Reflections on monadic lenses

Faris Abou-Saleh\textsuperscript{1}, James Cheney\textsuperscript{2}, Jeremy Gibbons\textsuperscript{1}, James McKinna\textsuperscript{2}, and Perdita Stevens\textsuperscript{2}

\textsuperscript{1} University of Oxford, firstname.lastname@cs.ox.ac.uk
\textsuperscript{2} University of Edinburgh, firstname.lastname@ed.ac.uk

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

1 Introduction

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

Over the past decade, a number of promising proposals for easing programming such \textit{bidirectional transformations} have emerged, including \textit{lenses} \cite{DBLP:journals/corr/abs-1803-02880}, bx based on consistency relations \cite{DBLP:journals/corr/abs-1709-08471}, \textit{symmetric lenses} \cite{DBLP:journals/corr/abs-1803-02880}, and a number of variants and extensions (e.g. \cite{DBLP:journals/corr/abs-1803-02880, DBLP:journals/corr/abs-1803-02880}). Most of these proposals consist of an interface with pure functions and some equational laws that characterize good behavior; the interaction of bidirectionality with other effects has received comparatively little attention.

Some programmers and researchers have already proposed ways to combine elenses and monadic effects \cite{DBLP:journals/corr/abs-1803-02880, DBLP:journals/corr/abs-1803-02880}. Recently, we have proposed symmetric notions of bidirectional computation based on \textit{entangled state monads} \cite{DBLP:journals/corr/abs-1803-02880, DBLP:journals/corr/abs-1803-02880} and \textit{coalgebras} \cite{DBLP:journals/corr/abs-1803-02880}. As a result, there are now several alternative proposals for bidirectional transformations with effects, and the relationships among them are not well understood.
In this paper we summarize and compare the existing proposals, offer some new alternatives, and attempt to provide general and useful definitions of “monadic lenses” and “symmetric monadic lenses”. Perhaps surprisingly, it appears challenging even to define the composition of lenses in the presence of effects, especially in the symmetric case. We first review the definition of pure lenses and two prior proposals for extending them with monadic effects. These definitions have some limitations, and we propose a new definition of monadic lens that overcomes them.

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

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

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

## 2 Asymmetric monadic lenses

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

```
data α ⇝ β = Lens { get :: α → β, put :: α → β → α }
```

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

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

$$(\cdot) :: (\alpha \rightsquigarrow \beta) \rightarrow (\beta \rightsquigarrow \gamma) \rightarrow (\alpha \rightsquigarrow \gamma)$$

$l_1 \cdot l_2 = \text{Lens } (l_2 \cdot \text{get} \circ l_1 \cdot \text{get}) \ (\lambda a c \rightarrow l_1 \cdot \text{put} a \ (l_2 \cdot \text{put} \ (l_1 \cdot \text{get} \ a) \ c))$

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

$$\text{data } \alpha \rightarrow \beta = \text{Lens } \{ \text{get} :: \alpha \rightarrow \beta, \text{put} :: \alpha \rightarrow \beta, \text{create} :: \beta \rightarrow \alpha \}$$

satisfying an additional law:

$$(\text{CreateGet}) \ \ \ \ \text{get} \ (\text{create} \ b) = b$$

2.1 A naive approach

As a first attempt, consider simply adding a monadic effect $\mu$ to the result types of get and put.

$$\text{data } [\alpha \rightsquigarrow_0 \beta]_\mu = \text{MLens}_0 \ \{ mget :: \alpha \rightarrow \mu \beta, mput :: \alpha \rightarrow \beta \rightarrow \mu \alpha \}$$

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

$$(\text{MGetPut}_0) \ \ \ do \ {b \leftarrow mget \ a; mput \ a \ b} = \text{return } a$$

$$(\text{MPutGet}_0) \ \ \ do \ {a' \leftarrow mput \ a \ b; mget \ a'} = \text{return } b$$

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

$$(\cdot) :: [\alpha \rightsquigarrow_0 \beta]_\mu \rightarrow [\beta \rightsquigarrow_0 \gamma]_\mu \rightarrow [\alpha \rightsquigarrow_0 \gamma]_\mu$$

$l_1 \cdot l_2 = \text{MLens}_0 \ (\lambda a \rightarrow \text{do} \ {b \leftarrow l_1 \cdot mget \ a; l_2 \cdot mget \ b})$

$$(\lambda a c \rightarrow \text{do} \ {b \leftarrow l_1 \cdot mget \ a; b' \leftarrow l_2 \cdot mput \ b \ c; l_1 \cdot mput \ a \ b'})$$

This proposal has at least two apparent problems. First, the $(\text{MGetPut}_0)$ law appears to sharply constrain $mget$: indeed, if $mget \ a$ has an irreversible side-effect then $(\text{MGetPut})$ cannot hold. This suggests that $mget$ must either be pure, or have side-effects that are reversible by $mput$, ruling out behaviors such as performing I/O during $mget$. Second, it appears difficult to compose these structures in a way that preserves the laws, unless we again make fairly draconian assumptions about $\mu$. In order to show $(\text{MGetPut}_0)$ for the composition $l_1 \cdot l_2$, it seems necessary to be able to commute $l_2 \cdot mget$ with $l_1 \cdot mget$ and we also need to know that doing $l_1 \cdot mget$ twice is the same as doing it just once. Likewise, to show $(\text{MPutGet}_0)$ we need to commute $l_2 \cdot mget$ with $l_1 \cdot mput$. 

Reflections on monadic lenses 3


2.2 Monadic put-lenses

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

\[
\text{data } [\alpha \rightarrow_1 \beta]_\mu = \text{MLens}_1 \{ \text{mget} :: \alpha \rightarrow \beta, \text{mput} :: \alpha \rightarrow \beta \rightarrow \mu \alpha \}
\]

(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 \( \mu \) has a *monad membership* operation

\[
(\in) :: \alpha \rightarrow \mu \alpha \rightarrow \text{Bool}
\]

satisfying the following two laws:

\[
\begin{align*}
(\in{-}\text{-ID}) & \quad x \in \text{return } x \iff \text{True} \\
(\in{-}\text{->>}) & \quad y \in (m \gg f) \iff \exists x . x \in m \land y \in (f x)
\end{align*}
\]

Then the laws for \( \text{MLens}_1 \) (taken from [12] Prop. 3, p49) are as follows:

\[
\begin{align*}
(\text{MGetPut}_1) & \quad v = \text{mget } s \implies \text{mput } s v = \text{return } s \\
(\text{MPutGet}_1) & \quad s' \in \text{mput } s v' \implies v' = \text{mget } s'
\end{align*}
\]

(In the first law we correct an apparent typo in the original paper, as well as removing the `Just` constructors from both laws.) By making `mget` pure, this definition avoids the immediate problems with composition discussed above, and Pacheco et al. outline a proof that their laws are preserved by composition. However, it is not obvious how to generalize their approach beyond monads that admit a sensible \( \in \) operation. Many interesting monads do have a sensible \( \in \) operation (e.g. `Maybe`, `[]`). Pacheco et al. suggest that \( \in \) can be defined for any monad as \( x \in m \equiv (\exists h : h m = x) \), where `h` is what they call a *(polymorphic) algebra* for `m`, that is, a polymorphic function `h :: m a \rightarrow a`. However, this definition doesn’t appear satisfactory for monads such as `IO`, for which there is no such (pure) function: the \( (\in{-}\text{-ID}) \) 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 \( m :: IO s \) could ultimately return any `a`-value, it seems safe (if perhaps overly conservative) to define \( x \in m = \text{True} \) 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 :: IO s` to find out what its eventual return value is. But then the `(MPutGet1)` law, whose precondition is always true, forces the view space to be trivial. These complications suggest, at least, that it would be advantageous to find a definition of monadic lenses that makes sense (and is preserved under composition) for any monad.
2.3 Monadic lenses

We propose the following definition of monadic lens:

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

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

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

\[
\begin{align*}
\text{(MGetPut)} & \quad \text{do } \{ \text{l.mput s (l.mget s)} \} = \text{return s} \\
\text{(MPutGet)} & \quad \text{do } \{ s' \leftarrow \text{l.mput s v}; k s' \} = \text{do } \{ s' \leftarrow \text{l.mput s v}; k s' v \}
\end{align*}
\]

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

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

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

\[
\begin{align*}
\text{lens2mlens} & :: \text{Monad } \mu \Rightarrow [\alpha \rightsquigarrow \beta] \to [\alpha \rightsquigarrow \beta]_\mu \\
\text{lens2mlens } l & = \text{MLens } (l.\text{get}) (\lambda a b \to \text{return } (l.\text{put a b}))
\end{align*}
\]

\[\blacklozenge\]

**Lemma 2.3.** If \( l :: \text{Lens } \alpha \beta \) is well-behaved, then so is \( \text{lens2mlens } l \).

\[\blacklozenge\]

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

\[
\begin{align*}
\text{idLens} & :: \alpha \rightsquigarrow \alpha \\
\text{idLens} & = \text{Lens } (\lambda a \to a) (\lambda a \to a) \\
\text{fstLens} & :: (\alpha, \beta) \rightsquigarrow \alpha \\
\text{fstLens} & = \text{MLens } \text{fst } (\lambda (s_1, s_2) \to (s'_1, s_2)) \\
\text{sndLens} & :: (\alpha, \beta) \rightsquigarrow \beta \\
\text{sndLens} & = \text{Lens } \text{snd } (\lambda (s_1, s_2) \to (s_1, s'_2))
\end{align*}
\]

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

\[\blacklozenge\]

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

\[
\text{constMLens} :: b \rightarrow [a \rightsquigarrow b]_{\text{Maybe}}
\]

\[
\text{constMLens } b = \text{MLens}(\text{const } b) \ (\lambda a b' \rightarrow \text{if } b \approx b' \text{ then } Just \ a \text{ else Nothing})
\]

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

As a more interesting example, consider:

\[
\text{absLens} :: [\text{Int} \rightsquigarrow \text{Int}]_{\text{Maybe}}
\]

\[
\text{absLens} = \text{MLens} \ (\lambda a b \rightarrow \text{if } b < 0 \text{ then Nothing else Just } (\text{if } a < 0 \text{ then } -b \text{ else } b))
\]

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

The following \textit{logging lens} takes a pure lens \(l\) and, whenever the source value \(a\) changes, records the previous \(a\) value.

\[
\text{logLens} :: \forall \alpha \rightarrow \exists \beta \rightarrow [\alpha \rightsquigarrow \beta]_{\text{Writer } \alpha}
\]

\[
\text{logLens } l = \text{MLens}(l \cdot \text{get}) \ (\lambda a b \rightarrow \text{let } a' = l \cdot \text{mput } a b \text{ in do } \{\text{if } a \not\approx a' \text{ then tell } a \text{ else return } (); \text{return } a'\})
\]

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

Next, we consider composition of monadic lenses.

\[
(\cdot) :: \text{Monad } \mu \Rightarrow [a \rightsquigarrow b]_{\mu} \rightarrow [b \rightsquigarrow c]_{\mu} \rightarrow [a \rightsquigarrow c]_{\mu}
\]

\[
l_1 \cdot l_2 = \text{MLens}(l_2 \cdot \text{mget} \cdot l_1 \cdot \text{mget}) \text{ mput where}
\]

\[
\text{mput } a c = \text{do } \{b \leftarrow l_2 \cdot \text{mput } (l_1 \cdot \text{get a}) c; l_1 \cdot \text{mput a b}\}
\]

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

\textbf{Theorem 2.5.} If \(l_1 :: [A \rightsquigarrow B]_{\text{M}}\) and \(l_2 :: [B \rightsquigarrow C]_{\text{M}}\) are well-behaved, then so is \(l_1 \cdot l_2\).

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

We propose the following alternative:

\[
data [\alpha \rightsquigarrow \beta]_{\mu} = \text{MLens} \{\text{mget} :: \alpha \rightarrow \beta, \text{mput} :: \alpha \rightarrow \beta \rightarrow \mu \ \alpha, \text{mcreate} :: \beta \rightarrow \mu \ \alpha\}
\]
and we consider such initializable monadic lenses to be well-behaved when they satisfy the following additional law:

\[(\text{MCreateGet}) \text{ do } \{ a \leftarrow \text{mcreate } b; k a (\text{mget } a) \} = \text{do } \{ a \leftarrow \text{mcreate } b; k a b \}\]

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

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

When the distinction is important, we use the term \textit{full} for well-behaved lenses equipped with a \textit{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 \text{get} operation of a full lens is surjective.

### 3 Symmetric monadic lenses and spans

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

\[
\text{data } \alpha \rightarrow^r \beta = \text{SLens } \{ \text{put}_R :: (\alpha, \gamma) \rightarrow (\beta, \gamma), \text{put}_L :: (\beta, \gamma) \rightarrow (\alpha, \gamma), \text{missing} :: \gamma \}
\]

Informally, \text{put}_R turns an \(\alpha\) into a \(\beta\), modifying a complement \(\gamma\) as it goes, and symmetrically for \text{put}_L; and \text{missing} is an initial complement, to get the ball rolling. Well-behavedness for symmetric lenses amounts to the following equational laws:

\[
\begin{align*}
\text{(PutRL)} & \quad \text{let } (b, c') = \text{sl.put}_R (a, c) \text{ in } \text{sl.put}_L (b, c') \\
& = \text{let } (b, c') = \text{sl.put}_R (a, c) \text{ in } (a, c') \\
\text{(PutLR)} & \quad \text{let } (a, c') = \text{sl.put}_L (b, c) \text{ in } \text{sl.put}_R (a, c') \\
& = \text{let } (a, c') = \text{sl.put}_L (b, c) \text{ in } (b, c')
\end{align*}
\]

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

\[
\begin{align*}
(\cdot) :: (\alpha \rightarrow^r_1 \beta) \rightarrow (\beta \rightarrow^r_2 \gamma) \rightarrow (\alpha \rightarrow^r_1 \gamma) \\
\text{put}_R (a, (s_1, s_2)) & = \text{let } (b, s'_1) = \text{put}_R (a, s_1) \\
& \quad (c, s'_2) = \text{put}_R (b, s_2) \\
& \quad \text{in } (c, (s'_1, s'_2)) \\
\text{put}_L (c, (s_1, s_2)) & = \text{let } (b, s'_2) = \text{put}_L (c, s_2) \\
& \quad (a, s'_1) = \text{put}_L (b, s_1) \\
& \quad \text{in } (a, (s'_1, s'_2))
\end{align*}
\]

We can define an \textit{identity} symmetric lens as follows:
\[ idSLens :: X \xrightarrow{id} X \]
\[ idSLens = SLens \ id \ id \]

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

**Definition 3.1.** Suppose \( r \subseteq C_1 \times C_2 \). Then \( f \sim_r g \) means that for all \( c_1, c_2, x \), if \( (c_1, c_2) \in r \) and \( (y, c'_1) = f(x, c_1) \) and \( (y', c'_2) = g(y, c_2) \), then \( y = y' \) and \( (c'_1, c'_2) \in r \).

**Definition 3.2 (Symmetric lens equivalence).** Two symmetric lenses \( sl_1 :: SLens C_1 X Y \) and \( sl_2 :: SLens C_2 X Y \) are considered equivalent \( (sl_1 \equiv_{sl} sl_2) \) if there is a relation \( r \subseteq C_1 \times C_2 \) such that

1. \( (sl_1, \text{missing}, sl_2, \text{missing}) \in r \),
2. \( sl_1 \text{put}_R \sim_r sl_2 \text{put}_R \), and
3. \( sl_1 \text{put}_L \sim_r sl_2 \text{put}_L \).

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

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

\[
\begin{align*}
\text{(Identity)} & \quad sl; idSLens \equiv_{sl} sl \equiv_{sl} idSLens ; sl \\
\text{(Assoc)} & \quad sl_1 ; (sl_2 ; sl_3) \equiv_{sl} (sl_1 ; sl_2) ; sl_3 \\
\text{(Cong)} & \quad sl_1 \equiv_{sl} sl'_1 \land sl_2 \equiv_{sl} sl'_2 \implies sl_1 ; sl_2 \equiv_{sl} sl'_1 ; sl'_2
\end{align*}
\]


### 3.1 Naive monadic symmetric lenses

We now consider an obvious monadic generalisation of symmetric lenses, in which the \( \text{put}_L \) and \( \text{put}_R \) functions are allowed to have effects in some monad \( M \):

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

\[
\begin{align*}
data \ [\alpha \xrightarrow{\gamma} \beta]_M = SMLens \{ m\text{put}_R :: (\alpha, \gamma) \rightarrow \mu (\beta, \gamma), \\
m\text{put}_L :: (\beta, \gamma) \rightarrow \mu (\alpha, \gamma), \\
\text{missing} :: \gamma \}\end{align*}
\]

Such a lens \( sl \) is called well-behaved if:

\[
\begin{align*}
\text{(PutRLM)} & \quad \text{do} \ \{(b, c') \leftarrow sl \text{put}_R (a, c) ; sl \text{put}_L (b, c')\} \\
& \quad = \text{do} \ \{(b, c') \leftarrow sl \text{put}_R (a, c) ; \text{return} (a, c')\} \\
\text{(PutLRM)} & \quad \text{do} \ \{(a, c') \leftarrow sl \text{put}_L (b, c) ; sl \text{put}_R (a, c')\} \\
& \quad = \text{do} \ \{(a, c') \leftarrow sl \text{put}_L (b, c) ; \text{return} (b, c')\}
\end{align*}
\]
The above monadic generalisation of symmetric lenses appears natural, but it turns out to have some idiosyncrasies, similar to those of the naive version of monadic lenses we considered in Section 2.1.

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

\[(\cdot) :: \text{Monad } \mu \Rightarrow [\alpha \xleftarrow{\sigma_1} \beta]_{\mu} \rightarrow [\beta \xrightarrow{\sigma_2} \gamma]_{\mu} \rightarrow [\alpha \xrightarrow{\sigma_1, \sigma_2} \gamma]_{\mu}\]

\[sl_1 : sl_2 = SMLens \text{ put}_R \text{ put}_L \text{ missing} \text{ where}\]

\[\text{put}_R (a, (s_1, s_2)) = \text{do} \{(b, s'_1) \leftarrow sl_1.\text{mput}_R (a, s_1);\]

\[(c, s'_2) \leftarrow sl_2.\text{mput}_R (b, s_2);\]

\[\text{return} (c, (s'_1, s'_2))\}\]

\[\text{put}_L (c, (s_1, s_2)) = \text{do} \{(b, s'_2) \leftarrow sl_2.\text{mput}_L (c, s_2);\]

\[(a, s'_1) \leftarrow sl_1.\text{mput}_L (b, s_1);\]

\[\text{return} (a, (s'_1, s'_2))\}\]

\[\text{missing} = (sl_1.\text{missing}, sl_2.\text{missing})\]

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:

\[\text{setBool} :: \text{Bool} \rightarrow [(\cdot) \leftrightarrow (\cdot)]_{\text{State Bool}}\]

\[\text{setBool} b = SMLens \text{ m m} () \text{ where m - = do} \{\text{set b; return} ((), ())\}\]

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

\[\text{Proposition 3.6.} \text{ setBool} b \text{ is well-behaved for } b \in \{\text{True, False}\}, \text{ but setBool True; setBool False is not well-behaved.}\]

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

### 3.2 Entangled state monads

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

\[\text{data} \{\alpha \cong \beta\}_\mu = \text{SetBX} \{\text{get}_L :: \mu \alpha, \text{set}_L :: \alpha \rightarrow \mu (),\]

\[\text{get}_R :: \mu \beta, \text{set}_R :: \beta \rightarrow \mu ()\}\]
subject to a subset of the State monad laws [13], such as:

\[
\begin{align*}
\text{Get}_L \text{Set}_L & \quad \text{do} \{ a \leftarrow \text{get}_L; \text{set}_L a \} = \text{return } () \\
\text{Set}_L \text{Get}_L & \quad \text{do} \{ \text{set}_L a; \text{get}_L \} = \text{do} \{ \text{set}_L a; \text{return } a \}
\end{align*}
\]

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 \(\text{set}_L\) and \(\text{set}_R\) operations do not commute.

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

### 3.3 Spans of monadic lenses

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

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

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

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):

\[
\begin{align*}
(\triangleleft) & \quad \text{Monad } \mu \Rightarrow [\alpha_1 \rightsquigarrow \sigma \rightsquigarrow \beta_1]_\mu \rightarrow [\alpha_2 \rightsquigarrow \sigma \rightsquigarrow \beta_2]_\mu \\
ml \triangleleft ml & = \text{Span} (\text{sp}.\text{left}; ml) (\text{sp}.\text{right}) \\
(\triangleright) & \quad \text{Monad } \mu \Rightarrow [\alpha \rightsquigarrow \sigma \rightsquigarrow \beta_1]_\mu \rightarrow [\beta_1 \rightsquigarrow \beta_2]_\mu \rightarrow [\alpha \rightsquigarrow \sigma \rightsquigarrow \beta_2]_\mu \\
sp \triangleright ml & = \text{Span sp}.\text{left} (\text{sp}.\text{right}; ml)
\end{align*}
\]

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

Given a span of monadic lenses

\[
\text{sp} :: [A \rightsquigarrow S \rightsquigarrow B]_M
\]
we can construct a symmetric lens

\[
\text{sl} :: [A \leftrightarrow S \leftrightarrow B]_M
\]
as follows:

\[
\text{span2smllens} \left( \text{left}, \text{right} \right) = \text{SMLens mput}_R \text{ mput}_L \text{ Nothing}
\]

where

\[
\text{mput}_R \left( a, \text{Just} \ s \right) = \text{do} \{ s' \leftarrow \text{left.mput} \ a; \text{return} \left( \text{right.mget} \ s', \text{Just} \ s' \right) \}
\]

\[
\text{mput}_R \left( a, \text{Nothing} \right) = \text{do} \{ s' \leftarrow \text{left.mcreate} \ a; \text{return} \left( \text{right.mget} \ s', \text{Just} \ s' \right) \}
\]

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

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

**Theorem 3.10.** If \( \text{sp} :: [A \leftarrow S \leftarrow B]_M \) is well-behaved, then \( \text{span2smllens} \ \text{sp} \) is also well-behaved.

Given \( \text{sl} :: [A \leftrightarrow C \rightarrow B]_M \), let \( S \subseteq A \times B \times C \) be the set of consistent triples \((a, b, c)\), that is, those for which \( \text{sl.mput}_R \ (a, c) = \text{return} \ (b, c) \) and \( \text{sl.mput}_L \ (b, c) = \text{return} \ (a, c) \). We construct \( \text{sp} :: [A \leftarrow S \leftarrow B]_M \) by...
\[ \text{smlens2span } s \text{=} \text{Span (MLens get}_L \text{put}_L \text{create}_L) \text{(MLens get}_R \text{put}_R \text{create}_R) \text{ where} \]

\[ \text{get}_L (a,b,c) = a \]

\[ \text{put}_L (a,b,c) = \begin{cases} (b',c') \leftarrow \text{sl.mput}_R (a',c); \text{return} \ (a',b',c') \end{cases} \]

\[ \text{create}_L a = \begin{cases} \text{return} \ (a,b',c') \end{cases} \]

\[ \text{get}_R (a,b,c) = b \]

\[ \text{put}_R (a,b,c) = \begin{cases} (a',c') \leftarrow \text{sl.mput}_L (b',c); \text{return} \ (a',b',c') \end{cases} \]

\[ \text{create}_R b = \begin{cases} \text{return} \ (a,b',c) \end{cases} \]

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

\textbf{Example 3.11.} Consider the following monadic symmetric lens construction:

\[ \text{fail} :: () \leftrightarrow () \]

\[ \text{fail} = \text{SMLens Nothing Nothing} () \]

This is well-behaved but \text{smlens2span} \text{fail} is not; each leg of the induced span is of the following form:

\[ \text{failMLens} :: \text{MLens Maybe} () () \]

\[ \text{failMLens} = \text{MLens id} (\lambda() () \rightarrow \text{Nothing}) (\lambda() \rightarrow \text{Nothing}) \]

which cannot satisfy (MGetPut). \hfill ◊

For pure symmetric lenses, \text{smlens2span} does preserve well-behavedness.

\textbf{Theorem 3.12.} If \( s :: \text{SMLens Id C A B} \) is well-behaved, then \text{smlens2span} \( s \) is also well-behaved, with state space \( S \) consisting of the consistent triples of \( s \). \hfill ◊

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

\section{4 Equivalence of spans}

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

\textbf{Definition 4.1 (Isomorphism Equivalence).} Two \( M \)-lens spans \( sp_1 :: [A \rightsquigarrow S_1 \rightsquigarrow B]_M \) and \( sp_2 :: [A \rightsquigarrow S_2 \rightsquigarrow B]_M \) are isomorphic \( (sp_1 \equiv sp_2) \) if there is an isomorphism \( h :: S_1 \rightarrow S_2 \) on their state spaces such that \( h; sp_2.\text{left} = sp_1.\text{left} \) and \( h; sp_2.\text{right} = sp_1.\text{right} \). \hfill ◊
Note that any isomorphism $h :: S_1 \rightarrow S_2$ can be made into a (monadic) lens; we omit an explicit conversion.

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

**Definition 4.2 (Span Equivalence).** Two $M$-lenses spans $sp_1 :: [A \rightsquigarrow S_1 \rightsquigarrow B]_M$ and $sp_2 :: [A \rightsquigarrow S_2 \rightsquigarrow B]_M$ are related by $\rightsquigarrow$ if there is a full lens $h :: S_1 \rightsquigarrow S_2$ such that $h(sp_1.left) = sp_2.left$ and $h(sp_2.right) = sp_1.right$. The equivalence relation $\equiv_s$ is the least equivalence relation containing $\rightsquigarrow$.

One important consideration (emphasized by Johnson and Rosebrugh) is the need to avoid making all compatible spans equivalent to the “trivial” span $[A \rightsquigarrow \emptyset \rightsquigarrow B]_M$. 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 $h\cdot c = id$. We chose instead to require $h$ to be a full lens. This is actually slightly stronger than Johnson and Rosebrugh’s definition (at least from a constructive perspective), because $h$ is equipped with a specific choice of $c = create$ satisfying $h\cdot c = id$ (the (CreateGet) law).

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

**Theorem 4.3.** Given $sp_1 :: [A \rightsquigarrow S_1 \rightsquigarrow B]_M$ and $sp_2 :: [A \rightsquigarrow S_2 \rightsquigarrow B]_M$, if $sp_1 \equiv_s sp_2$ then there exists $sp :: S_1 \rightsquigarrow S_2$ such that $sp.left = sp_1.left = sp_2.left$ and $sp.right = sp_1.right = sp_2.right$.

Thus, span equivalence is a doubly appropriate name for $\equiv_s$: it is an equivalence of spans witnessed by a (pure) span.

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

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

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

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

**Definition 4.5 (Bisimulation equivalence).** A bisimulation of $M$-lenses spans $sp_1 :: [A \rightsquigarrow S_1 \rightsquigarrow B]_M$ and $sp_2 :: [A \rightsquigarrow S_2 \rightsquigarrow B]_M$ is a $M$-lens span $sp :: [A \rightsquigarrow R \rightsquigarrow B]_M$ 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 to $sp_2$. We write $sp_1 \equiv_b sp_2$ when there is a bisimulation of spans $sp_1$ and $sp_2$.

**Proposition 4.6.** Each of the relations $\equiv_i$, $\equiv_s$ and $\equiv_b$ are equivalence relations on compatible spans of $M$-lenses and satisfy (Identity), (Assoc) and (Cong).

**Theorem 4.7.** $sp_1 \equiv_i sp_2$ implies $sp_1 \equiv_s sp_2$, but not the converse.
Proof. The forward direction is obvious; for the reverse direction, consider

\[
\begin{align*}
    h &:: \mathrm{Bool} \leadsto () \\
    h &:: \lambda x (\lambda y x) \to y \to x \\
    sp_1 &:: () \leadsto () \\
    sp_1 &:: \mu \mathrm{idMLens} \mu \mathrm{idMLens} \\
    sp_2 &:: (h; sp_1.\left, h; sp_2.\right)
\end{align*}
\]

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

Theorem 4.8. Given \( sp_1 :: [A \leadsto S_1 \leadsto B]_M, sp_2 :: [A \leadsto S_2 \leadsto B]_M, \) if \( sp_1 \equiv sp_2 \) then \( sp_1 \equiv b sp_2 \).

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

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

Theorem 4.9. Given \( sp_1 :: A \leadsto S_1 \leadsto B, sp_2 :: A \leadsto S_2 \leadsto B, \) if \( sp_1 \equiv b sp_2 \) then \( sp_1 \equiv s sp_2 \).

Proof. Given \( R \) and a span \( sp :: A \leadsto R \leadsto B \) constituting a bisimulation \( sp_1 \equiv b sp_2 \), it suffices to construct a span \( sp' = (l, r) :: S_1 \leadsto S_2 \leadsto S \) satisfying \( l ; sp_1.\left = r ; sp_2.\left \) and \( l ; sp_1.\right = r ; sp_2.\right \). \(\square\)

We leave it as an open question to determine whether \( \equiv b \) is equivalent to \( \equiv s \) for spans of monadic lenses, or whether an analogous result to Theorem 4.9 carries over to symmetric lenses.

5 Conclusions

Lenses are a popular and powerful abstraction for bidirectional transformations. Although they are most often studied in their conventional, pure form, practical applications of lenses typically grapple with side-effects, including exceptions, state, and user interaction. Some recent proposals for extending lenses with monadic effects have been made and our proposal for (asymmetric) monadic lenses improves on them because it is closed under composition for arbitrary monads. Furthermore, we investigated the symmetric case, and showed that spans of monadic lenses are closed under composition, while the obvious generalization of pure symmetric lenses to incorporate monadic effects is not closed under composition. Finally, we presented three notions of equivalence for spans of monadic lenses, related them, and proved a new (and perhaps surprising) result.
Although some of these ideas are present in recent papers [8, 9, 3, 1, 2], this paper reflects our desire to clarify these ideas and expose them in their clearest form — a desire that is strongly influenced by Wadler’s work on a wide variety of related topics [16, 11, 17], and by our interactions with him as a colleague. We hope that this presentation will not just be an interesting curiosity but will also help inspire further research and appreciation of bidirectional programming with effects.

References

A Proofs for Section 2

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

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

\[
\text{do \{ } l_1.mput a (l_1.mget a) \text{ \}}
\]

\[
\text{= } \begin{array}{c}
\text{do \{ } b \leftarrow l_2.mput (l_1.mget a) (l_2.mget (l_1.mget a)); l_1.mput a b \text{ \}} \\
\text{do \{ } b \leftarrow \text{return } (l_1.mget a); l_1.mput a b \text{ \}} \\
\text{= } (\text{MGetPut}) \\
\text{do \{ } l_1.mput a (l_1.mget a) \text{ \}} \\
\text{= } (\text{MGetPut}) \\
\text{\}} \text{ return } a
\]

For (MPutGet), the proof is as follows:

\[
\text{do \{ } a’ \leftarrow l_1.mput a c; \text{return } (a’, l_1.mget a’) \text{ \}}
\]

\[
\text{= } \begin{array}{c}
\text{do \{ } b \leftarrow l_2.mput (l_1.mget a) c; \\
\text{~ } a’ \leftarrow l_1.mput a b; \\
\text{~ } \text{return } (a’, l_2.mget (l_1.mget a’)) \text{ \}} \\
\text{\} \text{\} \text{ (MPutGet)} \\
\text{do \{ } b \leftarrow l_2.mput (l_1.mget a) c; \\
\text{~ } a’ \leftarrow l_1.mput a b; \\
\text{~ } \text{return } (a’, l_2.mget b) \text{ \}} \\
\text{\} \text{\} \text{ (MPutGet)} \\
\text{\text{do \{ } b \leftarrow l_2.mput (l_1.mget a) c; \\
\text{~ } a’ \leftarrow l_1.mput a b; \\
\text{~ } \text{return } (a’, c) \text{ \}} \\
\text{\} \text{\} \text{ definition} \\
\text{do \{ } a’ \leftarrow l_1.mput a c; \text{return } (a’, c) \text{ \}}
\]

B Proofs for Section 3

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

Proof. For the first part:

Let $s = \text{setBool } x$. We consider (PutRLM), and (PutLRM) is symmetric.

\[
\text{do \{ } (b, c’) \leftarrow (\text{setBool } x).\text{mput}_R ((\_); (\_)); (\text{setBool } x).\text{mput}_L (b, c’) \text{ \}}
\]

\[
\text{= } \begin{array}{c}
\text{\} \text{\} \text{ Definition}}
\]

\]
Reflections on monadic lenses

We proceed as follows:

\[
\begin{align*}
\text{do } \{ (b, c') &\leftarrow \text{do } \{ \text{set } x \text{; return } (((), ())) \}; \text{set } x \text{; return } (((), c')) \} \\
&= \quad \text{do } \{ \text{set } x ; (b, c') \leftarrow \text{return } (((), ())) ; \text{set } x ; \text{return } (((), c')) \} \\
&= \quad \text{do } \{ \text{set } x ; \text{set } x ; (b, c') \leftarrow \text{return } (((), ())) ; \text{return } (((), c')) \} \\
&= \quad \text{do } \{ \text{sl } \leftarrow \text{do } \{ \text{set } x ; \text{return } (((), ())) ; \text{set } x ; \text{return } (((), c')) \} \\
&= \quad \text{do } \{ \text{set } x ; \text{sl } \leftarrow \text{do } \{ \text{set } x \text{; return } (((), ())) \}; \text{return } (((), c')) \} \\
&= \quad \text{do } \{ (b, c') \leftarrow \text{do } \{ \text{set } x ; \text{return } (((), ())) \}; \text{return } (((), c')) \} \\
&= \quad \text{do } \{ (b, c') \leftarrow \text{do } \{ \text{putRLM } \text{. mput}_R (((), ())); \text{return } (((), c')) \} \\
\end{align*}
\]

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

\[
\begin{align*}
\text{do } \{ (c, s') &\leftarrow \text{do } \{ \text{sl } \leftarrow \text{do } \{ \text{sl } \leftarrow \text{do } \{ \text{sl } \} \}; \text{sl } \leftarrow \text{do } \{ \text{sl } \} \}; \text{return } (((), ())) \} \\
&= \quad \text{do } \{ (c, s') \leftarrow \text{do } \{ \text{sl } \leftarrow \text{do } \{ \text{sl } \} \}; \text{return } (((), ())) \} \\
&= \quad \text{do } \{ (b, s') \leftarrow \text{do } \{ \text{sl } \leftarrow \text{do } \{ \text{sl } \} \}; \text{return } (((), ())) \} \\
&= \quad \text{do } \{ (b, s') \leftarrow \text{do } \{ \text{sl } \leftarrow \text{do } \{ \text{sl } \} \}; \text{return } (((), ())) \} \\
&= \quad \text{do } \{ (b, s') \leftarrow \text{do } \{ \text{sl } \leftarrow \text{do } \{ \text{sl } \} \}; \text{return } (((), ())) \} \\
&= \quad \text{do } \{ (b, s') \leftarrow \text{do } \{ \text{sl } \leftarrow \text{do } \{ \text{sl } \} \}; \text{return } (((), ())) \} \\
\end{align*}
\]
(c, s₂) ← (setBool False).mputR (b, s₂);
(c', (s₁', s₂')) ← return (c, (s₁', s₂'));
return (c', (s₁', s₂'))

= [] Monad unit
  do [(b, s₁') ← (setBool True).mputR (a, s₁);
         (c, s₂) ← (setBool False).mputR (b, s₂);
         return (c, (s₁', s₂'))]

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 ml₁ :: [σ₁ ⊸ β]μ and ml₂ :: [σ₂ ⊸ β]μ are well-behaved then so is ml₁ & ml₂ :: [σ₁ × σ₂] ⊸ [σ₁ × σ₂]μ.

Proof. It suffices to consider the two lenses l₁ = MLens fst putL createL and l₂ = MLens snd putR createR in isolation. Moreover, the two cases are completely symmetric, so we only show the first.

For (MGetPut), we show:

  do { l₁.mput (s₁, s₂) (l₁.mget (s₁, s₂)) }
  = [] definition
  do { putL (s₁, s₂) (fst (s₁, s₂)) }
  = [] definition of putL and fst
  do { s₂ ← right.mput s₂ (left.mget s₁) }
  = [] (s₁, s₂) consistent
  do { s₂ ← right.mput s₂ (right.mget s₂) }
  = [] (MGetPut)
  return s

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

  do { (s₁', s₂) ← l₁.mput (s₁, s₂) v; return ((s₁', s₂), l₁.mget (s₁', s₂)) }
  = [] definition
  do { (s₁', s₂) ← putR (s₁, s₂) v; return ((s₁', s₂), fst (s₁', s₂)) }
  = [] definition
  do { s₂ ← ml₂.mput s₂ (ml₁.mget v); (s₁', s₂) ← return (v, s₂); return ((s₁', s₂), fst (s₁', s₂)) }
  = [] definition of fst
  do { s₂ ← ml₂.mput s₂ (ml₁.mget v); (s₁', s₂) ← return (v, s₂); return ((s₁', s₂), s₁') }
  = [] monad laws
  do { s₂ ← ml₂.mput s₂ (ml₁.mget v); (s₁', s₂) ← return (v, s₂); return ((s₁', s₂), s₁') }
  = [] definition
  do { s₁' ← ml₁.mput (s₁, s₂) v; return ((s₁', s₂), v) }
  = [] definition
  do { (s₁', s₂) ← l₁.mput (s₁, s₂) v; return ((s₁', s₂), v) }

The proof for (MCreateGet) is similar.

Finally, we show that putL :: (σ₁ ⊸ σ₂) → σ₁ → μ (σ₁ × σ₂), and in particular, that it maintains the consistency invariant on the state space σ₁ × σ₂. Assume that (s₁, s₂) ::
\(\sigma_1 \times \sigma_2\) and \(s'_1 :: \sigma_1\) are given. Thus, \(ml_1.mget s_1 = ml_2.mget s_2\). We must show that any value returned by \(put_L\) also satisfies this consistency criterion. By definition,
\[
put_L (s_1, s_2) s'_1 = \text{do} \{ s'_2 \leftarrow ml_2.mput s_2 (ml_1.mget s'_1); \text{return} (s'_1, s'_2) \}
\]
By \((\text{MPutGet})\), any \(s'_2\) resulting from \(ml_2.mput s_2 (ml_1.mget s'_1)\) will satisfy \(ml_2.mget s'_2 = ml_1.mget s'_1\). The proof that \(create_L :: \sigma_1 \rightarrow \mu (\sigma_1 \times \sigma_2)\) is similar, but simpler. \(\square\)

**Theorem 3.10.** If \(sp :: [A \leftarrow S \rightarrow B]_M\) is well-behaved, then \(span2smLens sp\) is also well-behaved. \(\diamondsuit\)

**Proof.** Let \(sl = span2smLens sp\). We need to show that the laws \((\text{PutRLM})\) and \((\text{PutLRM})\) hold. We show \((\text{PutRLM})\), and \((\text{PutLRM})\) is symmetric.

\[\text{[JRC: TODO: Need to update this]}\] We need to show that
\[
\text{do} \{ (b', mc') \leftarrow sl.mputR (a, mc); sl.mputL (b', mc') \}
= \\
\text{do} \{ (b', mc') \leftarrow sl.mputR (a, mc); \text{return} (a, mc') \}
\]

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

If \(mc = \text{Nothing}\) then we reason as follows:

\[
\text{do} \{ (b', mc') \leftarrow sl.mputR (a, \text{Nothing}); sl.mputL (b', mc') \}
= \\
\text{[[ Definition ]]}
\text{do} \{ s' \leftarrow sp.left.mcreate a; (b', mc') \leftarrow (sp.right.mget s', \text{Just} s'); sl.mputL (b', mc') \}
= \\
\text{[[ monad unit ]]}
\text{do} \{ s' \leftarrow sp.left.mcreate a; sl.mputL (sp.right.mget s', \text{Just} s') \}
= \\
\text{[[ definition ]]}
\text{do} \{ s' \leftarrow sp.left.mcreate a; s'' \leftarrow sp.right.mget s' (sp.right.mget s'); \text{return} (sp.left.mget s'', \text{Just} s'') \}
= \\
\text{[[ (\text{MGetPut})] []]}
\text{do} \{ s' \leftarrow sp.left.mcreate a; s'' \leftarrow \text{return} s'; \text{return} (sp.left.mget s'', \text{Just} s'') \}
= \\
\text{[[ monad unit ]]}
\text{do} \{ s' \leftarrow sp.left.mcreate a; \text{return} (sp.left.mget s', \text{Just} s') \}
= \\
\text{[[ labelMCreateGet ] []]}
\text{do} \{ s' \leftarrow sp.left.mcreate a; \text{return} (a, \text{Just} s') \}
= \\
\text{[[ monad unit ] []]}
\text{do} \{ s' \leftarrow sp.left.mcreate a; (b', mc') \leftarrow (sp.right.get s', \text{Just} s'); \text{return} (a, mc') \}
= \\
\text{[[ Definition ] []]}
\text{do} \{ (b', mc') \leftarrow sl.mputR (a, \text{Nothing}); \text{return} (a, mc') \}
\]

If \(mc = \text{Just}\) \(c\) then we reason as follows:

\[
\text{do} \{ (b', mc') \leftarrow sl.mputR (a, \text{Just} s); sl.mputL (b', mc') \}
= \\
\text{[[ Definition ]]}
\text{do} \{ s' \leftarrow sp.left.mput s a; (b', mc') \leftarrow (sp.right.mget s', \text{Just} s'); sl.mputL (b', mc') \}
= \\
\text{[[ monad unit ] []]}
\text{do} \{ s' \leftarrow sp.left.mput s a; sl.mputL (sp.right.mget s', \text{Just} s') \}
\]
Theorem 3.12. If \( sl :: SMLens Id C A B \) is well-behaved, then \( sm\text{lens2span} \, sl \) is also well-behaved, with state space \( S \) consisting of the consistent triples of \( sl \).

Proof. First we show that, given a symmetric lens \( sl \), the operations of \( sp = sm\text{lens2span} \, sl \) preserve consistency of the state. Assume \( (a, b, c) \) is consistent. To show that \( sp.\text{left.mput} \, (a, b, c) \, a' \) is consistent for any \( a' \), we have to show that \( (a', b', c') \) is consistent, where \( a' \) is arbitrary and \( (b', c') = sm\text{lens2span}_R \, (a', c) \). For one half of consistency, we have:

\[
\begin{align*}
\text{sl.mput}_L \, (b', c') &= \text{sl.mput}_R \, (a', c) = (b', c'), \text{ and (PutRLM)} \\
(a', c')
\end{align*}
\]

and then for the other half:

\[
\begin{align*}
\text{sl.mput}_R \, (a', c') &= \text{above, and (PutLRM)} \\
(b', c')
\end{align*}
\]

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

We will now show that \( sm\text{lens2span} \, sl \) is a well-behaved span for any symmetric lens \( sl \). For \( (MGetPut) \), we proceed as follows:

\[
\begin{align*}
\text{sl.left.mput} \, (a, b, c) \, (\text{sl.left.mget} \, (a, b, c)) &= \text{Definition} \\
\text{do} \{ b', c' \leftarrow \text{sl.mput}_R \, (a, c); \text{return} \, (a, b', c') \} &= \text{Consistency of \( (a, b, c) \)} \\
\text{do} \{ b', c' \leftarrow \text{return} \, (b, c); \text{return} \, (a, b', c') \} &= \text{Monad unit} \\
\text{return} \, (a, b, c)
\end{align*}
\]

For \( (MPutGet) \), we have:

\[
\begin{align*}
\text{do} \{ s' \leftarrow \text{sl.left.put} \, (a, b, c); \text{return} \, (\text{sl.left.mget} \, s', s') \} &= \text{Definition}
\end{align*}
\]
C Proofs for Section 4

The proof for (MCreateGet) is similar. For (MPutGet), we have:

\[
\begin{align*}
\textbf{do} \{ & (b', c') \leftarrow \text{sl.mputR} \ (d', c); s' \leftarrow \text{return} \ (a', b', c'); \text{return} \ (\text{sl.left.mget} \ s', s') \} \\
& = \left[ \begin{array}{l}
\text{monad unit} \\
\text{Definition}
\end{array} \right] \\
\textbf{do} \{ & (b', c') \leftarrow \text{sl.mputR} \ (d', c); \text{return} \ (a', (d', b', c')) \\
& = \left[ \begin{array}{l}
\text{Definition} \\
\text{monad unit}
\end{array} \right] \\
\textbf{do} \{ & (b', c') \leftarrow \text{sl.mputR} \ (d', c); s' \leftarrow \text{return} \ (a', b', c'); \text{return} \ (a', s') \\
& = \left[ \begin{array}{l}
\text{Definition} \\
\text{Definition}
\end{array} \right] \\
\textbf{do} \{ & s' \leftarrow \text{sl.left.put} \ (a, b, c) \ a'; \text{return} \ (a', s')
\end{align*}
\]

\[ \square \]

C Proofs for Section 4

Lemma C.1. Suppose \( l_1 : A \rightarrow B \) and \( l_2 : C \rightarrow B \) are pure lenses. Then \((l_1 \otimes l_2).left : l_1 = (l_1 \otimes l_2).right : l_2 \).

Proof. We show that each component of \((l_1 \otimes l_2).left : l_1\) equals the corresponding component of \((l_1 \otimes l_2).right : l_2\).

For get:

\[
\begin{align*}
(l_1 \otimes l_2).left : l_1 . \text{get} \ (a, c) \\
& = \left[ \begin{array}{l}
\text{Definition} \\
l_1 . \text{get} \ (l_1 \otimes l_2).left \ . \text{get} \ (a, c)
\end{array} \right] \\
& = \left[ \begin{array}{l}
\text{Definition} \\
l_1 . \text{get} \ a
\end{array} \right] \\
& = \left[ \begin{array}{l}
\text{Consistency} \\
l_2 . \text{get} \ c
\end{array} \right] \\
& = \left[ \begin{array}{l}
\text{Definition} \\
l_2 . \text{get} \ (l_1 \otimes l_2).right . \text{get} \ (a, c))
\end{array} \right] \\
& = \left[ \begin{array}{l}
\text{Definition}
\end{array} \right] \\
& (l_1 \otimes l_2).right : l_2 . \text{get} \ (a, c)
\end{align*}
\]

For put:
(l₁ ∙ l₂).left;l₁).put (a, c) b
= [itungDefinition][
(l₁ ∙ l₂).left.put (a, c) (l₁.put ((l₁ ∙ l₂).get (a, c)) b)
= [itungDefinition][
(l₁ ∙ l₂).left.put (a, c) (l₁.put a b)
= [itungDefinition][
let d' = l₁.put a b in
let c' = l₂.put c (l₁.get d') in (a', c')
= [itunginline let][
(l₁.put a b,l₂.put c (l₁.get (l₁.put a b)))
= [itung(PutGet)][
(l₁.put a b,l₂.put c b)
= [itungreverse above steps][
(l₁ ∙ l₂).right;l₂).put (a, c) b

Finally, for create:

(l₁ ∙ l₂).left;l₁).create b
= [itungDefinition][
(l₁ ∙ l₂).left.create (l₁.create b)
= [itungDefinition][
let c = l₂.create (l₁.get (l₁.create b)) in (l₁.create b, c)
= [itung(CreateGet)][
let c = l₂.create b in (l₁.create b, c)
= [itungInline let][
(l₁.create b,l₂.create b)
= [itungreverse above steps][
(l₁ ∙ l₂).right;l₂).create b

\)

Theorem 4.3. Given sp₁ :: [A ↘ S₁ ↘ B]ₘ and sp₂ :: [A ↘ S₂ ↘ B]ₘ, if sp₁ ⊑ₚ sp₂ then there exists a pure span sp :: S₁ ↘ S ↘ S₂ such that sp.left ; sp₁.left = sp.right ; sp₂.left and sp.left ; sp₁.right = sp.right ; sp₂.right.

Proof. Let sp₁ and sp₂ be given such that sp₁ ⊑ₚ sp₂. The proof is by induction on the length of the sequence of ⇐ steps linking sp₁ to sp₂.

If sp₁ = sp₂ then the result si immediate. If sp₁ ⇐ sp₂ then we can complete a span between S₁ and S₂ using identity lenses. For the inductive case, suppose that the result holds for sequences of up to n ⇐ steps, and suppose sp₁ ⊑ₚ sp₂' holds bn n ⇐ or ⇐ steps. There are two cases, depending on the direction of the first step. If sp₁ ⇐ sp₂' ⊑ₚ sp₂ then by induction we must have a pure span sp between S₁ and S₂ and sp₁ ⇐ sp₂ holds by virtue of a lens h :: S₁ → S₁, so we can simply compose h with sp.left to obtain the required span between S₁ and S₂. Otherwise, if sp₁ ⇐ sp₂' ⊑ₚ sp₂ then by induction we must have a pure span sp between S₁ and S₂ and we must have a lens h :: S₁ ≥ S₃ to S₃, so we use Lemma C.1 to form a span sp' :: S₁ ↘ S₁ × S₃ ↘ S₃ and extend sp'.right with sp.right to form the required span between S₁ and S₃. \)
Theorem 4.8. Given \( sp_1 :: [A ⇏ S_1 ⇕ B]_M, sp_2 :: [A ⇏ S_2 ⇕ B]_M \), if \( sp_1 \equiv_b sp_2 \) then \( sp_1 \equiv_b sp_2 \).

Proof. We give the details for the case \( sp_1 \rightsquigarrow sp_2 \). 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 \( sp = (l_0, r_0) :: [A ⇏ R ⇕ B]_M \) such that \( \text{fst} \) maps \( R \) to \( S_1 \) and \( \text{snd} \) maps \( sp \) to \( sp_2 \). We take \( R = \{ (s_1, s_2) \mid s_2 = l \cdot \text{get} \ (s_1) \} \) and proceed as follows:

\[
\begin{align*}
l_0 & ::= [R ⇏ A]_M \\
l_0, \text{mget} \ (s_1, s_2) & = l_1, \text{mget} \ s_1 \\
l_0, \text{mput} \ (s_1, s_2) \ a & = \text{do} \ \{ s_1 \leftarrow l_1, \text{mput} \ s_1 \ a; \text{return} \ (s_1, l \cdot \text{get} \ s_1) \} \\
l_0, \text{mcreate} \ a & = \text{do} \ \{ s_1 \leftarrow l_1, \text{mcreate} \ a; \text{return} \ (s_1, l \cdot \text{get} \ s_1) \} \\
r_0 & ::= [R ⇏ B]_M \\
r_0, \text{mget} \ (s_1, s_2) & = r_1, \text{mget} \ s_1 \\
r_0, \text{mput} \ (s_1, s_2) \ b & = \text{do} \ \{ s_1 \leftarrow r_1, \text{mput} \ s_1 \ b; \text{return} \ (s_1, l \cdot \text{get} \ s_1) \} \\
r_0, \text{mcreate} \ b & = \text{do} \ \{ s_1 \leftarrow r_1, \text{mcreate} \ a; \text{return} \ (s_1, l \cdot \text{get} \ s_1) \}
\end{align*}
\]

We must now show that \( l_0 \) and \( r_0 \) are well-behaved (full) lenses, and that the projections \( \text{fst} \) and \( \text{snd} \) 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 \( \text{MGetPut} \) we have:

\[
\begin{align*}
l_0, \text{mput} \ (s_1, s_2) \ (l_0, \text{mget} \ (s_1, s_2)) & = \text{do} \ \{ s'_1 \leftarrow l_1, \text{mput} \ s_1 \ (l_1, \text{mget} \ s_1); \text{return} \ (s'_1, l \cdot \text{get} \ s'_1) \} \\
& = \text{do} \ \{ s'_1 \leftarrow \text{return} \ s_1; \text{return} \ (s'_1, l \cdot \text{get} \ s'_1) \} \\
& = \text{return} \ (s_1, l \cdot \text{get} \ s_1) \\
& = \text{return} \ (s_1, s_2)
\end{align*}
\]

For \( \text{MPutGet} \) we have:

\[
\begin{align*}
\text{do} \ \{ (s''_1, s''_2) \leftarrow l_0, \text{mput} \ (s_1, s_2) \ a; \text{return} \ ((s''_1, s''_2), l_0, \text{mget} \ (s''_1, s''_2)) \} & = \text{do} \ \{ s'_1 \leftarrow l_1, \text{mput} \ s_1 \ a; (s''_1, s''_2) \leftarrow \text{return} \ (s'_1, l \cdot \text{get} \ s'_1); \text{return} \ ((s''_1, s''_2), l_1, \text{mget} \ s'_1) \} \\
& = \text{do} \ \{ s'_1 \leftarrow \text{return} \ s_1; \text{return} \ ((s'_1, l \cdot \text{get} \ s'_1), l_1, \text{mget} \ s'_1) \} \\
& = \text{do} \ \{ s'_1 \leftarrow l_1, \text{mput} \ s_1 \ a; \text{return} \ ((s'_1, l \cdot \text{get} \ s'_1), l_1, \text{mget} \ s'_1) \} \\
& = \text{do} \ \{ s'_1 \leftarrow l_1, \text{mput} \ s_1 \ a; (s''_1, s''_2) \leftarrow \text{return} \ (s'_1, l \cdot \text{get} \ s'_1); \text{return} \ ((s''_1, s''_2), a) \} \\
& = \text{do} \ \{ s'_1 \leftarrow l_1, \text{mput} \ s_1 \ a; (s''_1, s''_2) \leftarrow \text{return} \ (s'_1, l \cdot \text{get} \ s'_1); \text{return} \ ((s''_1, s''_2), a) \} \\
& = \text{do} \ \{ (s''_1, s''_2) \leftarrow l_0, \text{mput} \ (s_1, s_2) \ a; \text{return} \ ((s''_1, s''_2), a) \}
\end{align*}
\]

Finally, for \( \text{MCreateGet} \) we have:
mcreate

ify the following three equations that show that

applications of monad laws. To show that

\( \text{fst} \)

\( \text{mget} \)

\( \text{mput} \)

For the \( \text{mput} \)

equation:

\[
\text{do} \left\{ \begin{array}{l}
(s_1, s_2) \leftarrow l_0.\text{mcreate} \ a; \text{return} \ ((s_1, s_2), l_0.\text{mget} \ (s_1, s_2)) \end{array} \right.
\]

\[
= \left[ \begin{array}{l}
\text{Definition} \\
\text{Monad unit} \\
\text{Monad unit} \\
\text{Monad unit} \\
\text{Definition} \\
\text{Definition} \\
\end{array} \right]
\]

\[
\text{do} \left\{ \begin{array}{l}
(s'_1, s'_2) \leftarrow l_1.\text{mcreate} \ a; \ (s_1, s_2) \leftarrow \text{return} \ ((s'_1, l \cdot \text{get} \ s'_1), l_1.\text{mget} \ s_1) \end{array} \right.
\]

\[
= \left[ \begin{array}{l}
\text{Definition} \\
\text{Monad unit} \\
\text{Monad unit} \\
\text{Definition} \\
\end{array} \right]
\]

\[
\text{do} \left\{ \begin{array}{l}
(s'_1, s'_2) \leftarrow l_1.\text{mcreate} \ a; \ (s_1, s_2) \leftarrow \text{return} \ ((s'_1, l \cdot \text{get} \ s'_1), l_1.\text{mget} \ s_1) \end{array} \right.
\]

\[
= \left[ \begin{array}{l}
\text{Definition} \\
\text{Monad unit} \\
\text{Monad unit} \\
\text{Definition} \\
\end{array} \right]
\]

\[
\text{do} \left\{ \begin{array}{l}
(s_1, s_2) \leftarrow l_0.\text{mcreate} \ a; \text{return} \ ((s_1, s_2), a) \end{array} \right.
\]

\[
= \left[ \begin{array}{l}
\text{Definition} \\
\text{Monad unit} \\
\text{Monad unit} \\
\text{Definition} \\
\end{array} \right]
\]

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

\[
l_0.\text{mget} \ (s_1, s_2) = l_2.\text{mget} \ s_2
\]

\[
\text{do} \left\{ \begin{array}{l}
(s'_1, s'_2) \leftarrow l_0.\text{mput} \ (s_1, s_2) \ a; \text{return} \ s'_2 \end{array} \right.
\]

\[
= l_2.\text{mput} \ s_2 \ a
\]

\[
\text{do} \left\{ \begin{array}{l}
(s_1, s_2) \leftarrow l_0.\text{mcreate} \ a; \text{return} \ s_2 \end{array} \right.
\]

\[
= l_2.\text{mcreate} \ a
\]

For the \( \text{mget} \) equation:

\[
\text{notice that since } | (s_1, s_2) \in R |, \text{we have}
\]

\[
l_0.\text{mget} \ (s_1, s_2)
\]

\[
= \left[ \begin{array}{l}
\text{Definition} \\
\text{Definition of lens composition} \\
\text{Definition of lens composition} \\
\text{Definition of lens composition} \\
\end{array} \right]
\]

\[
l_2.\text{mget} \ s_2
\]

\[
= \left[ \begin{array}{l}
\text{Definition} \\
\text{Definition} \\
\text{Definition of lens composition} \\
\text{Definition of lens composition} \\
\end{array} \right]
\]

\[
\text{do} \left\{ \begin{array}{l}
(s'_1, s'_2) \leftarrow l_1.\text{mput} \ s_1 \ a; \ (s'_1, s'_2) \leftarrow \text{return} \ (s'_1, l \cdot \text{get} \ s'_1); \text{return} \ s'_2 \end{array} \right.
\]

\[
= \left[ \begin{array}{l}
\text{Definition} \\
\text{Monad laws} \\
\text{Monad laws} \\
\text{Monad laws} \\
\end{array} \right]
\]

\[
\text{do} \left\{ \begin{array}{l}
(s'_1, s'_2) \leftarrow l_1.\text{mput} \ s_1 \ a; \ (s'_1, s'_2) \leftarrow \text{return} \ (l \cdot \text{get} \ s'_1); \text{return} \ s'_2 \end{array} \right.
\]

\[
= \left[ \begin{array}{l}
\text{Definition} \\
\text{Monad laws} \\
\text{Monad laws} \\
\end{array} \right]
\]

\[
\text{do} \left\{ \begin{array}{l}
(s'_1, s'_2) \leftarrow l_1.\text{mput} \ s_1 \ a; \ (s'_1, s'_2) \leftarrow \text{return} \ (l \cdot \text{get} \ s'_1); \text{return} \ (l \cdot \text{get} \ s'_1) \end{array} \right.
\]

\[
= \left[ \begin{array}{l}
\text{Definition} \\
\text{Monad laws} \\
\end{array} \right]
\]
Proof. We list the properties that must hold by virtue of this bisimulation for any
(sp1, sp2) ∈ R:

\[
\begin{align*}
& l_0.\text{get} (s_1, s_2) = l_1.\text{get} s_1 & l_0.\text{get} (s_1, s_2) = l_2.\text{get} s_2 \\
& \text{fst} (l_0.\text{put} (s_1, s_2) a) = l_1.\text{put} s_1 a & \text{snd} (l_0.\text{put} (s_1, s_2) a) = l_2.\text{put} s_2 a \\
& \text{fst} (l_0.\text{create} a) = l_1.\text{create} s_1 & \text{snd} (l_0.\text{create} a) = l_2.\text{create} a \\
& r_0.\text{get} (s_1, s_2) = r_1.\text{get} s_1 & r_0.\text{get} (s_1, s_2) = r_2.\text{get} s_2 \\
& \text{fst} (r_0.\text{put} (s_1, s_2) b) = r_1.\text{put} s_1 b & \text{snd} (r_0.\text{put} (s_1, s_2) b) = r_2.\text{put} s_2 b \\
& \text{fst} (r_0.\text{create} b) = r_1.\text{create} s_1 & \text{snd} (r_0.\text{create} b) = r_2.\text{create} b
\end{align*}
\]

In addition, it follows that:

\[
\begin{align*}
& l_0.\text{put} (s_1, s_2) a = (l_1.\text{put} s_1 a, l_2.\text{put} s_2 a) \in R \\
& r_0.\text{put} (s_1, s_2) b = (r_1.\text{put} s_1 b, r_2.\text{put} s_2 b) \in R
\end{align*}
\]

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

\[\Box\]

**Theorem 4.9.** Given \(sp_1 :: A \rightsquigarrow S_1 \rightsquigarrow B, sp_2 :: A \rightsquigarrow S_2 \rightsquigarrow B\), if \(sp_1 \equiv_b sp_2\) then \(sp_1 \equiv_s sp_2\).

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

\[
\begin{align*}
& l_0.\text{get} (s_1, s_2) = l_1.\text{get} s_1 & l_0.\text{get} (s_1, s_2) = l_2.\text{get} s_2 \\
& \text{fst} (l_0.\text{put} (s_1, s_2) a) = l_1.\text{put} s_1 a & \text{snd} (l_0.\text{put} (s_1, s_2) a) = l_2.\text{put} s_2 a \\
& \text{fst} (l_0.\text{create} a) = l_1.\text{create} s_1 & \text{snd} (l_0.\text{create} a) = l_2.\text{create} a \\
& r_0.\text{get} (s_1, s_2) = r_1.\text{get} s_1 & r_0.\text{get} (s_1, s_2) = r_2.\text{get} s_2 \\
& \text{fst} (r_0.\text{put} (s_1, s_2) b) = r_1.\text{put} s_1 b & \text{snd} (r_0.\text{put} (s_1, s_2) b) = r_2.\text{put} s_2 b \\
& \text{fst} (r_0.\text{create} b) = r_1.\text{create} s_1 & \text{snd} (r_0.\text{create} b) = r_2.\text{create} b
\end{align*}
\]
\( l_0.\text{create} a = (l_1.\text{create} a, l_2.\text{create} a) \in R \)
\( r_0.\text{create} b = (r_1.\text{create} b, r_2.\text{create} b) \in R \)

which also implies the following 'twist' equations:

\[
\begin{align*}
    r_1.\text{get} (l_1.\text{put} s_1 a) &= r_0.\text{get} (l_1.\text{put} s_1 a, l_2.\text{put} s_2 a) = r_2.\text{get} (l_2.\text{put} s_2 a), \\
    l_1.\text{get} (r_1.\text{put} s_1 b) &= l_0.\text{get} (r_1.\text{put} s_1 b, r_2.\text{put} s_2 b) = l_2.\text{get} (r_2.\text{put} s_2 b), \\
    r_1.\text{get} (l_1.\text{create} a) &= r_0.\text{get} (l_1.\text{create} a, l_2.\text{create} a) = r_2.\text{get} (l_2.\text{create} a), \\
    l_1.\text{get} (r_1.\text{create} a) &= l_0.\text{get} (r_1.\text{create} a, r_2.\text{create} a) = l_2.\text{get} (r_2.\text{create} a).
\end{align*}
\]

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

\[
\begin{align*}
    l.\text{get} &= \text{fst} \\
    l.\text{put} (s_1, s_2) s'_1 &= l_0.\text{put} (s_1, s_2) (l_1.\text{get} s'_1) \\
    l.\text{create} s_1 &= l_0.\text{create} (l_1.\text{get} s_1)
\end{align*}
\]

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 \( \text{create} \) and \( \text{put} \) 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 \( \text{get} \):

\[
(l; l_1).\text{get} (s_1, s_2)
\]

\[
\begin{align*}
&= \llbracket \text{definition} \rrbracket \\
&= l_1.\text{get} (l.\text{get} (s_1, s_2)) \\
&= \llbracket \text{definition of } l.\text{get} = \text{fst}, \text{fst commutes with } \text{get} \rrbracket \\
&= l_0.\text{get} (s_1, s_2) \\
&= \llbracket \text{reverse reasoning} \rrbracket \\
&= (r; l_2).\text{get} (s_1, s_2)
\end{align*}
\]

For \( \text{put} \), we have:

\[
(l; l_1).\text{put} (s_1, s_2) a
\]

\[
\begin{align*}
&= \llbracket \text{Definition} \rrbracket \\
&= l.\text{put} (s_1, s_2) (l_1.\text{put} s_1 a) \\
&= \llbracket \text{Definition} \rrbracket \\
&= l_0.\text{put} (s_1, s_2) (l_1.\text{get} (l_1.\text{put} s_1 a)) \\
&= \llbracket \text{(PutGet) for } l_1 \rrbracket \\
&= l_0.\text{put} (s_1, s_2) a \\
&= \llbracket \text{(PutGet) for } l_2 \rrbracket \\
&= l_0.\text{put} (s_1, s_2) (l_2.\text{get} (l_2.\text{put} s_2 a)) \\
&= \llbracket \text{Definition} \rrbracket \\
&= r.\text{put} (s_1, s_2) (l_2.\text{put} s_2 a) \\
&= \llbracket \text{Definition} \rrbracket \\
&= (r; l_2).\text{put} (s_1, s_2) a
\end{align*}
\]

Finally, for \( \text{create} \) we have:

\[
\]
Reflections on monadic lenses

\[
(l; l_1).\text{create } a
\]
\[
= \quad \text{Definition}
\]
\[
\text{create } (l_1, \text{create } a)
\]
\[
= \quad \text{Definition}
\]
\[
l_0.\text{create } (l_1.\text{get } (l_1, \text{create } a))
\]
\[
= \quad \text{Definition}
\]
\[
l_0.\text{create } a
\]
\[
l_0.\text{create } (l_1.\text{get } (l_2, \text{create } a))
\]
\[
= \quad \text{Definition}
\]
\[
r.\text{create } (l_2, \text{create } a)
\]
\[
= \quad \text{Definition}
\]
\[
(r; l_2).\text{create } a
\]

Next, we show that \(l; r_1 = r; r_2\). For \text{get}:

\[
(l; r_1).\text{get } (s_1, s_2)
\]
\[
= \quad \text{Definition}
\]
\[
r_1.\text{get } (l.\text{get } (s_1, s_2))
\]
\[
= \quad \text{Definition}
\]
\[
r_0.\text{get } (s_1, s_2)
\]
\[
= \quad \text{Definition}
\]
\[
(r; r_2).\text{get } (s_1, s_2)
\]

For \text{put}, we have:

\[
(l; r_1).\text{put } (s_1, s_2) b
\]
\[
= \quad \text{Definition}
\]
\[
l.\text{put } (s_1, s_2) (r_1.\text{put } s_1 b)
\]
\[
= \quad \text{Definition}
\]
\[
l_0.\text{put } (s_1, s_2) (l_1.\text{get } (r_1.\text{put } s_1 b))
\]
\[
= \quad \text{Definition}
\]
\[
l_0.\text{put } (s_1, s_2) (l_2.\text{get } (r_2.\text{put } s_2 b))
\]
\[
= \quad \text{Definition}
\]
\[
r.\text{put } (s_1, s_2) (r_2.\text{put } s_2 b)
\]
\[
= \quad \text{Definition}
\]
\[
(r; r_2).\text{put } (s_1, s_2) b
\]

Finally, for \text{create} we have:

\[
(l; r_1).\text{create } b
\]
\[
= \quad \text{Definition}
\]
\[
l.\text{create } (r_1, \text{create } b)
\]
\[
= \quad \text{Definition}
\]
\[
l_0.\text{create } (l_1.\text{get } (r_1.\text{create } b))
\]
\[
= \quad \text{Definition}
\]
\[
l_0.\text{create } (l_2.\text{get } (r_2.\text{create } b))
\]
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):

\[
\text{\texttt{\textbf{l}.get}} \left( \text{\texttt{l}.put} \left( s_1, s_2 \right) s'_1 \right) \\
= \quad \text{\texttt{\textbf{\textit{Definition} \[\]}}}
\quad \text{\texttt{\textit{r}.create \ (r}.create \ b)} \\
= \quad \text{\texttt{\textit{Definition} \[\]}}
\quad \text{\texttt{(r};r_2).create \ b}
\]

For (PutGet):

\[
\text{\texttt{\textbf{l}.put}} \left( s_1, s_2 \right) \left( \text{\texttt{l}.get} \left( s_1, s_2 \right) \right) \\
= \quad \text{\texttt{\textit{Definition} \[\]}}
\quad \text{\texttt{l}_0.\text{put}} \left( s_1, s_2 \right) \left( l_1.\text{get} \ s_1 \right) \\
= \quad \text{\texttt{\textit{Eta-expansion for pairs} \[\]}}
\quad \text{\texttt{(fst \ (l}_0.\text{put} \left( s_1, s_2 \right) \left( l_1.\text{get} \ s_1 \right)), snd \ (l}_0.\text{put} \left( s_1, s_2 \right) \left( l_1.\text{get} \ s_1 \right))}}
\]

For (CreateGet):

\[
\text{\texttt{\textbf{l}.create}} \left( \text{\texttt{l}.get} \left( s_1, s_2 \right) \right) \\
= \quad \text{\texttt{\textit{Definition} \[\]}}
\quad \text{\texttt{l}_0.\text{create}} \left( l_1.\text{get} \ s_1 \right) \\
= \quad \text{\texttt{\textit{Eta-expansion for pairs} \[\]}}
\quad \text{\texttt{(fst \ (l}_0.\text{create} \ left( l_1.\text{get} \ s_1 \right)), snd \ (l}_0.\text{create} \ left( l_1.\text{get} \ s_1 \right))}}
\]

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

To conclude, \( sp = \left( l, r \right) \) constitutes a span of lenses witnessing that \( sp_1 \equiv sp_2 \). □