Entangled State Monads

Extended abstract

James Cheney, James McKinna, Perdita Stevens
School of Informatics, University of Edinburgh
firstname.lastname@ed.ac.uk

Jeremy Gibbons, Faris Abou-Saleh
Dept. of Computer Science, University of Oxford
firstname.lastname@cs.ox.ac.uk

ABSTRACT

We present a monadic treatment of symmetric state-based bidirectional transformations, and show how it arises naturally from the well-known asymmetric lens-based account. We introduce two presentations of a concept we dub the "entangled" state monad, and prove their equivalence. As a step towards a unifying account of bidirectionality in general, we exhibit existing classes of state-based approaches from the literature as instances of our new constructions. This extended abstract reports on work in progress.

1. INTRODUCTION

This extended abstract describes work in progress towards unifying approaches to formalising bidirectional transformations (bx). For purposes of this paper, a bx is a device for maintaining consistency between two or more information sources. In model driven development, such sources are usually models; for example, UML models of a system to be developed. Other artefacts treated with these techniques could include database tables, XML files, abstract syntax trees, code, etc. We use the (admittedly overloaded) term 'models' broadly to refer to any of these information sources.

There are multiple dimensions over which notions of bx vary. For example, they may operate on only two information sources, or several. They may insist that one source be a strict abstraction of the others (asymmetric case), or not (symmetric case).

Our main motivation is to lay foundations that we will later use to work towards a uniform, typed understanding of the extra information that is used by bx, besides the current states of the models that are to be synchronised. We begin in this paper with state-based bx, including those with explicit complement.

In formal semantics, stateful computations are often expressed in terms of *monads* [3], giving a unified account of impure side-effects in pure functional languages. They have since become an essential programming pattern in such languages [6], and we follow suit.

2. BACKGROUND

Monads for Effectful Functional Programming. The essential idea of monads in functional programming is to encapsulate a computation with side-effects, taking inputs of type A and returning a result of type B, as a function of type $A \to M\ B$, for a suitable type constructor M, known as a monad. Whereas inhabitants of the plain type A denote pure values, those of the monadic type $M\ A$ denote *computations*, which may incur computational effects before yielding a value of type A. For instance, one may describe non-deterministic computations of type $A \to B$ in terms of the *List* monad – i.e., as functions $A \to \mathit{List}\ B$, where each value $a : A$ is assigned a list of possible return values $[b_1, b_2, \ldots] : \mathit{List}\ B$. Monads can be used to capture side-effects, input/output, exceptions, probabilistic choice, and many other computational effects. In this paper we are concerned with computations which may depend on, and modify, various forms of mutable state; such computations are described by the *state* monad, as defined shortly.

More formally, a *monad* is a type constructor M equipped with the following structure of typed operations (parametric in A, B):

$$\begin{aligned}
&\mathsf{return} : A \to M\ A \\
&(\gg\!=) : M\ A \to (A \to M\ B) \to M\ B \\
&(\gg)\ \ : M\ A \to M\ B \to M\ B \\
&ma \gg mb = ma \gg\!= \lambda_ .\ mb
\end{aligned}$$

(We borrow the Haskell convention of writing an infix operator $\oplus$ in parentheses $(\oplus)$ in order to refer to it without arguments.) Here, the operation return simply returns its argument with no other effect. The 'bind' operation $ma \gg\!= f$ runs a computation ma returning an A, then runs a computation f, parameterized over A and returning a B, finally returning that B value. The definable operation 'sequence' $ma \gg mb$ is a special case of 'bind' in which the computation mb does not depend on the A value returned by ma.

We work in the equational theory of the λ-calculus, as is common when discussing monads in Haskell; our presentation is a special case of the general categorical treatment of monads. The monad operations are required to satisfy the following three equational laws. The first two assert that return is a left and right unit for the 'bind' operation and the third that 'bind' is associative. (As usual, λ-binding scope extends as far to the right as possible. In the third equation, a is not free in g.)

$$\begin{aligned}
&\mathsf{return}\ a \gg\!= f &&= f\ a \\
&ma \gg\!= \mathsf{return} &&= ma \\
&ma \gg\!= (\lambda a .\ (f\ a \gg\!= g)) &&= (ma \gg\!= f) \gg\!= g
\end{aligned}$$

As a corollary, 'sequential composition' $(\gg)$ is associative, with left unit $\mathsf{return}\ ()$.

The State Monad. A distinguished instance of the above concept is M_S, the *state monad on type S*, representing computations with access to a single updateable memory cell of type S. We define $M_S\ A = S \to A \times S$ so that a computation of type $A \to M_S\ B$ takes input $a : A$, *and then can query the (old) state $s : S$*, before returning *a new state* $s':S$ and a result $b:B$. The monadic operations of M_S are defined below. The return operation takes a value $a:A$ and produces a computation which, for any initial state $s:S$, returns the value a and leaves the state s untouched. The 'bind' operation $\gg=$ chains together two stateful computations, using the final state s' of the first computation as the initial state of the second.

$$\begin{array}{l}\mathsf{return}:A \to (S \to A \times S)\\ \mathsf{return}\ a = \lambda s\,.\,(a,s)\\ (\gg=):(S \to A\times S) \to (A \to (S \to B \times S)) \to (S \to B \times S)\\ ma \gg= f = \lambda s\,.\,\mathbf{let}\ (a,s') = ma\ s\ \mathbf{in}\ f\ a\ s'\end{array}$$

In addition to the generic operations return and $\gg=$, the state monad supports two operations *get*, *set*, to read and write the state:

$$\begin{array}{l}get:M_S\ S\\ get = \lambda s\,.\,(s,s)\\ set:S \to M_S\ ()\\ set\ s' = \lambda s\,.\,((),s')\end{array}$$

In general, one may characterise state monads with multiple memory cells in terms of an algebraic theory of reads and writes, with seven equations [4]. In the restricted setting of a single memory cell, the theory reduces to the following four equations:

$$\begin{array}{lll}(\mathrm{GG}) & get \gg= \lambda s\,.\,get \gg= \lambda s'\,.\,k\ s\ s' & = get \gg= \lambda s\,.\,k\ s\ s\\ (\mathrm{GS}) & get \gg= set & = \mathsf{return}\ ()\\ (\mathrm{SG}) & set\ s \gg get & = set\ s \gg \mathsf{return}\ s\\ (\mathrm{SS}) & set\ s \gg set\ s' & = set\ s'\end{array}$$

It is routine to verify that the above definitions of *get* and *set* satisfy these laws. However, in the algebraic perspective, one abstracts away from the specific concrete representation M_S and the corresponding implementations of *get* and *set*, and instead considers a 'state monad on S' abstractly to be *any* monad M equipped with the additional structure of *get* and *set* satisfying the above four laws.

Asymmetric lenses via the state monad. An asymmetric *lens* [1] between S and V consists of a pair l of functions, usually called 'get' and 'put', which we write as follows:

$$\begin{array}{l}l.get:S \to V\\ l.put:S \to V \to S\end{array}$$

The idea is that S and V represent *source* and *view* data, *e.g.* in a database; V is derived from S using $l.get$, and $l.put$ computes a modified S on the basis of an old S and an updated V.

Given such a lens l, the state monad M_S admits computations get_l, set_l, where set_l takes input from V, updates the state S, and returns void; and get_l is the trivially stateful operation that queries but doesn't change the state S, and returns the V view of it:

$$\begin{array}{ll}get_l & :\ M_S\ V\\ get_l & =\lambda s\,.\,(l.get\ s,s)\\ set_l & :\ V \to M_S\ ()\\ set_l\ v & = \lambda s\,.\,((),l.put\ s\ v)\end{array}$$

These computations do not allow us to observe, or update, the underlying state S, except via the view type V. But viewed as abstract operations relative to an arbitrary monad M, the structure

$$\begin{array}{l}get_l:M\ V\\ set_l:V \to M\ ()\end{array}$$

defines a state monad on V, provided that the equational laws hold.

In the special case of the *identity* lens $l = id$, between S and S, where $id.get$ just reads the state, and $id.set$ updates it, we have:

$$\begin{array}{l}get_{id} = \lambda s\,.\,(s,s)\\ set_{id}\ s' = \lambda s\,.\,((),s')\end{array}$$

i.e. we obtain the state monad structure (M_S, get, set) on S.

Thus, an asymmetric lens l gives rise to *two* distinct state monad structures, one on V derived from l, the other on S corresponding to the special case id. Each accesses the *same* underlying state; we say the two structures are *entangled*. In the rest of this paper, we consider such entangled state monads in general. The generalisation turns out to be both simple and powerful: several other bx formalisms are instances of this notion, corresponding to monads which present two updateable views of some shared, possibly hidden, state. In the next section we give details of the generalisation. We revisit the discussion of asymmetric (and other) lenses, in more detail, in Section 4.

3. ENTANGLED STATE MONADS

We now show that a monad that exhibits the structure of a state monad in two ways is essentially a bidirectional transformation. We do this by introducing two definitions, those of 'set-bx' (corresponding directly to state monads) and 'put-bx' (corresponding more closely to symmetric lenses) and showing that they are equivalent. (The proofs are included in an extended paper currently in preparation.) We use the umbrella term 'entangled state monad' for these two formulations.

3.1 Set-bx

Given types A, B, we define a *set-bx between* A *and* B to be a monad M, equipped with four operations:

$$\begin{array}{l}get_A:M\ A\\ get_B:M\ B\\ set_A:A \to M\ ()\\ set_B:B \to M\ ()\end{array}$$

that satisfy the three laws for get_A and set_A

$$\begin{array}{lll}(\mathrm{GG}) & get_A \gg= \lambda s\,.\,get_A \gg= \lambda s'\,.\,k\ s\ s' & \\ & & = get_A \gg= \lambda s\,.\,k\ s\ s\\ (\mathrm{GS}) & get_A \gg= set_A & = \mathsf{return}\ ()\\ (\mathrm{SG}) & set_A\ a \gg get_A & = set_A\ a \gg \mathsf{return}\ a\end{array}$$

and symmetrically for get_B and set_B. A set-bx that in addition satisfies the following:

$$(\mathrm{SS})\quad set_A\ a \gg set_A\ a' = set_A\ a'$$

(and symmetrically in B) is called *overwriteable*.

We write $(get_A, get_B, set_A, set_B):A \overset{M}{\Longleftrightarrow} B$ to indicate that M is a set-bx between A and B equipped with operations get_A, etc. When discussing more than one such structure, we write $t:A \overset{M}{\Longleftrightarrow} B$ and $t.get_A$ and so on for the operations of t.

3.2 Put-bx

Given types A, B, we define a *put-bx between* A *and* B to be a monad M, equipped with four operations:

$$\begin{array}{l}get_A:M\ A\\ get_B:M\ B\\ put_A^B:A \to M\ B\\ put_B^A:B \to M\ A\end{array}$$

satisfying the following laws:

$$\begin{array}{ll}(\mathrm{GG}) & get \gg= \lambda s\,.\,get \gg= \lambda s'\,.\,k\ s\ s' = get \gg= \lambda s\,.\,k\ s\ s\\ (\mathrm{GP}) & get_A \gg= put_A^B\ = get_B\\ (\mathrm{PG}_1) & put_A^B\ a \gg get_A = put_A^B\ a \gg \mathsf{return}\ a\\ (\mathrm{PG}_2) & put_A^B\ a \gg get_B = put_A^B\ a\end{array}$$

(and symmetrically, swapping A and B).

A put-bx that in addition satisfies the following:

$$\text{(PP)} \quad \mathit{put}_A^B\ a \gg \mathit{put}_A^B\ a' = \mathit{put}_A^B\ a'$$

(and symmetrically in B) is called *overwriteable*.

As above, we write $(\mathit{get}_A, \mathit{get}_B, \mathit{put}_A^B, \mathit{put}_B^A) : A \overset{M}{\Longleftrightarrow} B$ to indicate that M is a put-bx with operations get_A, etc., and write $t : A \overset{M}{\Longleftrightarrow} B$, $t.\mathit{get}_A$ and so on when discussing more than one such structure.

3.3 Relating set-bx and put-bx

We will show that set-bx and put-bx are equivalent in the following sense: for each set-bx $t : A \overset{M}{\Longleftrightarrow} B$ we can construct a put-bx $\mathit{set2pp}(t) : A \overset{M}{\Longleftrightarrow} B$ and for each put-bx $u : A \overset{M}{\Longleftrightarrow} B$ we can construct a set-bx $\mathit{pp2set}(u) : A \overset{M}{\Longleftrightarrow} B$. Moreover, the two constructions are inverses: $\mathit{pp2set}(\mathit{set2pp}(t)) = t$ and $\mathit{set2pp}(\mathit{pp2set}(u)) = u$. This means that any equation satisfied by all set-bx translates to an equation that holds for all put-bx, and vice versa. So, we can work with set-bx or put-bx as convenient, justifying our overloaded notation $t : A \overset{M}{\Longleftrightarrow} B$.

The translations are defined as follows. Given set-bx $t : A \overset{M}{\Longleftrightarrow} B$, define put-bx $\mathit{set2pp}(t)$ by:

$$\begin{aligned}
\mathit{set2pp}(t).\mathit{get}_A &= t.\mathit{get}_A \\
\mathit{set2pp}(t).\mathit{get}_B &= t.\mathit{get}_B \\
\mathit{set2pp}(t).\mathit{put}_A^B\ a &= t.\mathit{set}_A\ a \gg t.\mathit{get}_B \\
\mathit{set2pp}(t).\mathit{put}_B^A\ b &= t.\mathit{set}_B\ b \gg t.\mathit{get}_A
\end{aligned}$$

Likewise, given put-bx $u : A \overset{M}{\Longleftrightarrow} B$, we define set-bx $\mathit{pp2set}(u)$ as follows:

$$\begin{aligned}
\mathit{pp2set}(u).\mathit{get}_A &= u.\mathit{get}_A \\
\mathit{pp2set}(u).\mathit{get}_B &= u.\mathit{get}_B \\
\mathit{pp2set}(u).\mathit{set}_A\ a &= u.\mathit{put}_A^B\ a \gg \mathsf{return}\ () \\
\mathit{pp2set}(u).\mathit{set}_B\ b &= u.\mathit{put}_B^A\ b \gg \mathsf{return}\ ()
\end{aligned}$$

LEMMA 1. *If $t : A \overset{M}{\Longleftrightarrow} B$ is an (overwriteable) set-bx then $\mathit{set2pp}(t) : A \overset{M}{\Longleftrightarrow} B$ is an (overwriteable) put-bx.*

LEMMA 2. *If $u : A \overset{M}{\Longleftrightarrow} B$ is an (overwriteable) put-bx then $\mathit{pp2set}(u) : A \overset{M}{\Longleftrightarrow} B$ is an (overwriteable) set-bx.*

LEMMA 3. *Translations $\mathit{pp2set}(\cdot)$ and $\mathit{set2pp}(\cdot)$ are inverses.*

3.4 Entanglement

Note that the state monad on pairs $M_{A\times B}$ determines a set-bx, with

$$\begin{aligned}
\mathit{get}_A &= \mathit{get} \gg\!= \lambda(a,_)\ .\ \mathsf{return}\ a \\
\mathit{get}_B &= \mathit{get} \gg\!= \lambda(_,b)\ .\ \mathsf{return}\ b \\
\mathit{set}_A\ a &= \mathit{get} \gg\!= \lambda(_,b)\ .\ \mathit{set}\ (a,b) \\
\mathit{set}_B\ b &= \mathit{get} \gg\!= \lambda(a,_)\ .\ \mathit{set}\ (a,b)
\end{aligned}$$

However, this structure also satisfies stronger laws than our definitions require; in particular, commutativity of *set*s:

$$\mathit{set}_A\ a \gg \mathit{set}_B\ b = \mathit{set}_B\ b \gg \mathit{set}_A\ a$$

This law is not required of a set-bx; it is consistent with the set-bx laws that the A and B components of the state be "entangled", in the sense that setting one component also changes the other to restore consistency; in other words, that set_A and set_B need not commute. The monad $M_{A\times B}$ arises simply as a special case of our general analysis of algebraic bx in Section 4 below, in which the consistency relation is universally true: set_A automatically restores consistency without the need to change B and vice versa.

4. INSTANCES

In this section we justify our view that set-bx (and hence also put-bx) structures are a general form of state-based bx, by showing how they capture the usual presentations such as asymmetric and symmetric lenses. Even though symmetric lenses subsume asymmetric lenses and algebraic bx, it is instructive to start with the simpler cases. We also give a simple example of a stateful bx that is not (isomorphic to) a symmetric lens. Investigation of other instances, and their relationships, is ongoing work.

Asymmetric lenses. Let $l : A \rightleftharpoons B$ be a classic asymmetric lens, i.e. $l.\mathit{get} : A \to B$ and $l.\mathit{put} : A \to B \to A$. We may construct a set-bx $l : A \overset{M_A}{\Longleftrightarrow} B$ (where M_A is the state monad on state type A, as introduced in Section 2 above) as follows:

$$\begin{aligned}
\mathit{get}_A &= \lambda a\ .\ (a,a) \\
\mathit{get}_B &= \lambda a\ .\ (l.\mathit{get}\ a, a) \\
\mathit{set}_A\ a' &= \lambda a\ .\ ((), a') \\
\mathit{set}_B\ b' &= \lambda a\ .\ ((), l.\mathit{put}\ a\ b')
\end{aligned}$$

If l is a so-called 'well-behaved' lens, then it also satisfies:

$$\begin{aligned}
&\text{(GetPut)} \quad l.\mathit{put}\ a\ (l.\mathit{get}\ a) = a \\
&\text{(PutGet)} \quad l.\mathit{get}\ (l.\mathit{put}\ a\ b) = b
\end{aligned}$$

Finally, an asymmetric lens may optionally satisfy:

$$\text{(PutPut)} \quad l.\mathit{put}\ (l.\mathit{put}\ a\ b)\ b' = l.\mathit{put}\ a\ b'$$

in which case it is called *very well-behaved*.

LEMMA 4. *If the asymmetric lens $l : A \rightleftharpoons B$ is well-behaved, then the above definitions indeed make $l : A \overset{M_A}{\Longleftrightarrow} B$ into a set-bx. If l is very well-behaved, then $l : A \overset{M_A}{\Longleftrightarrow} B$ is also overwriteable.*

Algebraic bxs. Let $(R, \overrightarrow{R}, \overleftarrow{R})$ be an algebraic bx $A \leftrightarrow B$ in the style of Stevens [5], *i.e.*, $R \subseteq A \times B$, $\overrightarrow{R} : A \times B \to B$, $\overleftarrow{R} : A \times B \to A$, satisfying the conditions

$$\begin{aligned}
&\text{(Correct)} \quad (a, \overrightarrow{R}(a,b)) \in R \\
&\text{(Hippocratic)} \quad R(a,b) \Rightarrow \overrightarrow{R}(a,b) = b
\end{aligned}$$

and symmetrically for $\overleftarrow{R}$. We say R is *history-ignorant* if it also satisfies

$$\text{(HI)} \quad \overrightarrow{R}(a, \overrightarrow{R}(a',b)) = \overrightarrow{R}(a,b)$$

and symmetrically for $\overleftarrow{R}$.

Let M_R be the state monad over R, viewing R as a set of pairs, $R \subseteq A \times B$. Then we define the following operations:

$$\begin{aligned}
\mathit{get}_A &= \lambda(a,b)\ .\ (a,(a,b)) \\
\mathit{get}_B &= \lambda(a,b)\ .\ (b,(a,b)) \\
\mathit{set}_A\ a' &= \lambda(a,b)\ .\ ((),(a', \overrightarrow{R}(a',b))) \\
\mathit{set}_B\ b' &= \lambda(a,b)\ .\ ((),(\overleftarrow{R}(a,b'),b'))
\end{aligned}$$

The condition (Correct) ensures that $\mathit{set}_A\ a'$ and $\mathit{set}_B\ b'$ are well-defined functions $R \to () \times R$, and thus preserve the consistency of pairs $(a,b) \in R$.

LEMMA 5. *For any algebraic bx $(R, \overrightarrow{R}, \overleftarrow{R})$, the above operations make M_R into a set-bx. If $(R, \overrightarrow{R}, \overleftarrow{R})$ is history-ignorant, then M_R is also overwriteable.*

Symmetric lenses. Let $l : A \overset{C}{\longleftrightarrow} B$ be a symmetric lens as presented by Hofmann et al. [2]. That is, let $l = (putl, putr)$ consist of a pair of functions

$$\begin{aligned} &putl : A \times C \to B \times C \\ &putr : B \times C \to A \times C \end{aligned}$$

which satisfy

$$\begin{aligned} &\text{(PutRL)} \quad putr\ (a,c) = (b,c') \Rightarrow putl\ (b,c') = (a,c') \\ &\text{(PutLR)} \quad putl\ (b,c) = (a,c') \Rightarrow putr\ (a,c') = (b,c') \end{aligned}$$

Let M_l be the state monad M_T over the set T of *consistent* states in $A \times B \times C$, *i.e.*, those triples $(a,b,c) \in A \times B \times C$ satisfying

$$putr\ (a,c) = (b,c) \quad \text{and} \quad putl\ (b,c) = (a,c)$$

Then define the following operations for M_l:

$$\begin{aligned} &get_A \quad = \lambda(a,b,c)\,.\,(a,(a,b,c)) \\ &get_B \quad = \lambda(a,b,c)\,.\,(b,(a,b,c)) \\ &put_A^B\,a' = \lambda(a,b,c)\,.\,\mathbf{let}\,(b',c') = putr\,(a',c)\,\mathbf{in}\,(b',(a',b',c')) \\ &put_B^A\,b' = \lambda(a,b,c)\,.\,\mathbf{let}\,(a',c') = putl\,(b',c)\,\mathbf{in}\,(a',(a',b',c')) \end{aligned}$$

We need to show that these operations are well defined in the sense that they preserve consistency of the state, and this is where we need the symmetric lens laws – once this is done, it is easy to see that these definitions satisfy the put-bx laws.

LEMMA 6. *Given any symmetric lens $l = (putl, putr) : A \overset{C}{\longleftrightarrow} B$, the above operations are well-defined and make M_l into a put-bx.*

Stateful bx. We now consider an example that performs I/O side-effects, and thus by definition cannot be a symmetric lens (or any of the other bx mentioned above). We define a monad M that combines stateful updates (just on integer states, for simplicity) with Haskell-style monadic I/O; the latter is captured via a monad IO and an operation $print : String \to IO\ ()$. The return and $\gg\!=$ operations of M are therefore defined in terms of those of IO, so to be explicit we use subscripts below to disambiguate.

$$\begin{aligned} &M\ A = Integer \to IO\ (A, Integer) \\ &\mathsf{return}_M\ x\ = \lambda s\,.\,\mathsf{return}_{IO}\ (x,s) \\ &ma \gg\!=_M f = \lambda s\,.\,ma\ s \gg\!=_{IO} \lambda(a,s')\,.\,f\ a\ s' \\ &get_A \quad = \lambda s\,.\,\mathsf{return}_{IO}\ (s,s) \\ &get_B \quad = \lambda s\,.\,\mathsf{return}_{IO}\ (s,s) \\ &set_A\ a = \lambda s\,.\,(\mathbf{if}\ a \not\equiv s \\ &\qquad\qquad \mathbf{then}\ print\ \texttt{"Changed A"} \\ &\qquad\qquad \mathbf{else}\ \mathsf{return}_{IO}\ ()) \gg_{IO} \mathsf{return}_{IO}\ ((),a) \\ &set_B\ b = \lambda s\,.\,(\mathbf{if}\ b \not\equiv s \\ &\qquad\qquad \mathbf{then}\ print\ \texttt{"Changed B"} \\ &\qquad\qquad \mathbf{else}\ \mathsf{return}_{IO}\ ()) \gg_{IO} \mathsf{return}_{IO}\ ((),b) \end{aligned}$$

That is, a computation in monad M yielding a result of type A amounts to an IO-computation yielding a pair of an A and a new *Integer* state, given as input an old *Integer* state. This is a set-bx: in particular, its behaviour satisfies the laws (GG), (GS) and (SG). Its *set* operations are side-effecting, but the side-effects only occur when the state is changed. For simplicity, we have taken the underlying bidirectional transformation to be trivial, but we should be able to add similar stateful behaviour to any (symmetric) lens or algebraic bx following a similar pattern.

5. CONCLUSIONS

Lenses are traditionally presented asymmetrically, whereas many bx applications such as model synchronisation are entirely symmetric. Symmetric lenses [2] and algebraic bx [5] cover the more general symmetric case, but both formulations go beyond equational logic. We have shown a very simple *equational* characterisation that unifies lenses, symmetric lenses, and algebraic bx, by a natural generalisation of the 'get' and 'set' operations of the state monad. Interestingly, the notions of *consistency* for algebraic bx and *complement* disappear into the hidden state of the monad. We expect to be able to accommodate bx with richer complements or witness structures in the same way. Moreover, our approach offers the possibility of generalisation to reconcile effects such as I/O, nondeterminism, exceptions, or probabilistic choice with bidirectionality, drawing on the rich theory of monads, and possibly leading to a theory of *bidirectional programming with effects*.

This is work in progress. We are currently investigating the central issues of *equivalence* and *composition* of entangled state monads. Symmetric lenses are quotiented by an equivalence relation in order for properties such as associativity of composition to hold. We expect something similar to be needed for entangled state monads. Indeed, the question of whether entangled state monads can be composed seems nontrivial; some restrictions on the class of monads considered may be necessary for composability.

We have considered entangled state monads only in relatively standard settings, such as the category of sets and functions (in the guise of Haskell types and functions). Another interesting direction may be to explore other settings, such as partial orders, metric spaces, or topologies, which may offer insights into notions of least change or predictable behaviour.

Acknowledgements

We thank the participants at the Banff Bx workshop, and Benjamin Pierce, for helpful comments, as well as the anonymous reviewers for their generous and thoughtful remarks. The work is partly supported by EPSRC grants EP/K020218/1 and EP/K020919/1.

6. REFERENCES

[1] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, May 2007.

[2] M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Austin, Texas*, Jan. 2011.

[3] E. Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23. IEEE Computer Society, 1989.

[4] G. D. Plotkin and J. Power. Notions of computation determine monads. In *FoSSaCS*, pages 342–356, 2002.

[5] P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. *Journal of Software and Systems Modeling (SoSyM)*, 9(1):7–20, 2010.

[6] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.