (l_0.put\ (s_1, s_2)\ (l_1.get\ s_1')) \\
= & \llbracket\ fst \text{ commutes with } put\ \rrbracket \\
& l_1.put\ s_1\ (l_1.get\ s_1')) \\
= & \llbracket \text{ (GetPut) for } l_1\ \rrbracket \\
& s_1'
\end{array}$$

For (PutGet):

$$\begin{array}{ll}
& l.put\ (s_1, s_2)\ (l.get\ (s_1, s_2)) \\
= & \llbracket \text{ Definition } \rrbracket \\
& l_0.put\ (s_1, s_2)\ (l_1.get\ s_1) \\
= & \llbracket \text{ Eta-expansion for pairs } \rrbracket \\
& (fst\ (l_0.put\ (s_1, s_2)\ (l_1.get\ s_1)), snd\ (l_0.put\ (s_1, s_2)\ (l_1.get\ s_1))) \\
= & \llbracket\ fst, snd \text{ commutes with } put\ \rrbracket \\
& (l_1.put\ s_1\ (l_1.get\ s_1), l_2.put\ s_2\ (l_1.get\ s_1)) \\
= & \llbracket\ l_1.get\ s_1 = l_2.get\ s_2\ \rrbracket \\
& (l_1.put\ s_1\ (l_1.get\ s_1), l_2.put\ s_2\ (l_2.get\ s_2)) \\
= & \llbracket \text{ (PutGet) for } l_1, l_2\ \rrbracket \\
& (s_1, s_2)
\end{array}$$

For (CreateGet):

$$\begin{array}{ll}
& l.create\ (l.get\ (s_1, s_2)) \\
= & \llbracket \text{ Definition } \rrbracket \\
& l_0.create\ (l_1.get\ s_1) \\
= & \llbracket \text{ Eta-expansion for pairs } \rrbracket \\
& (fst\ (l_0.create\ (l_1.get\ s_1)), snd\ (l_0.create\ (l_1.get\ s_1))) \\
= & \llbracket\ fst, snd \text{ commutes with } put\ \rrbracket \\
& (l_1.create\ (l_1.get\ s_1), l_1.create\ (l_1.get\ s_1)) \\
= & \llbracket\ l_1.get\ s_1 = l_2.get\ s_2\ \rrbracket \\
& (l_1.create\ (l_1.get\ s_1), l_1.create\ (l_2.get\ s_2)) \\
= & \llbracket \text{ (CreateGet) } \rrbracket \\
& (s_1, s_2)
\end{array}$$

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 $sp_1 \equiv_{\mathrm{s}} sp_2$. $\square$

References

Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: Notions of bidirectional computation and entangled state monads. In: Hinze, R., Voigtländer, J. (eds.) MPC 2015. LNCS, vol. 9129, pp. 187–214. Springer, Heidelberg (2015a)

Abou-Saleh, F., McKinna, J., Gibbons, J.: Coalgebraic aspects of bidirectional computation. In: BX 2015, CEUR-WS, vol. 1396, pp. 15–30 (2015b)

Cheney, J., McKinna, J., Stevens, P., Gibbons, J., Abou-Saleh, F.: Entangled state monads. In: Terwilliger and Hidaka (2014)

Diviánszky, P.: LGtk API correction. http://people.inf.elte.hu/divip/LGtk/CorrectedAPI.html

Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. TOPLAS **29**(3), 17 (2007)

Foster, N., Matsuda, K., Voigtländer, J.: Three complementary approaches to bidirectional programming. In: Gibbons, J. (ed.) Generic and Indexed Programming. LNCS, vol. 7470, pp. 1–46. Springer, Heidelberg (2012)

Hofmann, M., Pierce, B.C., Wagner, D.: Symmetric lenses. In: POPL, pp. 371–384. ACM (2011)

Johnson, M., Rosebrugh, R.: Spans of lenses. In: Terwilliger and Hidaka (2014)

Jones, M.P., Duponcheel, L.: Composing monads. Technical report RR-1004, DCS, Yale (1993)

King, D.J., Wadler, P.: Combining monads. In: Proceedings of the 1992 Glasgow Workshop on Functional Programming, pp. 134–143 (1992)

Liang, S., Hudak, P., Jones, M.P.: Monad transformers and modular interpreters. In: POPL, pp. 333–343 (1995)

Pacheco, H., Hu, Z., Fischer, S.: Monadic combinators for "putback" style bidirectional programming. In: PEPM, pp. 39–50. ACM (2014). http://doi.acm.org/10.1145/2543728.2543737

Plotkin, G., Power, J.: Notions of computation determine monads. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 342–356. Springer, Heidelberg (2002)

Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. SoSyM **9**(1), 7–20 (2010)

Terwilliger, J., Hidaka, S. (eds.): BX Workshop (2014). http://ceur-ws.org/Vol-1133/#bx

TLCBX Project: a theory of least change for bidirectional transformations (2013–2016). http://www.cs.ox.ac.uk/projects/tlcbx/, http://groups.inf.ed.ac.uk/bx/

Wadler, P.: Comprehending monads. Math. Struct. Comput. Sci. **2**(4), 461–493 (1992). http://dx.org/10.1017/S0960129500001560

Wadler, P.: Monads for functional programming. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 24–52. Springer, Heidelberg (1995)